The contents of this manual describe the product, BDS-PROLOG for Windows (hereinafter referred to as WIN-PROLOG), version 7.0, and are believed correct at time of going to press. They do not embody a commitment on the part of Brian D Steel (BDS), who may from time to time make changes to the specification of the product, in line with his policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of BDS.

Copyright (c) 1989-2019 Brian D Steel

Designed and Written by Brian D Steel

The "wallpaper" used in the screen shots in this publication is based on the Willow Boughs design by William Morris (1834-96)

Brian D Steel  
Fairbanks  
Alfold Road  
Cranleigh  
Surrey  
GU6 8NB  
England

phone: +44 (0) 1483 274838  
mobile: +44 (0) 7774 211281  
email: prolog@solanum.org  
web: http://www.solanum.org

BDS-PROLOG and WIN-PROLOG are trademarks of Brian D Steel, Surrey, UK, and are marketed and supported by Logic Programming Associates (LPA); for info, sales and technical support, please use the following contact points:

phone: +44 (0) 20 8871 2016  
info: info_team@lpa.co.uk  
sales: sale_team@lpa.co.uk  
support: tech_team@lpa.co.uk  
web: http://www.lpa.co.uk

Welcome to WIN-PROLOG 7.0!
## Contents

**WIN-PROLOG Technical Reference**......................................................................................................................... 2

**Introduction**........................................................................................................................................................................ 29
  What's in WIN-PROLOG?.................................................................................................................................................. 29
  About this Manual ........................................................................................................................................................... 29

**The Technical Reference** .................................................................................................................................................. 30
  File Names......................................................................................................................................................................... 30
  Font and Layout Conventions ............................................................................................................................................ 31
  The "Text" Font ............................................................................................................................................................. 31
  The "Computer" Font .................................................................................................................................................... 31
  Special Fonts ................................................................................................................................................................. 32
  Predicate Entries .......................................................................................................................................................... 32
  functor/arity .................................................................................................................................................................. 33
  Predicate Heading ....................................................................................................................................................... 33
  Predicate Summary ..................................................................................................................................................... 33
  Argument Modes ........................................................................................................................................................ 33
  Argument Types .......................................................................................................................................................... 34
  Predicate Comments .................................................................................................................................................. 35
  Predicate Example ...................................................................................................................................................... 35

**Predicate Categories** .................................................................................................................................................. 36
  Arithmetic Predicates .................................................................................................................................................. 36
  Arithmetic Functions ................................................................................................................................................... 36
  Control Predicates ...................................................................................................................................................... 37
  Meta Programming Predicates ..................................................................................................................................... 37
  Solution Predicates ..................................................................................................................................................... 37
  List Processing Predicates ........................................................................................................................................... 37
  Term Manipulation Predicates ..................................................................................................................................... 38
  Term Comparison Predicates ....................................................................................................................................... 38
  Term Unification Predicates ......................................................................................................................................... 38
  Term Type Predicates ................................................................................................................................................ 38
  System Status Predicates .......................................................................................................................................... 38
  Operating System Predicates .................................................................................................................................... 39
  Date and Time Predicates ........................................................................................................................................... 39
  Stream Handling Predicates ........................................................................................................................................ 40
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Input/Output Predicates</td>
<td>40</td>
</tr>
<tr>
<td>Stream Data Processing Predicates</td>
<td>41</td>
</tr>
<tr>
<td>Console Input/Output Predicates</td>
<td>41</td>
</tr>
<tr>
<td>Operator Handling Predicates</td>
<td>41</td>
</tr>
<tr>
<td>Grammar Rule Predicates</td>
<td>42</td>
</tr>
<tr>
<td>Compiler Predicates</td>
<td>42</td>
</tr>
<tr>
<td>System and User Hooks</td>
<td>42</td>
</tr>
<tr>
<td>Debugging Predicates</td>
<td>43</td>
</tr>
<tr>
<td>Error, Timer and Message Predicates</td>
<td>43</td>
</tr>
<tr>
<td>Garbage Collection Predicates</td>
<td>43</td>
</tr>
<tr>
<td>Graphics Predicates</td>
<td>44</td>
</tr>
<tr>
<td>Printer Control Predicates</td>
<td>44</td>
</tr>
<tr>
<td>Menu Handling Predicates</td>
<td>45</td>
</tr>
<tr>
<td>Windows API and DLL Interface Predicates</td>
<td>45</td>
</tr>
<tr>
<td>Window Handling: Generic Predicates</td>
<td>45</td>
</tr>
<tr>
<td>Window Handling: Button Controls</td>
<td>45</td>
</tr>
<tr>
<td>Window Handling: Combobox Controls</td>
<td>46</td>
</tr>
<tr>
<td>Window Handling: Edit Controls</td>
<td>46</td>
</tr>
<tr>
<td>Window Handling: Listbox Controls</td>
<td>46</td>
</tr>
<tr>
<td>Window Handling: NudgeBar Controls</td>
<td>46</td>
</tr>
<tr>
<td>Window Handling: Rich Controls</td>
<td>46</td>
</tr>
<tr>
<td>Window Handling: Scrollbar Controls</td>
<td>47</td>
</tr>
<tr>
<td>Window Handling: StripBar Controls</td>
<td>47</td>
</tr>
<tr>
<td>Window Handling: TabBox Controls</td>
<td>47</td>
</tr>
<tr>
<td>Window Handling: TrackBar Controls</td>
<td>47</td>
</tr>
<tr>
<td>Window Handling: TreeView Controls</td>
<td>47</td>
</tr>
<tr>
<td>Window Handling: Dialogs</td>
<td>48</td>
</tr>
<tr>
<td>Window Handling: Environment Features</td>
<td>48</td>
</tr>
<tr>
<td>Windows Sockets (Winsock) Predicates</td>
<td>48</td>
</tr>
<tr>
<td>Musical Instrument Digital Interface (MIDI) Predicates</td>
<td>48</td>
</tr>
<tr>
<td>Undocumented Predicates</td>
<td>49</td>
</tr>
<tr>
<td>Alphabetical Reference</td>
<td>50</td>
</tr>
<tr>
<td>;/2</td>
<td>51</td>
</tr>
<tr>
<td>-&gt;/2</td>
<td>52</td>
</tr>
<tr>
<td>;/2</td>
<td>55</td>
</tr>
<tr>
<td>;/2</td>
<td>57</td>
</tr>
<tr>
<td>Procedure</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>assertz/1</td>
<td>113</td>
</tr>
<tr>
<td>at_end_of_file/0</td>
<td>115</td>
</tr>
<tr>
<td>at_end_of_line/0</td>
<td>116</td>
</tr>
<tr>
<td>atom/1</td>
<td>117</td>
</tr>
<tr>
<td>atom_chars/2</td>
<td>119</td>
</tr>
<tr>
<td>atom_string/2</td>
<td>120</td>
</tr>
<tr>
<td>atomic/1</td>
<td>121</td>
</tr>
<tr>
<td>attrib/2</td>
<td>122</td>
</tr>
<tr>
<td>bagof/3</td>
<td>123</td>
</tr>
<tr>
<td>bdsbox/2</td>
<td>126</td>
</tr>
<tr>
<td>beep/2</td>
<td>128</td>
</tr>
<tr>
<td>break/0</td>
<td>129</td>
</tr>
<tr>
<td>break/1</td>
<td>131</td>
</tr>
<tr>
<td>?BREAK?/1</td>
<td>132</td>
</tr>
<tr>
<td>break_hook/1</td>
<td>133</td>
</tr>
<tr>
<td>brsbox/2</td>
<td>134</td>
</tr>
<tr>
<td>busy/1</td>
<td>135</td>
</tr>
<tr>
<td>'C'/3</td>
<td>136</td>
</tr>
<tr>
<td>call/1</td>
<td>137</td>
</tr>
<tr>
<td>call/2</td>
<td>140</td>
</tr>
<tr>
<td>call_dialog/2</td>
<td>142</td>
</tr>
<tr>
<td>callable/1</td>
<td>144</td>
</tr>
<tr>
<td>cat/3</td>
<td>145</td>
</tr>
<tr>
<td>catch/2</td>
<td>146</td>
</tr>
<tr>
<td>catch/3</td>
<td>149</td>
</tr>
<tr>
<td>char/1</td>
<td>150</td>
</tr>
<tr>
<td>chars/1</td>
<td>151</td>
</tr>
<tr>
<td>chdir/1</td>
<td>152</td>
</tr>
<tr>
<td>clause/2</td>
<td>153</td>
</tr>
<tr>
<td>clause/3</td>
<td>155</td>
</tr>
<tr>
<td>clauses/2</td>
<td>157</td>
</tr>
<tr>
<td>close/1</td>
<td>158</td>
</tr>
<tr>
<td>cmp/3</td>
<td>159</td>
</tr>
<tr>
<td>compare/3</td>
<td>161</td>
</tr>
<tr>
<td>compile/1</td>
<td>163</td>
</tr>
<tr>
<td>compound/1</td>
<td>164</td>
</tr>
</tbody>
</table>
equiv/2 ................................................................. 217
eread/1 .................................................................. 219
eread/2 .................................................................. 220
ererrmsg/2 .............................................................. 221
?ERROR?/2 ............................................................. 222
error_hook/2 ........................................................... 223
error_message/2 ...................................................... 224
etoks/1 .................................................................. 225
etoks/2 .................................................................. 227
ewrite/1 .................................................................. 228
ewrite/2 .................................................................. 229
exec/3 .................................................................... 230
exit/1 ..................................................................... 231
expand_dcg/2 .......................................................... 233
expand_term/2 .......................................................... 234
fail/0 ..................................................................... 235
false/0 ................................................................... 236
fclose/1 .................................................................. 237
fcreate/5 ................................................................ 238
fdata/5 .................................................................. 242
fdict/2 .................................................................. 244
file/1 ..................................................................... 245
file/3 ..................................................................... 247
file_search_path/2 .................................................... 249
fileerrors/0 ............................................................. 251
find/3 ..................................................................... 252
findall/3 ................................................................ 255
flag/1 ..................................................................... 256
float/1 .................................................................... 257
fluff/3 ................................................................... 259
flush/0 .................................................................. 261
fname/4 .................................................................. 262
fntbox/3 ................................................................ 264
fnthdl/2 ................................................................ 266
fonts/1 .................................................................. 267
forall/2 .................................................................. 268
force/1 .......................................................... 270
fprint/2 .......................................................... 271
fread/4 .......................................................... 275
free/9 .............................................................. 280
functor/3 .......................................................... 281
fwrite/4 ............................................................ 282
garbage_collect/0 .................................................. 287
garbage_collect/1 .................................................. 288
gc/0 ................................................................. 289
gc/1 ................................................................. 290
get/1 ................................................................. 291
get0/1 ................................................................. 292
getb/1 ............................................................... 293
getx/2 ............................................................... 294
gfx/1 ............................................................... 295
gfx_back_close/1 .................................................... 296
gfx_back_create/4 .................................................. 297
gfx_back_dict/2 ..................................................... 298
gfx_back_handle/2 .................................................. 299
gfx_begin/1 ........................................................ 300
gfx_begin/3 ........................................................ 303
gfx_bitmap_close/1 .................................................. 306
gfx_bitmap_dict/2 .................................................. 307
gfx_bitmap_handle/2 ................................................ 308
gfx_bitmap_load/2 .................................................. 309
gfx_brush_close/1 .................................................... 310
gfx_brush_create/5 .................................................. 311
gfx_brush_dict/2 ..................................................... 312
gfx_brush_handle/2 .................................................. 313
gfx_cleanup/0 ........................................................ 314
gfx_clipping/4 ....................................................... 315
gfx_cursor_handle/2 ............................................... 317
gfx_end/1 ........................................................... 318
gfx_end/3 ........................................................... 320
gfx_font_close/1 ..................................................... 322
gfx_font_create/4 ................................................... 323
optimize/1 ................................................................. 479
optimize_files/1 .......................................................... 480
otherwise/0 .................................................................. 481
outpos/1 .................................................................. 482
output/1 .................................................................. 483
pause/1 .................................................................. 485
pdict/4 .................................................................. 486
phrase/2 .................................................................. 488
phrase/3 .................................................................. 490
popup_menu/4 ................................................................. 492
portray/1 .................................................................. 493
portray_clause/1 ............................................................ 494
portray_clause/2 ............................................................ 495
predicate_property/2 ....................................................... 496
prgbox/1 ................................................................ 501
prgbox/3 ................................................................ 503
print/1 ................................................................ 504
prnbox/4 ................................................................ 505
prnend/1 ................................................................ 507
prnini/4 ................................................................ 508
prnpag/1 ................................................................ 509
prnstt/1 ................................................................ 510
profile/1 ................................................................ 511
profile/4 ................................................................ 512
prolog_flag/2 ................................................................. 513
prolog_flag/3 ................................................................. 517
prolog_load_context/2 .................................................... 519
prompt/2 ................................................................ 520
prompts/2 ................................................................ 521
put/1 ................................................................ 522
putb/1 ................................................................ 523
putx/2 ................................................................ 524
read/1 ................................................................ 525
reconsult/1 ................................................................. 526
remove/3 ................................................................. 527
removeall/3 .............................................................. 527
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ren/2</td>
<td>528</td>
</tr>
<tr>
<td>repeat/0</td>
<td>529</td>
</tr>
<tr>
<td>repeat/1</td>
<td>530</td>
</tr>
<tr>
<td>repeat/2</td>
<td>531</td>
</tr>
<tr>
<td>repeat/3</td>
<td>532</td>
</tr>
<tr>
<td>retract/1</td>
<td>533</td>
</tr>
<tr>
<td>retract/2</td>
<td>536</td>
</tr>
<tr>
<td>retractall/1</td>
<td>539</td>
</tr>
<tr>
<td>reverse/2</td>
<td>541</td>
</tr>
<tr>
<td>reverse/3</td>
<td>542</td>
</tr>
<tr>
<td>rgbbox/4</td>
<td>543</td>
</tr>
<tr>
<td>rich_format/3</td>
<td>545</td>
</tr>
<tr>
<td>rich_print/5</td>
<td>548</td>
</tr>
<tr>
<td>rich_syntax/1</td>
<td>549</td>
</tr>
<tr>
<td>rich_syntax/2</td>
<td>550</td>
</tr>
<tr>
<td>rmdir/1</td>
<td>551</td>
</tr>
<tr>
<td>rpn/2</td>
<td>552</td>
</tr>
<tr>
<td>safe/1</td>
<td>555</td>
</tr>
<tr>
<td>savbox/5</td>
<td>556</td>
</tr>
<tr>
<td>save_predicates/2</td>
<td>558</td>
</tr>
<tr>
<td>scan/3</td>
<td>560</td>
</tr>
<tr>
<td>sckhd1/2</td>
<td>564</td>
</tr>
<tr>
<td>sclose/1</td>
<td>566</td>
</tr>
<tr>
<td>screate/2</td>
<td>568</td>
</tr>
<tr>
<td>sdict/2</td>
<td>570</td>
</tr>
<tr>
<td>see/1</td>
<td>571</td>
</tr>
<tr>
<td>seed/1</td>
<td>572</td>
</tr>
<tr>
<td>seeing/1</td>
<td>573</td>
</tr>
<tr>
<td>seen/0</td>
<td>574</td>
</tr>
<tr>
<td>setof/3</td>
<td>575</td>
</tr>
<tr>
<td>sha/3</td>
<td>578</td>
</tr>
<tr>
<td>show_dialog/1</td>
<td>580</td>
</tr>
<tr>
<td>simple/1</td>
<td>583</td>
</tr>
<tr>
<td>size_dialog/2</td>
<td>584</td>
</tr>
<tr>
<td>skip/1</td>
<td>586</td>
</tr>
<tr>
<td>skip_comments/0</td>
<td>587</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------</td>
<td>------</td>
</tr>
<tr>
<td>switch/2</td>
<td>644</td>
</tr>
<tr>
<td>swrite/1</td>
<td>645</td>
</tr>
<tr>
<td>swrite/2</td>
<td>646</td>
</tr>
<tr>
<td>sysops/0</td>
<td>647</td>
</tr>
<tr>
<td>system_menu/3</td>
<td>648</td>
</tr>
<tr>
<td>tab/1</td>
<td>649</td>
</tr>
<tr>
<td>tell/1</td>
<td>650</td>
</tr>
<tr>
<td>telling/1</td>
<td>651</td>
</tr>
<tr>
<td>term_expansion/2</td>
<td>652</td>
</tr>
<tr>
<td>throw/2</td>
<td>654</td>
</tr>
<tr>
<td>time/2</td>
<td>656</td>
</tr>
<tr>
<td>time/3</td>
<td>658</td>
</tr>
<tr>
<td>time/4</td>
<td>659</td>
</tr>
<tr>
<td>time/5</td>
<td>660</td>
</tr>
<tr>
<td>time/7</td>
<td>661</td>
</tr>
<tr>
<td>timer_close/1</td>
<td>663</td>
</tr>
<tr>
<td>timer_create/2</td>
<td>664</td>
</tr>
<tr>
<td>timer_data/2</td>
<td>666</td>
</tr>
<tr>
<td>timer_dict/2</td>
<td>667</td>
</tr>
<tr>
<td>timer_get/2</td>
<td>668</td>
</tr>
<tr>
<td>timer_set/2</td>
<td>669</td>
</tr>
<tr>
<td>tipbox/3</td>
<td>671</td>
</tr>
<tr>
<td>told/0</td>
<td>673</td>
</tr>
<tr>
<td>total/9</td>
<td>674</td>
</tr>
<tr>
<td>toteall/3</td>
<td>675</td>
</tr>
<tr>
<td>touch/0</td>
<td>677</td>
</tr>
<tr>
<td>touch/3</td>
<td>678</td>
</tr>
<tr>
<td>trace/0</td>
<td>681</td>
</tr>
<tr>
<td>true/0</td>
<td>682</td>
</tr>
<tr>
<td>ttyflush/0</td>
<td>684</td>
</tr>
<tr>
<td>ttyget/1</td>
<td>685</td>
</tr>
<tr>
<td>ttyget0/1</td>
<td>686</td>
</tr>
<tr>
<td>ttyin/0</td>
<td>687</td>
</tr>
<tr>
<td>ttyput/1</td>
<td>688</td>
</tr>
<tr>
<td>ttyskip/1</td>
<td>689</td>
</tr>
<tr>
<td>ttytab/1</td>
<td>690</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>wrchlin/4</td>
<td>801</td>
</tr>
<tr>
<td>wrchlod/3</td>
<td>802</td>
</tr>
<tr>
<td>wrchpxy/4</td>
<td>803</td>
</tr>
<tr>
<td>wrchsav/3</td>
<td>805</td>
</tr>
<tr>
<td>wrchsel/3</td>
<td>806</td>
</tr>
<tr>
<td>wrchset/3</td>
<td>807</td>
</tr>
<tr>
<td>wrchtxt/3</td>
<td>808</td>
</tr>
<tr>
<td>write/1</td>
<td>810</td>
</tr>
<tr>
<td>write_canonical/1</td>
<td>811</td>
</tr>
<tr>
<td>writeq/1</td>
<td>812</td>
</tr>
<tr>
<td>wscpipos/2</td>
<td>813</td>
</tr>
<tr>
<td>wscirlng/3</td>
<td>815</td>
</tr>
<tr>
<td>wshow/2</td>
<td>817</td>
</tr>
<tr>
<td>wsize/5</td>
<td>819</td>
</tr>
<tr>
<td>wstppos/2</td>
<td>820</td>
</tr>
<tr>
<td>wstprng/3</td>
<td>822</td>
</tr>
<tr>
<td>wstyle/2</td>
<td>824</td>
</tr>
<tr>
<td>wtabbadd/4</td>
<td>826</td>
</tr>
<tr>
<td>wtabdel/2</td>
<td>827</td>
</tr>
<tr>
<td>wtabget/4</td>
<td>828</td>
</tr>
<tr>
<td>wtabra/5</td>
<td>829</td>
</tr>
<tr>
<td>wtabsel/2</td>
<td>830</td>
</tr>
<tr>
<td>wtcreate/6</td>
<td>831</td>
</tr>
<tr>
<td>wtext/2</td>
<td>832</td>
</tr>
<tr>
<td>wthumb/3</td>
<td>833</td>
</tr>
<tr>
<td>wtrkpos/2</td>
<td>835</td>
</tr>
<tr>
<td>wtrkrng/3</td>
<td>837</td>
</tr>
<tr>
<td>wtwwadd/5</td>
<td>839</td>
</tr>
<tr>
<td>wtwwbtn/3</td>
<td>840</td>
</tr>
<tr>
<td>wtwwdel/2</td>
<td>841</td>
</tr>
<tr>
<td>wtwwget/4</td>
<td>842</td>
</tr>
<tr>
<td>wtwwlnk/4</td>
<td>843</td>
</tr>
<tr>
<td>wtwwsel/2</td>
<td>845</td>
</tr>
<tr>
<td>wucreate/2</td>
<td>846</td>
</tr>
<tr>
<td>xinit/9</td>
<td>847</td>
</tr>
<tr>
<td>xload/1</td>
<td>849</td>
</tr>
<tr>
<td>Message</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>msg_horz</td>
<td>882</td>
</tr>
<tr>
<td>msg_key</td>
<td>883</td>
</tr>
<tr>
<td>msg_leftdouble</td>
<td>884</td>
</tr>
<tr>
<td>msg_leftdown</td>
<td>885</td>
</tr>
<tr>
<td>msg_leftup</td>
<td>886</td>
</tr>
<tr>
<td>msg_menu</td>
<td>887</td>
</tr>
<tr>
<td>msg_mouseenter</td>
<td>888</td>
</tr>
<tr>
<td>msg_mousehover</td>
<td>889</td>
</tr>
<tr>
<td>msg_mouseleave</td>
<td>890</td>
</tr>
<tr>
<td>msg_mousemove</td>
<td>891</td>
</tr>
<tr>
<td>msg_move</td>
<td>892</td>
</tr>
<tr>
<td>msg_paint</td>
<td>893</td>
</tr>
<tr>
<td>msg_popup</td>
<td>894</td>
</tr>
<tr>
<td>msg_rightdouble</td>
<td>895</td>
</tr>
<tr>
<td>msg_rightdown</td>
<td>896</td>
</tr>
<tr>
<td>msg_rightup</td>
<td>897</td>
</tr>
<tr>
<td>msg_select</td>
<td>898</td>
</tr>
<tr>
<td>msg_size</td>
<td>899</td>
</tr>
<tr>
<td>msg_sysmenu</td>
<td>900</td>
</tr>
<tr>
<td>msg_touch</td>
<td>901</td>
</tr>
<tr>
<td>msg_tvcancel</td>
<td>902</td>
</tr>
<tr>
<td>msg_tvdrag</td>
<td>903</td>
</tr>
<tr>
<td>msg_tvedit</td>
<td>904</td>
</tr>
<tr>
<td>msg_tventer</td>
<td>905</td>
</tr>
<tr>
<td>msg_tvhide</td>
<td>906</td>
</tr>
<tr>
<td>msg_tvleave</td>
<td>907</td>
</tr>
<tr>
<td>msg_tvrename</td>
<td>908</td>
</tr>
<tr>
<td>msg_tvshow</td>
<td>909</td>
</tr>
<tr>
<td>msg_vert</td>
<td>910</td>
</tr>
<tr>
<td>msg_wheeldouble</td>
<td>911</td>
</tr>
<tr>
<td>msg_wheeldown</td>
<td>912</td>
</tr>
<tr>
<td>msg_wheelup</td>
<td>913</td>
</tr>
</tbody>
</table>

Appendix E - The GraFiX Language

GraFiX: Level 1 Code
GraFiX: Level 2 Code
GraFiX: Level 3 Code
The GraFiX Predicates: Object Maintenance ................................................................. 916
GraFiX Predicates: Device Context Management .......................................................... 916
GraFiX Predicates: Device Context Settings ................................................................. 917
GraFiX Predicates: Hit Tests ....................................................................................... 918
GraFiX Predicates: Grafix Window Control ................................................................. 918
GraFiX Predicates: GraFiX! ......................................................................................... 918
The GraFiX Language: Structure ............................................................................ 919
The GraFiX Language: Selectable Objects .................................................................. 921
The GraFiX Language: Selectable Transformations .................................................... 922
The GraFiX Language: Graphics Primitives ................................................................. 923
arc(x1,y1,x2,y2,xs,ys,xf,yf) ......................................................................................... 923
bitmap(x1,y1,x2,y2,xo,yo,bitmap) ............................................................................. 923
chord(x1,y1,x2,y2,xs,ys,xf,yf) .................................................................................... 923
eclipse(x1,y1,x2,y2) .................................................................................................... 924
icon(x1,y1,icon) ............................................................................................................ 924
metafile(x1,y1,x2,y2,meta) .......................................................................................... 924
pie(x1,y1,x2,y2,xs,ys,xf,yf) ........................................................................................ 924
polybez(x1,y1,x11,y11,x12,y12,x2,y2,..xn,yn) ............................................................. 924
polygon(x1,y1,x2,y2,..xn,yn) ..................................................................................... 924
polyline(x1,y1,x2,y2,..xn,yn) ................................................................................... 924
rectangle(x1,y1,x2,y2) .............................................................................................. 925
roundrect(x1,y1,x2,y2,xd,yd) ..................................................................................... 925
text(x1,y1,string) ....................................................................................................... 925
The GraFiX Technical Reference: Support Predicates .................................................. 925
gfx/1 ............................................................................................................................. 925
gfx_back_close/1 ........................................................................................................ 925
gfx_back_create/4 ....................................................................................................... 925
gfx_back_dict/2 .......................................................................................................... 925
gfx_back_handle/2 ...................................................................................................... 926
gfx_begin/1 .................................................................................................................. 926
gfx_begin/3 .................................................................................................................. 926
gfx_bitmap_close/1 ...................................................................................................... 927
gfx_bitmap_dict/2 ........................................................................................................ 927
gfx_bitmap_handle/2 ................................................................................................. 927
gfx_bitmap_load/2 ....................................................................................................... 927
gfx_brush_close/1 ................................................................. 927
gfx_brush_create/5 ................................................................. 927
gfx_brush_dict/2 ................................................................. 928
gfx_brush_handle/2 ................................................................. 928
gfx_cleanup/0 ................................................................. 928
gfx_clipping/4 ................................................................. 928
gfx_cursor_handle/2 ................................................................. 928
gfx_end/1 ................................................................. 929
gfx_end/3 ................................................................. 929
gfx_font_close/1 ................................................................. 930
gfx_font_create/4 ................................................................. 930
gfx_font_dict/2 ................................................................. 930
gfx_font_handle/2 ................................................................. 930
gfx_fore_close/1 ................................................................. 931
gfx_fore_create/4 ................................................................. 931
gfx_fore_dict/2 ................................................................. 931
gfx_fore_handle/2 ................................................................. 931
gfx_icon_close/1 ................................................................. 931
gfx_icon_dict/2 ................................................................. 931
gfx_icon_handle/2 ................................................................. 932
gfx_icon_load/3 ................................................................. 932
gfx_mapping/4 ................................................................. 932
gfx_metafile_close/1 ................................................................. 932
gfx_metafile_dict/2 ................................................................. 932
gfx_metafile_handle/2 ................................................................. 932
gfx_metafile_load/2 ................................................................. 933
gfx_origin/2 ................................................................. 933
gfx_paint/1 ................................................................. 933
gfx_pen_close/1 ................................................................. 933
gfx_pen_create/5 ................................................................. 933
gfx_pen_dict/2 ................................................................. 934
gfx_pen_handle/2 ................................................................. 934
gfx_resolution/4 ................................................................. 934
gfx_rop_handle/2 ................................................................. 934
gfx_select/1 ................................................................. 935
gfx_test/1 ................................................................. 936
Appendix F - The Development Environment ................................................................. 937
  The File Menu: system_menu( W, file, F ) ................................................................. 937
  The Edit Menu: system_menu( W, edit, F ) ................................................................. 938
  The Search Menu: system_menu( W, search, F ) ......................................................... 939
  The Run Menu: system_menu( W, query, F ) ............................................................... 939
  The Options Menu: system_menu( W, options, F ) ...................................................... 940
  The Window Menu ....................................................................................................... 941
  The Help Menu: system_menu( W, help, F ) ................................................................ 941
Appendix G - Errors, Names and Numbers ..................................................................... 942
  Success and Failure ...................................................................................................... 942
  Keyboard Break Error ................................................................................................. 942
  Memory Errors ............................................................................................................ 942
  Console I/O Errors ..................................................................................................... 944
  Predicate Errors ........................................................................................................ 946
  File Handling Errors ................................................................................................. 947
  Term I/O Errors .......................................................................................................... 948
  Arithmetic Handling Errors ...................................................................................... 950
  Assembly and Compilation Errors ............................................................................. 950
  DOS Errors and Unknown Errors .............................................................................. 951
Appendix H - Prolog Memory Management ................................................................. 952
  Memory: Stacks, Heaps and Buffers ........................................................................... 952
  Memory Stacks ........................................................................................................... 952
  Memory Heaps ............................................................................................................ 954
  Memory Buffers .......................................................................................................... 955
  Controlling Stack Usage ............................................................................................ 955
  The Backtrack Stack .................................................................................................. 955
  The Local Stack .......................................................................................................... 959
  The Reset Stack ......................................................................................................... 960
  The System Stack ....................................................................................................... 961
  Setting Stack Sizes .................................................................................................... 961
  Setting Heap Sizes .................................................................................................... 965
  Command Line Switches ............................................................................................ 969
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix Q - MIDI and Music</td>
<td>1024</td>
</tr>
<tr>
<td>Introduction</td>
<td>1024</td>
</tr>
<tr>
<td>The Birth of MIDI</td>
<td>1025</td>
</tr>
<tr>
<td>MIDI and Computers</td>
<td>1025</td>
</tr>
<tr>
<td>Serial, USB, FireWire and Soundcards</td>
<td>1025</td>
</tr>
<tr>
<td>MIDI and Prolog</td>
<td>1026</td>
</tr>
<tr>
<td>MIDI in WIN-PROLOG</td>
<td>1026</td>
</tr>
<tr>
<td>Long and Short Messages</td>
<td>1027</td>
</tr>
<tr>
<td>Buffering MIDI Events</td>
<td>1028</td>
</tr>
<tr>
<td>Creating a MIDI Device</td>
<td>1028</td>
</tr>
<tr>
<td>Sending and Receiving MIDI Data</td>
<td>1029</td>
</tr>
<tr>
<td>A Simple Polling Sequencer</td>
<td>1030</td>
</tr>
<tr>
<td>MIDI Events and Handlers</td>
<td>1032</td>
</tr>
<tr>
<td>More on Timer Events</td>
<td>1033</td>
</tr>
<tr>
<td>More on Handlers</td>
<td>1034</td>
</tr>
<tr>
<td>Ancillary Predicates</td>
<td>1035</td>
</tr>
<tr>
<td>Appendix R - MIDI Message Definitions</td>
<td>1037</td>
</tr>
<tr>
<td>Appendix S - Standard Syntax</td>
<td>1043</td>
</tr>
<tr>
<td>The &quot;New&quot; Standard Syntax</td>
<td>1043</td>
</tr>
<tr>
<td>Canonical Operator-Free Syntax</td>
<td>1044</td>
</tr>
<tr>
<td>Edinburgh Operator Ambiguities</td>
<td>1044</td>
</tr>
<tr>
<td>Interoperability with Other Languages</td>
<td>1046</td>
</tr>
<tr>
<td>The Standard Syntax Predicates</td>
<td>1046</td>
</tr>
<tr>
<td>Pretty Printing</td>
<td>1047</td>
</tr>
<tr>
<td>Terminating Standard Syntax Terms</td>
<td>1048</td>
</tr>
<tr>
<td>The Standard Syntax Source Format</td>
<td>1048</td>
</tr>
<tr>
<td>Appendix T - TreeView Controls</td>
<td>1050</td>
</tr>
<tr>
<td>About TreeView Controls</td>
<td>1050</td>
</tr>
<tr>
<td>Keeping Track of Location</td>
<td>1050</td>
</tr>
<tr>
<td>Planting a Tree</td>
<td>1051</td>
</tr>
<tr>
<td>Branching Out</td>
<td>1051</td>
</tr>
<tr>
<td>Climbing the Tree</td>
<td>1053</td>
</tr>
<tr>
<td>Pruning and Decorating the Tree</td>
<td>1054</td>
</tr>
<tr>
<td>Messages in the Trees</td>
<td>1055</td>
</tr>
</tbody>
</table>
Introduction

Welcome to the latest release of WIN-PROLOG! This exciting version of Brian D Steel's acclaimed Prolog compiler is the first to be released with an all-new, fully 64-bit Prolog engine. Inspired by the design of his existing 32-bit engine, the new system inherits all the stability and extensive functionality of a time-proven design, but now with the unlimited memory resources of the 64-bit world. Both systems have been hand-built in assembler language for maximum performance, and carefully tested against one another to maintain maximum compatibility between the X86 (32-bit) and X64 (64-bit) platforms. As well as providing access to as much of your machine's memory as you want, WIN-PROLOG provides convenient access to a large number of Windows Graphical User Interface (GUI) functions, allowing you to create polished Windows applications.

What's in WIN-PROLOG?

WIN-PROLOG is a no-compromise 64-bit Prolog compiler and programming environment, and is highly compatible with the existing 32-bit compiler. Fully conforming to the Edinburgh standard, a complete set of Clocksin and Mellish predicates is augmented to provide a high degree of inter-operability with Quintus Prolog. A unique string data type and memory-buffered files permit powerful file, window and other input/output (I/O) processing not normally possible in Prolog, as well as direct control of the Windows GUI and DLLs, DDE and OLE automation applications.

WIN-PROLOG provides an MDI-compliant programming environment, featuring multiple program editor windows, with automatic rich syntax colouring, incremental and optimised compilation, source level debugging, and comprehensive text search and replace facilities. All of the GUI features used by the environment, and many more besides, are directly available to Prolog programs, allowing custom dialogs and windows to be built and shipped as part of an application.

Dynamic Link Libraries (DLLs) written in C/C++, Pascal or any other Windows development language can be loaded and called by WIN-PROLOG, and all types of data can be passed to Prolog programs and the DLLs' functions. Furthermore, DLLs can send messages to Prolog at any point, allowing background processing, modeless dialogs, and interprocess communication to be built in easily.

About this Manual

This manual describes the technical features of WIN-PROLOG, and should be used in conjunction with the Windows Programming Guide. Throughout this manual, illustrated examples show how the various features of WIN-PROLOG can be combined to provide a truly powerful data processing engine. Please enjoy this manual, and have fun with WIN-PROLOG!

Brian D Steel, 14 Mar 2019
The Technical Reference

The Technical Reference contains detailed information about the built-in predicates in the Windows version of LPA-PROLOG (hereafter referred to as WIN-PROLOG). For ease of access, the reference pages for the predicates are presented in alphabetical order. Users new to WIN-PROLOG are advised to read the Programming Guides, which are organised according to subject.

This manual is divided into two parts: the first part is an alphabetical reference to majority of predicates in WIN-PROLOG; the second part comprises a number of appendices which contain additional information about WIN-PROLOG.

File Names

The WIN-PROLOG system comprises several software layers, and although the majority of this manual is concerned with the topmost, Quintus Prolog-compatible layer, occasional references will be made to predicates from one or more of the inner layers. Normally the user will be unaffected by the distinction, but with one class of predicates in particular, namely those handling file names, the differentiation is important.

Most of the file handling predicates mentioned in this manual are from the top layer, designed for compatibility with Quintus Prolog. These predicates make use of "logical file names" that are automatically converted to "absolute file names" before being used. For example, to read input from a file you would normally use the see/1 predicate, such as in the command:

?- see( foo ).

When this command is entered, WIN-PROLOG will first convert the name "foo" to an absolute file name, perhaps:

'c:\program files\win-prolog 5000\foo.pl'

and then open the file for input. The conversion from logical to absolute file name is performed automatically by a call to the absolute_file_name/2 predicate, which is discussed in detail in the alphabetical reference section.

It is possible to open files for input and output using inner-layer predicates; for example, you could also open a file with fcreate/5 and set it up as the input stream with input/1 as follows:

?- fcreate( foo, 'foo.pl', 0, 0, 0 ), input( foo ).

This would cause all input to be read from "foo.pl", in much the same way as in the see/1 example above. The major difference, however, is that while in the former case, WIN-PROLOG "knows" the file as the atom:
'c:\program files\win-prolog 5000\foo.pl'

in the latter case it "knows" it simply as the atom:

    foo

In general, it is not easy to write code which mixes file predicates from the top, Quintus Prolog-compatible with those from lower layers, without making extensive and explicit use of absolute_file_name/2.

Font and Layout Conventions

Just three primary font families are used in this manual, with exceptions only being made for special tradenames or font-based examples. The main font families are used respectively for general text, computer text and tabular listings; throughout the remainder of this manual they are therefore referred to simply as the "text", "computer" and "table" fonts. These are described in more detail in the following sections.

The "Text" Font

The bulk of the text in this manual, including headings and narrative text, is presented in four variants of Bitstream Benguiat Gothic:

Benguiat Gothic Book
Benguiat Gothic Book Italic
Benguiat Gothic Bold
Benguiat Gothic Bold Italic

This "text" font has been chosen because it is easy to read, thanks to its clean lines, gentle curves and large "x height". The "Bold" variants are generally reserved for headings, while the "Book" (normal) variants are used for narrative (body) text. The use of italics is generally limited to computer terms or Prolog predicate names.

The "Computer" Font

Text which represents computer input and output, program listings, etc., is shown in two variants of Adobe Courier:

Courier Medium
Courier Bold

This "computer" font has been chosen because it closely resembles the default console font (actually Courier New), allowing command and program lines
to be shown with exactly the same layout as in WIN-PROLOG itself. In general, the medium variant is used to denote output, and the bold variant to denote input.

**Special Fonts**

The only other fonts used are reserved for special names and trademarks, and occasional places where a fixed width font is required; these include, but are not limited to, VAG Rounded and ITC Garamond Book Condensed:

- **VAG Rounded**
  - ITC Garamond Book Condensed

VAG Rounded is used to display the word "PROLOG", when referring to any of the systems, DOS-PROLOG, WIN-PROLOG, and BDS-PROLOG, as well as the three-letter prefix. Garamond is used only to refer to the MacProlog32 system.

Chapter and section headings have been kept simple, and with the exception of the Alphabetical Reference Section (see next), indentation is limited to code examples, tables and enumerated listings.

**Predicate Entries**

The main part of the Technical Reference consists of an alphabetical listing of the majority of the predicates in WIN-PROLOG. A few special system predicates are have been omitted: these provide low-level support for other predicates, and are of little interest in their own right. Each predicate in this section is described, starting on a fresh page, in a standard format, as outlined below:
functor/arity

simple description of predicate

functor(Arg1, Arg2 ... Argn)

?Arg1 <type1>
?Arg2 <type2>
...
?Argn <typen>

Comments
A detailed description of this predicate and its arguments.

Examples
Explanation of any examples that follow:

?- bold(shows, which, bits, you, type).
  all output is shown in medium

Notes
Any special notes pertaining to this predicate.

Predicate Heading
The heading for each predicate shows the functor (name) and arity (argument count) of the predicate. The reference section in this manual is sorted alphabetically with respect to the functor and arity predicate.

Predicate Summary
Following the predicate heading is a summary. This comprises a one-line description, shown in italic "text" font, followed by a typical call pattern, shown in bold "computer" font (see above). This is followed by a listing of the arguments, in normal computer font, together with information about their acceptable modes and types (see below).

Argument Modes
The "mode" of an argument defines whether the argument can be used for input (argument is already bound at time of call), output (argument is a variable at the time of call), or both (it does not matter). These modes are denoted by three symbols, as shown in the following table:
Symbol | Mode
---|---
+ | input argument
- | output argument
? | input/output argument

### Argument Types

As well as a mode, each argument is shown as being of one or more of a number of "types"; these types may be further qualified by "domains" or "ranges", for example showing the valid range of an integer argument. The argument types are shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;address&gt;</code></td>
<td>an integer specifying a memory address as an offset within the application’s data segment, or a conjunction of integers specifying a full little-endian (offset,segment) address</td>
</tr>
<tr>
<td><code>&lt;arity&gt;</code></td>
<td>an integer in the range [0..128] that denotes the number of arguments of a predicate</td>
</tr>
<tr>
<td><code>&lt;atom&gt;</code></td>
<td>a Prolog atom</td>
</tr>
<tr>
<td><code>&lt;char&gt;</code></td>
<td>a 32-bit integer representing a character code</td>
</tr>
<tr>
<td><code>&lt;char_list&gt;</code></td>
<td>a list of 32-bit integers each representing a character code</td>
</tr>
<tr>
<td><code>&lt;clause&gt;</code></td>
<td>an atom, compound term or construct of the form: head :- body, where head may be an atom or compound term and body is a goal, or a conjunction or disjunction of goals, the whole representing a clause in a Prolog program</td>
</tr>
<tr>
<td><code>&lt;compound_term&gt;</code></td>
<td>a construct of the form: functor(arguments), where functor is any term and arguments is a sequence of terms separated by commas</td>
</tr>
<tr>
<td><code>&lt;date_time&gt;</code></td>
<td>a conjunction of two integers, comprising a day and millisecond count respectively</td>
</tr>
<tr>
<td><code>&lt;empty_list&gt;</code></td>
<td>a list with no entries, written &quot;[]&quot;</td>
</tr>
<tr>
<td><code>&lt;expr&gt;</code></td>
<td>an arithmetic expression or a number</td>
</tr>
<tr>
<td><code>&lt;file_name&gt;</code></td>
<td>an atom representing a file</td>
</tr>
<tr>
<td><code>&lt;file_spec&gt;</code></td>
<td>an atom representing a file or a compound term of the form: path_alias(filename)</td>
</tr>
<tr>
<td><code>&lt;file_specs&gt;</code></td>
<td>a file_spec or a list or conjunction of file_specs</td>
</tr>
<tr>
<td><code>&lt;float&gt;</code></td>
<td>a 64-bit floating point number in the range [2.2e-308..1.7e308]</td>
</tr>
<tr>
<td><code>&lt;functor&gt;</code></td>
<td>an atom representing the name of a predicate</td>
</tr>
<tr>
<td><code>&lt;goal&gt;</code></td>
<td>a compound term or atom that represents a call to a Prolog program</td>
</tr>
</tbody>
</table>
### <integer>
(X86) a 32-bit integer in the range \([-2147483648..2147483647]\)
(X64) a 64-bit integer in the range \([-9223372036854775808..9223372036854775807]\)

### <integer_expr>
either an expression which evaluates to an integer or an integer

### <list>
the Prolog list data type

### <list_of Type>
a list with elements of the given type

### <number>
an integer or a float

### <object_handle>
an atom naming a user-defined GraFiX object, or a compound of the form atom(integer), where the atom is object type and the integer its raw handle, or a term of the form stock(atom), where atom is the name of a stock object

### <path_alias>
an atom representing a logical path alias

### <pred_spec>
a structure of the form: functor/arity where functor is the name of the predicate and arity is the number of arguments of the predicate

### <pred_specs>
a single predicate specification, or a list or conjunction of predicate specifications

### <stream>
an integer in the range \([0..2]\), or an atom naming an open file, or a conjunction of (string,integer)

### <string>
the LPA-PROLOG string data type

### <term>
any valid Prolog term: a variable, integer, floating point number, atom, string, list, char list or compound term

### <variable>
an uninstantiated Prolog variable

### <varlist>
a list of conjunctions, each containing an atom and a variable, used to name variables during input and output

### <window_handle>
an atom, conjunction of atom and integer or integer that refers to a window created by Prolog, or an integer that refers to an external window.

### Predicate Comments

This section contains more detailed comments about the predicate, including any relevant background information. The text is shown in normal "text" font, with any predicates or argument references shown in italic "text" font; any tables are shown in variants of the "table" font (see above).

### Predicate Example

Most predicate descriptions include one or more simple examples that help explain how it works. Any explanatory text is shown in normal "text" font, while code listings and commands are shown in "computer" font. With commands, anything typed by the user is shown in bold, and where appropriate, the names of any special keys are shown, surrounded by angle brackets, in italicised "text" font (see above).
Predicate Categories

Classification is always a controversial issue, and none more so than when attempting to categorise the several hundred predicates in WIN-PROLOG into meaningful groups. The following attempt is the result of a day's brainstorming, followed by several iterations of refinement, and it is hoped that this will help give some feel as to the "geography" of WIN-PROLOG.

Arithmetic Predicates

These predicates perform arithmetic computations and comparisons, with the exception of seed/1, which is used to initialise the pseudo random number generator.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=/2</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>=:=/2</td>
<td>Equal to</td>
</tr>
<tr>
<td>==/2</td>
<td>Not equal to</td>
</tr>
<tr>
<td>=&lt;/2</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;/2</td>
<td>Greater than</td>
</tr>
<tr>
<td>rpn/2</td>
<td>RPN notation</td>
</tr>
<tr>
<td>seed/1</td>
<td>Initialise pseudo random number generator</td>
</tr>
</tbody>
</table>

Arithmetic Functions

These are not predicates in their own right, but can be used in expressions in any of the Arithmetic Predicates (see above), and are explained in greater detail in the entry for is/2 in the main alphabetical section.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*(X,Y)</td>
<td>Product</td>
</tr>
<tr>
<td>+(X)</td>
<td>Sum</td>
</tr>
<tr>
<td>+(X,Y)</td>
<td>Plus</td>
</tr>
<tr>
<td>-(X)</td>
<td>Minus</td>
</tr>
<tr>
<td>-(X,Y)</td>
<td>Difference</td>
</tr>
<tr>
<td>/(X,Y)</td>
<td>Divide</td>
</tr>
<tr>
<td>fPn/2</td>
<td>Floating point number</td>
</tr>
<tr>
<td>is/2</td>
<td>Integer</td>
</tr>
<tr>
<td>a(l,J)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>abs(X)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>acos(X)</td>
<td>Arc cosine</td>
</tr>
<tr>
<td>aln(X)</td>
<td>Logarithm (natural base)</td>
</tr>
<tr>
<td>alog(X)</td>
<td>Logarithm (base 10)</td>
</tr>
<tr>
<td>aSin(X)</td>
<td>Arc sine</td>
</tr>
<tr>
<td>atan(X)</td>
<td>Arc tangent</td>
</tr>
<tr>
<td>cos(X)</td>
<td>Cosine</td>
</tr>
<tr>
<td>fp(X)</td>
<td>Floating point</td>
</tr>
<tr>
<td>int(X)</td>
<td>Integer</td>
</tr>
<tr>
<td>ip(X)</td>
<td>Integer part</td>
</tr>
<tr>
<td>ln(X)</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>log(X)</td>
<td>Logarithm</td>
</tr>
<tr>
<td>max(X,Y)</td>
<td>Maximum</td>
</tr>
<tr>
<td>min(X,Y)</td>
<td>Minimum</td>
</tr>
<tr>
<td>mod(X,Y)</td>
<td>Modulus</td>
</tr>
<tr>
<td>o(l,J)</td>
<td>Odd part</td>
</tr>
<tr>
<td>p(l,J)</td>
<td>Positive part</td>
</tr>
<tr>
<td>rand(X)</td>
<td>Random number</td>
</tr>
<tr>
<td>sign(X)</td>
<td>Sign</td>
</tr>
<tr>
<td>sin(X)</td>
<td>Sine</td>
</tr>
<tr>
<td>sq(X)</td>
<td>Square</td>
</tr>
<tr>
<td>sqrt(X)</td>
<td>Square root</td>
</tr>
<tr>
<td>tan(X)</td>
<td>Tangent</td>
</tr>
<tr>
<td>x(l,J)</td>
<td>Exponent part</td>
</tr>
</tbody>
</table>
Control Predicates

These predicates provide control of program execution, including the handling of backtracking and abrupt exits.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>!/0</td>
<td>’;/2</td>
<td>-&gt;/2</td>
<td>;/2</td>
</tr>
<tr>
<td>+/1</td>
<td>abort/0</td>
<td>exit/1</td>
<td>fail/0</td>
</tr>
<tr>
<td>false/0</td>
<td>halt/0</td>
<td>halt/1</td>
<td>not/1</td>
</tr>
<tr>
<td>one/1</td>
<td>otherwise/0</td>
<td>repeat/0</td>
<td>repeat/1</td>
</tr>
<tr>
<td>repeat/2</td>
<td>repeat/3</td>
<td>true/0</td>
<td></td>
</tr>
</tbody>
</table>

Meta Programming Predicates

These predicates take Prolog goals as one or more of their arguments, and execute them as metacalls.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>’;/2</td>
<td>-&gt;/2</td>
<td>;/2</td>
<td>+/1</td>
</tr>
<tr>
<td>^/2</td>
<td>&lt;=/2</td>
<td>=&gt;/2</td>
<td></td>
</tr>
<tr>
<td>call/1</td>
<td>call/2</td>
<td>catch/2</td>
<td>catch/3</td>
</tr>
<tr>
<td>dynamic_call/1</td>
<td>findall/3</td>
<td>forall/2</td>
<td>force/1</td>
</tr>
<tr>
<td>ms/2</td>
<td>setof/3</td>
<td>solution/2</td>
<td></td>
</tr>
</tbody>
</table>

Solution Predicates

This small group of predicates can be used to collect or obtain multiple solutions to a Prolog goal or query.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>asn/2</td>
<td>bagof/3</td>
<td>findall/3</td>
<td>setof/3</td>
</tr>
<tr>
<td>solution/2</td>
<td>toteall/3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List Processing Predicates

These predicates provide support for common list processing requirements.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>append/3</td>
<td>keysort/2</td>
<td>len/2</td>
<td>length/2</td>
</tr>
</tbody>
</table>
Term Manipulation Predicates

These predicates provide conversions between terms of different data types, as well as access to subterms within compound terms.

<table>
<thead>
<tr>
<th>member/2</th>
<th>member/3</th>
<th>member/4</th>
<th>remove/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeall/3</td>
<td>reverse/2</td>
<td>reverse/3</td>
<td>sort/2</td>
</tr>
<tr>
<td>sort/3</td>
<td>stir/2</td>
<td>sublist/2</td>
<td></td>
</tr>
</tbody>
</table>

| =../2            | arg/3           | atom_chars/2    | atom_string/2  |
| cat/3                | copy_term/2     | dup/2           | functor/3      |
| hide/2              | lwrupr/2        | mem/3           | name/2         |
| numbervars/3        | number_atom/2   | number_chars/2  | number_string/2|
| phrase/2            | phrase/3        | string_chars/2  | strutf/3       |
| vars/2              |                 |                 |                |

Term Comparison Predicates

These predicates perform comparisons between pairs of terms according to the standard ordering of terms.

| ==/2            | \=/2           | @</2           | @=/2           |
| @<=/2              | @>/2           | @>/=/2         | @\=/2         |
| cmp/3              | compare/3      | eqv/2          |                 |

Term Unification Predicates

These predicates perform unification, or tests on unification, between terms.

| /=2            | \=/2           | occurs_chk/2   | subsumes_chk/2 |
| unifiable/2   |                |                |                |

Term Type Predicates

These predicates provide a comprehensive set of type tests for terms:
<table>
<thead>
<tr>
<th>arity/1</th>
<th>atom/1</th>
<th>atomic/1</th>
<th>callable/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>char/1</td>
<td>chars/1</td>
<td>compound/1</td>
<td>constant/1</td>
</tr>
<tr>
<td>float/1</td>
<td>ground/1</td>
<td>integer/1</td>
<td>list/1</td>
</tr>
<tr>
<td>lst/1</td>
<td>nonvar/1</td>
<td>number/1</td>
<td>simple/1</td>
</tr>
<tr>
<td>string/1</td>
<td>type/2</td>
<td>var/1</td>
<td></td>
</tr>
</tbody>
</table>

**System Status Predicates**

This set of predicates provides access to system status parameters, both to test the existing state and to set new behaviour states.

<table>
<thead>
<tr>
<th>current_atom/1</th>
<th>current_predicate/1</th>
<th>current_predicate/2</th>
<th>dict/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>elex/1</td>
<td>fileerrors/0</td>
<td>free/9</td>
<td>nofileerrors/0</td>
</tr>
<tr>
<td>no_style_check/1</td>
<td>prolog_flag/2</td>
<td>prolog_flag/3</td>
<td>prolog_load_context/2</td>
</tr>
<tr>
<td>statistics/0</td>
<td>statistics/2</td>
<td>stats/4</td>
<td>total/9</td>
</tr>
<tr>
<td>style_check/1</td>
<td>ver/1</td>
<td>ver/4</td>
<td>version/0</td>
</tr>
<tr>
<td>version/1</td>
<td>xinit/9</td>
<td>xload/1</td>
<td>xsave/1</td>
</tr>
</tbody>
</table>

**Operating System Predicates**

These predicates provide low level access to the operating system, allowing the management of files and directories, together with the ability to execute and control external processes.

<table>
<thead>
<tr>
<th>attrib/2</th>
<th>chdir/1</th>
<th>del/1</th>
<th>dir/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir/4</td>
<td>dos/0</td>
<td>dos/1</td>
<td>exec/3</td>
</tr>
<tr>
<td>exit/1</td>
<td>env/1</td>
<td>file/3</td>
<td>fname/4</td>
</tr>
<tr>
<td>halt/0</td>
<td>halt/1</td>
<td>help/1</td>
<td>help/2</td>
</tr>
<tr>
<td>mkdir/1</td>
<td>profile/1</td>
<td>profile/4</td>
<td>ren/2</td>
</tr>
<tr>
<td>rmdir/1</td>
<td>stamp/2</td>
<td>switch/2</td>
<td>winapi/4</td>
</tr>
</tbody>
</table>
Date and Time Predicates

This small group of predicates provides access to local and elapsed time, and provides conversions between days, dates, milliseconds and "h:m:s" times.

<table>
<thead>
<tr>
<th>ms/2</th>
<th>pause/1</th>
<th>time/2</th>
<th>time/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>time/4</td>
<td>time/5</td>
<td>time/7</td>
<td></td>
</tr>
</tbody>
</table>

Stream Handling Predicates

These predicates are concerned with the maintenance of input/output streams: the input and output predicates themselves are listed in the next classification.

<table>
<thead>
<tr>
<th>←~/2</th>
<th>~/2</th>
<th>absolute_file_name/2</th>
<th>absolute_file_name/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>at_end_of_file/0</td>
<td>at_end_of_line/0</td>
<td>close/1</td>
<td>empty/0</td>
</tr>
<tr>
<td>eof/0</td>
<td>fclose/1</td>
<td>fcreate/5</td>
<td>fdata/5</td>
</tr>
<tr>
<td>fdict/2</td>
<td>file/1</td>
<td>file_search_path/2</td>
<td>flush/0</td>
</tr>
<tr>
<td>inpos/1</td>
<td>input/1</td>
<td>library_directory/1</td>
<td>open/2</td>
</tr>
<tr>
<td>open/3</td>
<td>outpos/1</td>
<td>output/1</td>
<td>see/1</td>
</tr>
<tr>
<td>seeing/1</td>
<td>seen/0</td>
<td>stream_position/2</td>
<td>stream_position/3</td>
</tr>
<tr>
<td>tell/1</td>
<td>telling/1</td>
<td>told/0</td>
<td></td>
</tr>
</tbody>
</table>

Stream Input/Output Predicates

These predicates perform input and output of terms and other lower-level data from and to streams created by the predicates in the previous classification.

<table>
<thead>
<tr>
<th>display/1</th>
<th>eread/1</th>
<th>eread/2</th>
<th>eprint/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>eprint/2</td>
<td>eprint/3</td>
<td>etoks/1</td>
<td>etoks/2</td>
</tr>
<tr>
<td>ewrite/1</td>
<td>ewrite/2</td>
<td>fread/4</td>
<td>fwrite/4</td>
</tr>
<tr>
<td>get/1</td>
<td>get0/1</td>
<td>getx/2</td>
<td>nl/0</td>
</tr>
<tr>
<td>portray/1</td>
<td>portray_clause/1</td>
<td>portray_clause/2</td>
<td></td>
</tr>
</tbody>
</table>
Stream Data Processing Predicates

This small, special purpose group of predicates performs a variety of direct stream input and output functions, bypassing intermediate Prolog terms, and provides data copying, encryption, compression and searching features.

<table>
<thead>
<tr>
<th>copy/2</th>
<th>crc/3</th>
<th>decode/2</th>
<th>encode/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>find/3</td>
<td>fluff/3</td>
<td>mdf/3</td>
<td>sha/3</td>
</tr>
<tr>
<td>stuff/4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Console Input/Output Predicates

Related to the stream input/output predicates described earlier, this set of predicates always perform their input and output directly from and to the console window (user device).

<table>
<thead>
<tr>
<th>beep/2</th>
<th>getb/1</th>
<th>grab/1</th>
<th>keys/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>prompt/2</td>
<td>prompts/2</td>
<td>putb/1</td>
<td>ttyflush/0</td>
</tr>
<tr>
<td>ttyget/1</td>
<td>ttyget0/1</td>
<td>ttynl/0</td>
<td>ttyput/1</td>
</tr>
<tr>
<td>ttyskip/1</td>
<td>ttytab/1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operator Handling Predicates

This trio of predicates allows changes to be made to the syntax of Prolog terms, modifying the behaviour of many of the term-oriented stream input/output predicates, including read/1, write/1 and writeq/1 among others.

<table>
<thead>
<tr>
<th>current_op/3</th>
<th>op/3</th>
<th>sysops/0</th>
</tr>
</thead>
</table>
Grammar Rule Predicates

This quartet of predicates allows further changes to be made to the syntax of Prolog terms on input, modifying the meaning of program code during compilation, although it does not directly affect the stream input predicates.

| 'C'/3 | expand_dcg/2 | expand_term/2 | term_expansion/2 |

Compiler Predicates

These predicates are used to create, destroy, load and save Prolog programs at run-time, using any of the incremental, hashing and optimising compilers.

| abolish/1 | abolish/2 | abolish_files/1 | assert/1 |
| assert/2 | asserta/1 | assertz/1 | clause/2 |
| clause/3 | clauses/2 | compile/1 | consult/1 |
| decompile/3 | def/3 | defs/2 | dynamic/1 |
| ensure_loaded/1 | hash/1 | hash/3 | index/2 |
| initialization/1 | listing/0 | listing/1 | load_files/1 |
| load_files/2 | multifile/1 | nohash/1 | nohash/3 |
| optimize/1 | optimize_files/1 | pdict/4 | predicate_property/2 |
| reconsult/1 | retract/1 | retract/2 | retractall/1 |
| save_predicates/2 | source_file/1 | source_file/2 | source_file/3 |
| volatile/1 |

System and User Hooks

Among this collection are both a number of built-in predicates, as well as a series of user-definable hooks which can be used to intercept various run-time events.

| '?'ABORT?'/0 | '?'BREAK?'/1 | '?DEBUG?'/1 | '?ERROR?'/2 |
| ?'MESSAGE?'/4 | abort_hook/0 | break_hook/1 | debug_hook/1 |
| error_hook/2 | message_hook/4 | portray/1 | term_expansion/2 |
### Debugging Predicates

These predicates provide support for the debugging of Prolog programs.

<table>
<thead>
<tr>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>break/0</code></td>
</tr>
<tr>
<td><code>debug/0</code></td>
</tr>
<tr>
<td><code>debug_hook/1</code></td>
</tr>
<tr>
<td><code>debugging/0</code></td>
</tr>
<tr>
<td><code>force/1</code></td>
</tr>
<tr>
<td><code>leash/2</code></td>
</tr>
<tr>
<td><code>leashed/2</code></td>
</tr>
<tr>
<td><code>nodebug/0</code></td>
</tr>
<tr>
<td><code>nospy/1</code></td>
</tr>
<tr>
<td><code>nospyall/0</code></td>
</tr>
<tr>
<td><code>notrace/0</code></td>
</tr>
<tr>
<td><code>spy/1</code></td>
</tr>
<tr>
<td><code>trace/0</code></td>
</tr>
</tbody>
</table>

### Error, Timer and Message Predicates

This collection of predicates combines those that support error handling with others relating to timers, window messages and other interrupt events.

<table>
<thead>
<tr>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abort_hook/0</code></td>
</tr>
<tr>
<td><code>break/1</code></td>
</tr>
<tr>
<td><code>break_hook/1</code></td>
</tr>
<tr>
<td><code>busy/1</code></td>
</tr>
<tr>
<td><code>catch/2</code></td>
</tr>
<tr>
<td><code>catch/3</code></td>
</tr>
<tr>
<td><code>errmsg/2</code></td>
</tr>
<tr>
<td><code>error_hook/2</code></td>
</tr>
<tr>
<td><code>error_message/2</code></td>
</tr>
<tr>
<td><code>flag/1</code></td>
</tr>
<tr>
<td><code>message_hook/4</code></td>
</tr>
<tr>
<td><code>safe/1</code></td>
</tr>
<tr>
<td><code>sndmsg/5</code></td>
</tr>
<tr>
<td><code>throw/2</code></td>
</tr>
<tr>
<td><code>timer_close/1</code></td>
</tr>
<tr>
<td><code>timer_create/2</code></td>
</tr>
<tr>
<td><code>timer_data/2</code></td>
</tr>
<tr>
<td><code>timer_dict/2</code></td>
</tr>
<tr>
<td><code>timer_get/2</code></td>
</tr>
<tr>
<td><code>timer_set/2</code></td>
</tr>
<tr>
<td><code>unknown/2</code></td>
</tr>
<tr>
<td><code>unknown_predicate_handler/2</code></td>
</tr>
<tr>
<td><code>touch/0</code></td>
</tr>
<tr>
<td><code>touch/3</code></td>
</tr>
</tbody>
</table>

### Garbage Collection Predicates

Although it is always enabled, and called automatically whenever required, the garbage collector can be invoked directly by some of this group of predicates.

<table>
<thead>
<tr>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>garbage_collect/0</code></td>
</tr>
<tr>
<td><code>garbage_collect/1</code></td>
</tr>
<tr>
<td><code>gc/0</code></td>
</tr>
<tr>
<td><code>gc/1</code></td>
</tr>
<tr>
<td><code>nogc/0</code></td>
</tr>
</tbody>
</table>
### Graphics Predicates

This easily-recognised group of predicates are used to perform graphics, both to windows and to the printer (see below for information about printer control).

<table>
<thead>
<tr>
<th>gfx/1</th>
<th>gfx_back_close/1</th>
<th>gfx_back_create/4</th>
<th>gfx_back_dict/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfx_back_handle/2</td>
<td>gfx_begin/1</td>
<td>gfx_begin/3</td>
<td>gfx_bitmap_close/1</td>
</tr>
<tr>
<td>gfx_bitmap_dict/2</td>
<td>gfx_bitmap_handle/2</td>
<td>gfx_bitmap_load/2</td>
<td>gfx_brush_close/1</td>
</tr>
<tr>
<td>gfx_brush_create/5</td>
<td>gfx_brush_dict/2</td>
<td>gfx_brush_handle/2</td>
<td>gfx_cleanup/0</td>
</tr>
<tr>
<td>gfx_clipping/4</td>
<td>gfx_cursor_handle/2</td>
<td>gfx_end/1</td>
<td>gfx_end/3</td>
</tr>
<tr>
<td>gfx_font_close/1</td>
<td>gfx_font_create/4</td>
<td>gfx_font_dict/2</td>
<td>gfx_font_handle/2</td>
</tr>
<tr>
<td>gfx_fore_close/1</td>
<td>gfx_fore_create/4</td>
<td>gfx_fore_dict/2</td>
<td>gfx_fore_handle/2</td>
</tr>
<tr>
<td>gfx_icon_close/1</td>
<td>gfx_icon_dict/2</td>
<td>gfx_icon_handle/2</td>
<td>gfx_icon_load/3</td>
</tr>
<tr>
<td>gfx_mapping/4</td>
<td>gfx_metafile_close/1</td>
<td>gfx_metafile_dict/2</td>
<td>gfx_metafile_handle/2</td>
</tr>
<tr>
<td>gfx_metafile_load/2</td>
<td>gfx_origin/2</td>
<td>gfx_paint/1</td>
<td>gfx_pen_close/1</td>
</tr>
<tr>
<td>gfx_pen_create/5</td>
<td>gfx_pen_dict/2</td>
<td>gfx_pen_handle/2</td>
<td>gfx_resolution/4</td>
</tr>
<tr>
<td>gfx_rop_handle/2</td>
<td>gfx_select/1</td>
<td>gfx_test/1</td>
<td>gfx_transform/4</td>
</tr>
<tr>
<td>gfx_window_cursor/2</td>
<td>gfx_window_redraw/5</td>
<td>gfx_window_scroll/3</td>
<td></td>
</tr>
</tbody>
</table>

### Printer Control Predicates

This small set of predicates provides job control for the printer.

|prnend/1 | prnini/4 | prnpag/1 | prnstt/1|

### Font Handling Predicates

These predicates provide for the creation and utilisation of fonts with respect to windows in general: note that a similar set of font predicates is also present among the graphics predicates (see above).

<table>
<thead>
<tr>
<th>fnthdl/2</th>
<th>fonts/1</th>
<th>wfclose/1</th>
<th>wfcreate/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfdatal/5</td>
<td>wfdict/2</td>
<td>wfont/2</td>
<td>wfsze/4</td>
</tr>
</tbody>
</table>
Menu Handling Predicates

These predicates provide for the creation and utilisation of menus.

<table>
<thead>
<tr>
<th>predicate</th>
<th>predicate</th>
<th>predicate</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>popup_menu/4</td>
<td>mnuhdl/2</td>
<td>wmclose/1</td>
<td>wmcreate/1</td>
</tr>
<tr>
<td>wmdict/2</td>
<td>wmnuadd/4</td>
<td>wmnuget/4</td>
<td></td>
</tr>
<tr>
<td>wmnuunbl/3</td>
<td>wmnusel/3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Windows API and DLL Interface Predicates

An important trio of predicates, these provide direct calls to the Windows API as well as to most other DLLs.

<table>
<thead>
<tr>
<th>predicate</th>
<th>predicate</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>sndmsg/5</td>
<td>winapi/4</td>
<td>wintxt/4</td>
</tr>
</tbody>
</table>

Window Handling: Generic Predicates

This collection of predicates provides generic handling of windows.

<table>
<thead>
<tr>
<th>predicate</th>
<th>predicate</th>
<th>predicate</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>warea/5</td>
<td>wccreate/8</td>
<td>wclass/2</td>
<td>wclosel/1</td>
</tr>
<tr>
<td>wcount/4</td>
<td>wcreate/8</td>
<td>wdcreate/7</td>
<td>wdict/2</td>
</tr>
<tr>
<td>wenable/2</td>
<td>wfind/3</td>
<td>wfocus/1</td>
<td>window_handler/2</td>
</tr>
<tr>
<td>window_handler/4</td>
<td>wlink/3</td>
<td>wndhdl/2</td>
<td>wshow/2</td>
</tr>
<tr>
<td>wsize/5</td>
<td>wstyle/2</td>
<td>wtcreate/6</td>
<td>wtext/2</td>
</tr>
<tr>
<td>wucreate/6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window Handling: Button Controls

This single-predicate group provides support for button controls.

<table>
<thead>
<tr>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>wbtnsel/2</td>
</tr>
</tbody>
</table>
**Window Handling: Combobox Controls**

This group of predicates provides support for combobox controls.

<table>
<thead>
<tr>
<th>wcmbadd/4</th>
<th>wcmbdel/2</th>
<th>wcmbfnd/4</th>
<th>wcmbget/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>wcmbsel/2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Window Handling: Edit Controls**

This group of predicates provides support for edit controls.

<table>
<thead>
<tr>
<th>wedtclp/2</th>
<th>wedtfnd/6</th>
<th>wedtlin/4</th>
<th>wedtpxy/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>wedtsel/3</td>
<td>wedttxt/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Window Handling: Listbox Controls**

This group of predicates provides support for listbox controls.

<table>
<thead>
<tr>
<th>wlstadd/4</th>
<th>wlstdel/2</th>
<th>wlstfnd/4</th>
<th>wlstget/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>wlstsel/2</td>
<td>wlstsel/3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Window Handling: NudgeBar Controls**

This group of predicates provides support for NudgeBar controls.

<table>
<thead>
<tr>
<th>wndgbdy/2</th>
<th>wndgrng/3</th>
<th>wndgpos/2</th>
</tr>
</thead>
</table>

**Window Handling: Rich Controls**

This group of predicates provides support for rich controls.

<table>
<thead>
<tr>
<th>rich_colour/5</th>
<th>rich_format/3</th>
<th>wrchclp/2</th>
<th>wrchfnd/6</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrchget/3</td>
<td>wrchlen/3</td>
<td>wrchlin/4</td>
<td>wrchlod/3</td>
</tr>
</tbody>
</table>
Window Handling: Scrollbar Controls

This group of predicates provides support for scrollbar controls.

<table>
<thead>
<tr>
<th>wrchpxy/4</th>
<th>wrchsav/3</th>
<th>wrchsel/3</th>
<th>wrchset/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrchtxt/3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window Handling: StripBar Controls

This group of predicates provides support for StripBar controls.

<table>
<thead>
<tr>
<th>wrange/5</th>
<th>wthumb/3</th>
<th>wsclrng/3</th>
<th>wsclpos/2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window Handling: TabBox Controls

This group of predicates provides support for TabBox controls.

<table>
<thead>
<tr>
<th>wtabadd/4</th>
<th>wtabdel/2</th>
<th>wtabget/4</th>
<th>wtabrea/5</th>
</tr>
</thead>
<tbody>
<tr>
<td>wtabsel/2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window Handling: TrackBar Controls

This group of predicates provides support for TrackBar controls.

<table>
<thead>
<tr>
<th>wtrkrng/3</th>
<th>wtrkpos/2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window Handling: TreeView Controls

This group of predicates provides support for TreeView controls.

<table>
<thead>
<tr>
<th>wtvwadd/5</th>
<th>wtvwbtn/3</th>
<th>wtvwdel/2</th>
<th>wtvwget/4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Window Handling: Dialogs

This group of predicates provides support for user-defined dialogs, as well as a number of pre-defined "common dialogs" for activities such as opening and saving files, setting up print jobs, and displaying messages.

<table>
<thead>
<tr>
<th>bdsbox/2</th>
<th>brsbox/2</th>
<th>call_dialog/2</th>
<th>fntbox/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>hide_dialog/1</td>
<td>message_box/3</td>
<td>msgbox/4</td>
<td>opnbox/5</td>
</tr>
<tr>
<td>prgbox/1</td>
<td>prgbox/3</td>
<td>prnbox/4</td>
<td>rgbbox/4</td>
</tr>
<tr>
<td>savbox/5</td>
<td>seabox/1</td>
<td>show_dialog/1</td>
<td>size_dialog/2</td>
</tr>
<tr>
<td>tipbox/2</td>
<td>wait_dialog/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Window Handling: Environment Features

This group of predicates provides support for environment features.

<table>
<thead>
<tr>
<th>history/1</th>
<th>rich_print/5</th>
<th>rich_syntax/1</th>
<th>rich_syntax/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>status_bars/1</td>
<td>system_menu/3</td>
<td>word_wrap/1</td>
<td></td>
</tr>
</tbody>
</table>

Windows Sockets (Winsock) Predicates

This group comprises a carefully designed set of predicates which are used to access Windows Sockets (Winsock) functions, allowing WIN-PROLOG applications to have simple and powerful control over TCP/IP networks.

<table>
<thead>
<tr>
<th>sclose/1</th>
<th>sckhdl/2</th>
<th>screate/2</th>
<th>sdct/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket_handler/2</td>
<td>socket_handler/3</td>
<td>srecv/2</td>
<td>ssend/2</td>
</tr>
<tr>
<td>sstat/2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Musical Instrument Digital Interface (MIDI) Predicates

This group comprises a set of predicates which are used to access Musical Instrument Digital Interface (MIDI) functions, allowing WIN-PROLOG applications to record, play back, analyse and process music data from synthesizers and other musical instruments.
Undocumented Predicates

This final group comprises a set of predicates which are otherwise undocumented in the present manual, and is included to help the user avoid predicate name clashes between the WIN-PROLOG system and his or her programs.

<table>
<thead>
<tr>
<th>1/1</th>
<th>1/2</th>
<th>'!MAIN'?/0</th>
<th>addcls/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>addprd/3</td>
<td>addflg/3</td>
<td>atmchr/2</td>
<td>clrbox/5</td>
</tr>
<tr>
<td>cmpbox/2</td>
<td>debug/1</td>
<td>delcls/2</td>
<td>delflg/3</td>
</tr>
<tr>
<td>delprd/2</td>
<td>edop/3</td>
<td>erase_status_box/0</td>
<td>free/1</td>
</tr>
<tr>
<td>fun/3</td>
<td>generate_message/3</td>
<td>getcls/2</td>
<td>getflg/3</td>
</tr>
<tr>
<td>getpid/3</td>
<td>gfxdev/4</td>
<td>hshprd/7</td>
<td>idxcls/3</td>
</tr>
<tr>
<td>key_hook/2</td>
<td>lstprd/2</td>
<td>main_hook/0</td>
<td>meta_system/2</td>
</tr>
<tr>
<td>print/2</td>
<td>print/3</td>
<td>print_message/2</td>
<td>print_message_lines/3</td>
</tr>
<tr>
<td>printq/1</td>
<td>printq/2</td>
<td>printq/3</td>
<td>register_class/7</td>
</tr>
<tr>
<td>save_state/1</td>
<td>slist/1</td>
<td>sload/1</td>
<td>status_box/1</td>
</tr>
<tr>
<td>stratm/2</td>
<td>strchr/2</td>
<td>timer/2</td>
<td>timer_hook/3</td>
</tr>
<tr>
<td>wlbxadd/3</td>
<td>wlbxdel/2</td>
<td>wlbxfnd/4</td>
<td>wlbxget/3</td>
</tr>
<tr>
<td>wlbxsel/3</td>
<td>xfnbox/4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alphabetical Reference

The remainder of this manual, up to the Appendices, contains detailed descriptions of each of the WIN-PROLOG predicates, listed in alphabetical order. For a topic based discussion of these predicates you should refer to the appropriate "Programming Guide".
"conjunction": test that two goals are true

\[ \text{Call1} , \text{Call2} \]

+Call1 \quad \text{<goal>}
+Call2 \quad \text{<goal>}

**Comments**  
This predicate succeeds if both of its goals, \text{Call1} and \text{Call2}, are true (succeed). Either call may itself be a conjunction or disjunction, or a simple goal.

**Examples**  
The first example simply calls two goals, and succeeds because both are true:

\[ ?- \text{integer( 3 ), float( 3.14 ).} \]

yes

The second example calls three goals, but fails because the final one is false:

\[ ?- \text{integer( 3 ), float( 3.14 ), atom(X).} \]

no

**Notes**  
The \\
\text{',/2} \] predicate is used more than almost any other in Prolog, since it forms the primary link between successive goals in both programs and commands.

Any cut (call to \text{!/0}) that occurs in either goal affects execution in exactly the same way as if it had occurred outside the conjunction.
"cut": control backtracking

The predicate `/0 always succeeds, but it has the important side effect of removing pending choice points within the current clause and relation. Its purpose is to enable programs to commit to solutions, and to retrieve space that would otherwise be tied up in unnecessary choice points.

Consider the following program:

```prolog
foo( X, Y ) :-
    ( X = 1
    ; X = 2
    ),
    ( Y = 3
    ; Y = 4
    ).

foo( 5, 6 ).
```

If you were to call this program with a pair of variables, it would return a total of five solutions during backtracking:

```prolog
?- foo( X, Y ).
X = 1 ,
Y = 3 ;
X = 1 ,
Y = 4 ;
X = 2 ,
Y = 3 ;
X = 2 ,
```
Now consider a similar program, this time with a "cut" (call to !/0) inserted midway through the first clause:

```
bar( X, Y ) :-
  ( X = 1
    ; X = 2
  ),!
  ( Y = 3
    ; Y = 4
  ).

bar( 5, 6 ).
```

The cut prevents backtracking to any parts of the clause that occur before (above) it, and to any clauses that have not yet been tried. In this case, it limits the program to two solutions:

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

```
X = 1 ,
Y = 3 ;
X = 1 ,
Y = 4
```

Note that you can still explicitly check for solutions which were not returned above; for example:

```
?- bar( 2, 3 ).
```

```
yes
```

The cut will only affect choice points not yet tried at the time !/0 is called.
Notes

The cut is one of the most widely misused features in Prolog. Logically, cut should be used simply to "commit" to a given solution when the program(mer) knows that this is the required result. From a practical view point, committing to a result frees up valuable memory resources, and may, for example, be necessary in order to allow a loop to run indefinitely.

What is to be avoided is the random peppering of code with cuts, because it "seems a good idea". At best, this will simply slow execution, but at worst, it can cause programs to behave in unexpected and difficult to understand ways. Please see Appendix H for a more detailed examination of the cut and other memory management issues.

Any cut (call to !/0) that occurs within either goal of a conjunction (’,/2) or disjunction (;/2) affects execution in exactly the same way as if it had occurred outside the construct. The same applies to the "then" goal of the "if:then" (->/2) and the "then" and "else" goals of the "if:then:else" (;/2) constructs. Note that cuts should not normally be placed in the "if" goal in either of these constructs: this goal is automatically deterministic, and the scope of any such cuts are local to it.
The ->/2 predicate provides an "if:then" control construct, and succeeds if the If goal succeeds and the Then goal also succeeds. It differs from ',/2 in that it performs a "local cut" after executing its first goal, preventing backtracking into it. This is shown in the example below.

The main use of this predicate is to perform simple "if:then" programming. For example, consider the program:

```prolog
greeting( Word ) :-
   ( member( Word, [hello,hi,good_morning] )
    -> write( Word ),
            write( ' is a greeting!' )
   ).

greeting( _ ) :-
   write( 'All Done!' ).
```

The "if" goal tests whether a given word is in a list of greetings, and if so, runs the "then" goal to perform some output. Because of the call to ->/2, no backtracking will occur within the call to member/2. However, unlike with the cut (!/0), backtracking into the second clause of greeting/1 is still possible.

Because of their relative operator precedences (1000 and 1050 respectively), the conjunction operator "," binds more strongly than "->": thus, a term such as:

```
foo, bar -> sux, you
```

is equivalent to the term:

```
(foo, bar) -> (sux, you)
```
Please note that "if:then:else", which is written "if-then;else", is actually a feature of the disjunction predicate ;/2, and not a special case of ->/2. Once again, this is down to operator precedences: the disjunction operator ;, which is set at 1100, binds less strongly than ":->", and so a term such as:

```
foo, bar -> sux, you ; too, far
```

is equivalent to the term:

```
((foo, bar) -> (sux, you)) ; (too, far)
```

Any cut (call to !/0) that occurs in the "then" goal affects execution in exactly the same way as if it had occurred outside the "if:then" construct. Note that cuts should not normally be placed in the "if" goal, which is automatically deterministic; however, the scope of any such cuts will be local to this goal.
"disjunction": test whether either of two goals are true

Either ; Or

+Either <goal>
+Or <goal>

Comments

This predicate succeeds if either or both of its goals, Either and Or, are true (succeed). Either call may itself be a conjunction or disjunction, or a simple goal.

A special case of ;/2 supports the "if:then:else" construct (see notes below). The goal:

(if -> then ; else)

succeeds either if "if" is true, and so is "then", or if "if" is false and "else" is true. In this special case, the "->" symbol behaves as if it had performed a "local cut" after executing the "if" code, much as in the way the true "if:then" predicate, ->/2, works.

Examples

The first example simply tests two goals, and succeeds because although the first is false, the second is true:

?- integer( 3.14 ) ; float( 3.14 ).
yes

The second example can succeed in any of three ways, thanks to backtracking:

?- X = 1 ; X = 2 ; X = 3.

X = 1 ;
X = 2 ;
X = 3

One of the most common uses of disjunction is in its special "if:then:else" mode. Consider the program:
show_info( Number ) :-
    ( member( Number, [1,3,5,7,9] )
    -> write( 'small odd number' )
    ; write( 'some other number' )
    ).

This program tests whether a given number is on the list "[1,3,5,7,9]", and if so, performs a local cut, preventing further backtracking into member/2, before outputting the text "small odd number". For any other number or data item, the membership test fails, and the text "some other number" is output.

Notes
Please note that "if:then:else", which is written "if->then;else", is actually a feature of the disjunction predicate ;/2, and not a special case of ->/2. This is down to operator precedences: the ",","->" and ",;" operators have precedences of 1000, 1050 and 1100 respectively, and this governs their binding upon input so that a term such as:

    foo, bar -> sux, you ; too, far

is equivalent to the term:

    ((foo, bar) -> (sux, you)) ; (too, far)

Any cut (call to !/0) that occurs in the "either" or "or" goals of normal disjunction, or in the "then" or "else" goals of the "if:then:else" special case, affects execution in exactly the same way as if it had occurred outside the construct. Note that cuts should not normally be placed in the "if" goal, which is automatically deterministic; however, the scope of any such cuts will be local to this goal.
<2

**test if the result of one expression is less than another**

\[ E1 < E2 \]

**Comments**
Succeeds if the result of the arithmetic expression \( E1 \) is less than the result of the arithmetic expression \( E2 \).

**Examples**
The following command checks whether the result of evaluating the expression "22/7" is less than the number "3.5", and succeeds:

?\(-\) 22/7 < 3.5.

yes

The next example checks to see whether "4+5" evaluates to a result less than "3*3", and fails accordingly:

?\(-\) 4+5 < 3*3.

no

**Notes**
Either parameter may be a simple number, a mathematical expression, or a list with a number as its only element. It is not legal to provide either argument as an unbound variable: you must use the evaluation predicate, \( \text{is}/2 \), to return the computed results of mathematical expressions.
re-direct input from a file or a string

Call <- In

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

Comments If In is a file specification, the file is automatically opened if necessary, and all input performed by the Call is re-directed from that file; if In is a string, all input performed by the Call is re-directed from that string. If the Call is re-satisfiable, then on backtracking into the Call any further input is also re-directed from the given file or string.

Examples The following call performs a single read from a file called "foo.pl", opening it automatically if required:

?- read( T ) <- 'foo.pl'.
T = bar(sux)

This example assumed the file contained a term, "bar(sux).", and is directly equivalent to the call sequence:

?- seeing( X ), see( 'foo.pl' ), read( T ), see( X ).
X = user,
T = bar(sux)

Apart from being a short, convenient notation, the <-/2 predicate really comes into its own when used in conjunction with WIN-PROLOG strings:

?- read( X ) <- `foo( A ). `.
X = foo(_12345)

Notes Although <-/2 opens files automatically, it does not close them. After completing the final input from a file, it should be closed with a call to close/1, in order to free resources.

When reading Prolog terms from strings (or, indeed, files), please remember that the correct ending for a term is "dot-white-space", and not simply "dot". If you forget to place at least one white-space character after the period, you will get an error when attempting the read.
"univ": define the relationship between a term and a list

\[
\text{Term} = \ldots \text{List}
\]

?Term \hspace{1cm} \text{<term> or <variable>}

?List \hspace{1cm} \text{<list> or <variable>}

Comments

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.

Examples

The following call creates a compound term from the given list:

\[
?- X = \ldots [\text{likes}, \text{brian}, \text{prolog}].
\]

X = likes(brian, prolog)

When a single-element list is provided, an atomic term is returned:

\[
?- X = \ldots [123].
\]

X = 123

You can also convert a compound term into a list:

\[
?- \text{foo(bar)} = \ldots L.
\]

L = [foo, bar]

The conversion can work in both directions at once, filling in any variables automatically:

\[
?- \text{foo(123,A,789,bar)} = \ldots [X, 123, 456|Y].
\]

A = 456 ,

X = foo ,

Y = [789, bar]
The `=../2` predicate is usually called "univ", pronounced "you-neeve", and is widely used in meta-programming, where terms representing calls are pulled apart, modified as lists, and rebuilt into terms which are then executed.
equals": test whether two terms unify

\[ \text{Term1} = \text{Term2} \]

?Term1 <term>
?Term2 <term>

Comments
This predicate succeeds if terms Term1 and Term2 unify.

Examples
The following example tests that two given lists can be unified, and binds the variables as required:

\[ ?- [1,X,Y] = [Z,2,3]. \]
\[ X = 2 , \]
\[ Y = 3 , \]
\[ Z = 1 \]

The following call fails, because the two lists cannot be unified (the heads of the lists are "$1" and "$4" respectively:

\[ ?- [1,X,Y] = [4,2,3]. \]
\[ no \]

Notes
The \( =/2 \) predicate is possibly the simplest of all Prolog programs, and is effectively defined by the clause:

\[ X = X. \]

Unification should not be confused with tests for term identity (see \( ==/2 \)), numerical equality (see \( =:=/2 \)) or arithmetic evaluation (see \( \text{is}/2 \)).
test if the results of two expressions are numerically equal

E1 =:= E2

Comments
Succeeds if the arithmetic expressions E1 and E2 both evaluate to the same value.

Examples
The following command checks whether the result of evaluating the expression "22/7" is the same as the number "3.5", and fails:

?- 22/7 =:= 3.5.
no

The next example checks to see whether "4+5" evaluates to the same result as "3*3", and succeeds accordingly:

?- 4+5 =:= 3*3.
yes

Notes
Either parameter may be a simple number, a mathematical expression, or a list with a number as its only element. It is not legal to provide either argument as an unbound variable: you must use the evaluation predicate, is/2, to return the computed results of mathematical expressions.
test if one expression is less than or equal to another

\[ E_1 =\leq E_2 \]

Comments: Succeeds if the result of the arithmetic expression \( E_1 \) is less than or equal to the result of the arithmetic expression \( E_2 \).

Examples: The following command checks whether the result of evaluating the expression "22/7" is less than or equal to the number "3.5", and succeeds:

\[ ?- \ 22/7 =\leq 3.5. \]

yes

The next example checks to see whether "4+5" evaluates to a result less than or equal to "3*3", and succeeds accordingly:

\[ ?- \ 4+5 =\leq 3*3. \]

yes

Notes: Either parameter may be a simple number, a mathematical expression, or a list with a number as its only element. It is not legal to provide either argument as an unbound variable: you must use the evaluation predicate, \( is/2 \), to return the computed results of mathematical expressions.
==/2

check that two terms are identical

Term1 == Term2

?Term1 <term>
?Term2 <term>

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

Examples  The following call succeeds because both given terms are identical:

?- foo(Q,W) == foo(Q,W).
Q = _ ,
W = _

The following call fails, because the terms are not identical:

?- foo(Q,a) == foo(Q,W).
no

Notes  The ==/2 predicate tests that two terms are already identical: it does not perform unification, and so does not bind any variables. Other predicates should be used to test unification (see =/2) or numerical equality (see =:=/2).
E1 \=\= E2

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

Examples
The following command checks whether the result of evaluating the expression "22/7" is different from the number "3.5", and succeeds:

?- 22/7 \=\= 3.5.
yes

The next example checks to see whether "4+5" evaluates to a different result than "3*3", and fails accordingly:

?- 4+5 \=\= 3*3.
no

Notes
Either parameter may be a simple number, a mathematical expression, or a list with a number as its only element. It is not legal to provide either argument as an unbound variable: you must use the evaluation predicate, is/2, to return the computed results of mathematical expressions.
test if the result of one expression is greater than another

\[ E1 > E2 \]

+\(E1\) <expr>
+\(E2\) <expr>

Comments
Succeeds if the result of the arithmetic expression \(E1\) is greater than the result of the arithmetic expression \(E2\).

Examples
The following command checks whether the result of evaluating the expression "22/7" is greater than the number "3.5", and fails:

\(?- 22/7 > 3.5.\)
\(\text{no} \quad \text{<enter>}\)

The next example checks to see whether "4+5" evaluates to a result greater than "3*3", and fails accordingly:

\(?- 4+5 > 3*3.\)
\(\text{no} \quad \text{<enter>}\)

Notes
Either parameter may be a simple number, a mathematical expression, or a list with a number as its only element. It is not legal to provide either argument as an unbound variable: you must use the evaluation predicate, \(\text{is}/2\), to return the computed results of mathematical expressions.
test if one expression is greater than or equal to another

\[
E1 \geq E2
\]

+\(E1\) <expr>
+\(E2\) <expr>

**Comments**  Succeeds if the result of the arithmetic expression \(E1\) is greater than or equal to the result of the arithmetic expression \(E2\).

**Examples**  The following command checks whether the result of evaluating the expression "22/7" is greater than or equal to the number "3.5", and fails:

\[
?- 22/7 \geq 3.5. \\
\text{no}
\]

The next example checks to see whether "4+5" evaluates to a result greater than or equal to "3*3", and succeeds accordingly:

\[
?- 4+5 \geq 3*3. \\
\text{yes}
\]

**Notes**  Either parameter may be a simple number, a mathematical expression, or a list with a number as its only element. It is not legal to provide either argument as an unbound variable: you must use the evaluation predicate, \(is/2\), to return the computed results of mathematical expressions.
test if one term is less than another

Term1 @< Term2

?Term1 <term>
?Term2 <term>

Comments
Succeeds if Term1 is less than Term2 according to the "standard ordering of terms" (see notes below).

Examples
The following commands check a variety of terms, succeeding and failing as appropriate depending upon their standard ordering relationship:

?- foo @< bar. <enter>
no

?- bar @< bar. <enter>
no

?- 123 @< bar. <enter>
yes

?- foo(123) @< foo(bar). <enter>
yes

?- 4+5 @< 3*3. <enter>
no

Notes
The standard ordering of terms uniquely defines the relationship between any two terms: full details of standard ordering can be found in Appendix A, but here is a brief summary:

If two terms are of different types, comparison is simply defined by the relationship between these types (atoms are "less" than strings, etc). If two terms have the same type, and are atomic (numbers, atoms, etc), comparison is based on their value. If two terms have the same type, but are compound (terms, lists, etc), comparison is based recursively first on their respective heads, and then on their tails.
Please note that the "@*/2" comparison predicates do not evaluate numerical expressions, but rather compare the actual structures representing these expressions: the equivalent "+/2" predicate should be used for evaluative comparison.
@=/2

**test if one term is the same as another**

```
Term1 @= Term2
```

?Term1 <term>
?Term2 <term>

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

**Examples**  
The following call succeeds because both given terms are identical:

```
?- foo(Q,W) @= foo(Q,W).
Q = _ ,
W = _
```

The following call fails, because the terms are not identical:

```
?- foo(Q,a) @= foo(Q,W).
no
```

**Notes**  
The @=/2 predicate tests that two terms are already identical: it does not perform unification, and so does not bind any variables. Other predicates should be used to test unification (see =/2) or numerical equality (see =:=/2).

This predicate is identical to ==/2, but is included for the sake of completeness.
@test if one term is less than or equal to another

\[
\text{Term1 } \mathbin{@=}\mathbin{\langle} \text{Term2}
\]

?\text{Term1} \quad \langle\text{term}\rangle
?\text{Term2} \quad \langle\text{term}\rangle

Comments  Succeeds if Term1 is less than or equal to Term2 according to the "standard ordering of terms" (see notes below).

Examples  The following commands check a variety of terms, succeeding and failing as appropriate depending upon their standard ordering relationship:

\[
?\text{- foo } \mathbin{@=}\mathbin{\langle} \text{bar}.
\]
no

\[
?\text{- bar } \mathbin{@=}\mathbin{\langle} \text{bar}.
\]
yes

\[
?\text{- 123 } \mathbin{@=}\mathbin{\langle} \text{bar}.
\]
yes

\[
?\text{- foo(123) } \mathbin{@=}\mathbin{\langle} \text{foo(bar)}.
\]
yes

\[
?\text{- 4+5 } \mathbin{@=}\mathbin{\langle} 3*3.
\]
no

Notes  The standard ordering of terms uniquely defines the relationship between any two terms: full details of standard ordering can be found in Appendix A, but here is a brief summary:

If two terms are of different types, comparison is simply defined by the relationship between these types (atoms are "less" than strings, etc). If two terms have the same type, and are atomic (numbers, atoms, etc), comparison is based on their value. If two terms have the same type, but are compound (terms, lists, etc), comparison is based recursively first on their respective heads, and then on their tails.
Please note that the "@*/2" comparison predicates do not evaluate numerical expressions, but rather compare the actual structures representing these expressions: the equivalent "/2" predicate should be used for evaluative comparison.
@>/2

**test if one term is greater than another**

\[
\text{Term1 } @> \text{ Term2}
\]

?Term1  <term>
?Term2  <term>

**Comments**  Succeeds if *Term1* is greater than *Term2* according to the "standard ordering of terms" (see notes below).

**Examples**  The following commands check a variety of terms, succeeding and failing as appropriate depending upon their standard ordering relationship:

?- foo @> bar.  
yes

?- bar @> bar.  
no

?- 123 @> bar.  
no

?- foo(123) @> foo(bar).  
no

?- 4+5 @> 3*3.  
yes

**Notes**  The standard ordering of terms uniquely defines the relationship between any two terms: full details of standard ordering can be found in *Appendix A*, but here is a brief summary:

If two terms are of different types, comparison is simply defined by the relationship between these types (atoms are "less" than strings, etc). If two terms have the same type, and are atomic (numbers, atoms, etc), comparison is based on their value. If two terms have the same type, but are compound (terms, lists, etc), comparison is based recursively first on their respective heads, and then on their tails.
Please note that the "@*/2" comparison predicates do not evaluate numerical expressions, but rather compare the actual structures representing these expressions: the equivalent "*/2" predicate should be used for evaluative comparison.
@>=/2

test if one term is greater than or equal to another

Term1 @>= Term2

?Term1 <term>
?Term2 <term>

Comments Succeeds if Term1 is greater than or equal to Term2 according to the "standard ordering of terms" (see notes below).

Examples The following commands check a variety of terms, succeeding and failing as appropriate depending upon their standard ordering relationship:

?- foo @>= bar. <enter>
yes

?- bar @>= bar. <enter>
yes

?- 123 @>= bar. <enter>
no

?- foo(123) @>= foo(bar). <enter>
no

?- 4+5 @>= 3*3. <enter>
yes

Notes The standard ordering of terms uniquely defines the relationship between any two terms: full details of standard ordering can be found in Appendix A, but here is a brief summary:

If two terms are of different types, comparison is simply defined by the relationship between these types (atoms are "less" than strings, etc). If two terms have the same type, and are atomic (numbers, atoms, etc), comparison is based on their value. If two terms have the same type, but are compound (terms, lists, etc), comparison is based recursively first on their respective heads, and then on their tails.
Please note that the "@*/2" comparison predicates do not evaluate numerical expressions, but rather compare the actual structures representing these expressions: the equivalent "*/2" predicate should be used for evaluative comparison.
`@\=/2

  test if one term is not the same as another

  Term1 @\= Term2

  ?Term1 <term>
  ?Term2 <term>

Comments
This predicate succeeds if Term1 is not identical to Term2.

Examples
The following call succeeds, because the two specified terms are different from each other:

    ?- foo(bar) @\= foo(sux).
    yes

The next call also succeeds, because even though the two terms could unify, nonetheless they are different:

    ?- foo(bar) @\= foo(X).
    X = _

Notes
The `@\=/2` predicate is the exact opposite of `@=/2`, and is effectively defined as:

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

It is important to note that this is not a test for non-unification: even if two terms could be unified, unless they are already identical, they will fail in connection with this predicate.

This predicate is identical to `\=/2`, but is included for the sake of completeness.
The \+/1 predicate never binds variables: if the given goal fails, then nothing is bound by definition; if the goal succeeds and binds some variables, then \+/1 fails, and the bindings are undone:

?- \+ (X = foo, fail).
X = _

?- \+ (X = foo, true).
no

Although variables are unbound by \+/1, note that any side effects are left in place:

?- \+ (write( hello ), nl).
hello
no
Notes

Care should be taken to distinguish `\+/1`, which implements "negation as failure", from `not/1`, which implements "logical negation". In the former case, if the goal being tested succeeds by binding one or more variables, `\+/1` simply fails, undoing the bindings; in the latter case, `not/1` generates an error.


\=/2

check that two terms do not unify

\( \text{Term1} \ \neq \ \text{Term2} \)

?Term1 <term>
?Term2 <term>

Comments
This predicate succeeds if Term1 does not unify with Term2, and never binds any variables.

Examples
The following call succeeds, because the two specified terms do not unify:

?- \( \text{foo(bar)} \ \neq \ \text{foo(sux)} \).
yes

The following call fails, because although the given terms are different, they can unify:

?- \( \text{foo(bar)} \ \neq \ \text{foo(X)} \).
no

Notes
The \=/2 predicate is the exact opposite of =/2, and is effectively defined as:

\( X \ \neq \ Y : - \\
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ + \ X = Y. \)

It is important to note that this is not a test for inequality: two very different terms might still be able to unify, and as such will fail in connection with this predicate.
check that two terms are not identical

Term1 \== Term2

?Term1 <term>
?Term2 <term>

Comments  This predicate succeeds if Term1 is not identical to Term2.

Examples  The following call succeeds, because the two specified terms are different from each other:

?- foo(bar) \== foo(sux).
  yes

The following call also succeeds, because even though the two terms could unify, nonetheless they are different:

?- foo(bar) \== foo(X).
  X = _

Notes  The \==/2 predicate is the exact opposite of ==/2, and is effectively defined as:

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

It is important to note that this is not a test for non-unification: even if two terms could be unified, unless they are already identical, they will fail in connection with this predicate.
"existential quantification": there exists an $X$ such that $Y$

\[ X ^ \land Goal \]

?X  <term>
+Goal  <goal>

**Comments**

Logically speaking, the goal "$X \land Goal" can be read as: "there exists an $X$ such that $Goal$ is true". This "existential quantification" is not really programmed into $^/2$ at all: normally this predicate simply calls the given $Goal$, completely ignoring the value of $X$. The one exception to this rule is when this predicate is used in the second argument of calls to setof/3 and bagof/3: in these cases, the variable(s) in $X$ are used to partition sets of results.

**Examples**

The following command simply invokes the specified goal, and backtracks through its solutions. In effect the "$X \land" part of this goal is ignored:

?\[- X \land member( X, \{1,2,3\} ). \]
X = 1 ; 
X = 2 ; 
X = 3 ; 
no

**Notes**

This predicate is really a dummy placeholder, and is effectively implemented as:

\[ _ \land Goal :- 
   \]

\[ Goal. \]

In this respect, it is virtually identical to call/1. To see instances of where $^/2$ has an important special meaning, please refer to the entries for the setof/3 and bagof/3 predicates.
re-direct output to a file or a string

**Call ~> Out**

<table>
<thead>
<tr>
<th>+Call</th>
<th>&lt;goal&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Out</td>
<td>&lt;file_spec&gt; or &lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**

If `Out` is a file specification, the file is automatically created and/or opened if necessary, and all output performed by the `Call` is re-directed to that file; if `Out` is a variable, all output performed by the `Call` is re-directed to a string which is then bound to the variable. If the `Call` is re-satisfiable, then on backtracking into the `Call` any further output is also re-directed to the given file or string.

**Examples**

The following call performs a single write to a file called "foo.pl", creating and/or opening it automatically if required:

```
?- write( hello ) ~> 'foo.pl'.
yes
```

The above example is directly equivalent to the call sequence:

```
?- telling( X ), tell( 'foo.pl' ), write( hello), tell( X ).
X = user
```

Apart from being a short, convenient notation, the `~>/2` predicate really comes into its own when used in conjunction with variables, which become bound to **WIN-PROLOG** strings:

```
?- (write( hello ) ; write( world )) ~> X.
X = `hello` ;
X = `helloworld`
```

**Notes**

Although `~>/2` creates and/or opens files automatically, it does not close them. After completing the final output to a file, it should be closed with a call to `close/1`, in order to free resources.
abolish/1

_delete all predicates specified by the given argument_

abolish( Preds )

+Preds          <pred_specs>

Comments

This removes all trace of any predicates specified in Preds, freeing up memory resources for other programs. If the specified program(s) do(es) not exist, it simply succeeds.

Examples

The following call ensures that any predicate called foo/1 has been thoroughly deleted:

?- abolish( foo/1 ).
yes

A number of predicates can be deleted in a single call, using either a list or a conjunction:

yes

?- abolish( (bar/2,sux/3,you/4) ).
yes

Notes

Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, WIN-PROLOG immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are added to a program, they are inserted into memory immediately.

Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, WIN-PROLOG contains a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is a copy which is actually running. See the entry for dynamic/1, as well as Appendix B, to find out more about this feature.
abolish/2

*delete the predicate specified by the given functor and arity*

abolish( Functor, Arity )

+Functor  <functor>
+Arity  <arity>

**Comments**
This removes all trace of the single predicate, *Functor*/Arity, freeing up memory resources for other programs; if the specified program does not exist, it simply succeeds.

**Examples**
The following call ensures that any predicate called foo/1 has been thoroughly deleted:

?− abolish( foo , 1 ).

yes

**Notes**
This predicate is an archaic form: most applications use the abolish/1 predicate instead.

Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, **WIN-PROLOG** immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are added to a program, they are inserted into memory immediately.

Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, **WIN-PROLOG** contains a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is a copy which is actually running. See the entry for *dynamic*/1, as well as *Appendix B*, to find out more about this feature.
abolish_files/1

abolish all predicates associated with the given file

abolish_files( FileSpec )

+FileSpec <file_specs>

Comments

All predicates that were originally loaded from the file(s) specified by FileSpec are abolished (see abolish/1): effectively, this predicate is the reverse of consult/1.

Examples

The following sequence first loads a file, using consult/1, then runs a program within it, before finally removing all trace of the programs in the file:

?- consult( examples(easter) ).  
# 0.000 seconds to consult easter.pl [c:\pro386w\examples\]
yes

?- easter( 2000, Month, Day ).
Month = 4 ,
Day = 23

?- abolish_files( examples(easter) ).
yes

Programs from several files can be removed in a single call:

?- abolish_files( [foo,bar,prolog(sux)] ).
yes

Notes

The abolish_files/1 predicate relies on "housekeeping" that associates predicates with the files from which they were originally loaded. It cannot delete predicates which have been asserted dynamically at run time, unless the original file contained a dynamic/1 for that predicate. Even then, if the predicate has been deleted using abolish/1, and then re-created, its link to the original file will have been lost.

Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, WIN-PROLOG immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are
added to a program, they are inserted into memory immediately.

Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, **WIN-PROLOG** contains a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is a copy which is actually running. See the entry for `dynamic/1`, as well as Appendix B, to find out more about this feature.
**abort/0**

abort execution of the current program

**abort**

**Comments**
Abandons the program that is currently being executed and returns abruptly to the top level of Prolog.

**Examples**
The following command begins performing output, and is then aborted before it has a chance to finish:

```prolog
?- write( hello ), abort, write( world ).
    hello
```

Here is a simple error handler (see `?ERROR?/2`) which makes use of `abort/0` to terminate execution after printing a message to the console:

```prolog
'?ERROR?'( Number, Goal ) :-
    errmsg( Number, Message ),
    output( 0 ),
    write( error - Number - Message - Goal ),
    nl,
    abort.
```

**Notes**
Because the `abort/0` predicate directly resets all internal stacks and forces **WIN-PROLOG** to resume execution from "the top", it is normally only used to abandon execution of a query when an error has occurred for which there is no other method of recovery. It is also used in stand-alone applications to restart a system cleanly after each successful run.

In the **WIN-PROLOG** development environment, executing `abort/0` clears all stacks, before returning control either to a predicate called `abort_hook/0` or, if it exists, to a special user-defined "hook predicate" called `?ABORT?/0`.

When creating a stand-alone application, any arity-zero predicate can be nominated to receive control after execution of `abort/0`. This "abort hook" is specified in a compulsory field in the "Run/Application" dialog when the application is created.
user-defined hook to gain control after an abort

'\?ABORT?'

Comments
This is not a built-in predicate, but an optional hook that can be written by the user to gain control after a program has made a call to the \texttt{abort/0} predicate, or after an unhandled error condition has forced an abort.

Examples
Suppose a program is being developed which generates a database in a relation called "my\_data/2", and it is desired to automatically delete this database after any serious errors. The following definition of \texttt{'\?ABORT?/0} will achieve this:

\begin{verbatim}
'\?ABORT?':- 
    write( 'Abolishing my\_data/2...' ),
    abolish( my\_data/2 ),
    abort\_hook.
\end{verbatim}

Notes
The \texttt{'\?ABORT?/0} hook predicate is only significant during program development, while the development environment is active. In stand-alone applications, any arity-zero predicate can be nominated for the task of recovering after an abort. Because the requirements of a stand-alone application may differ from that of development-time testing, it is usually best to choose a predicate other than \texttt{'\?ABORT?/0} in applications.

In the absence of a user-supplied definition of \texttt{'\?ABORT?/0}, the \texttt{WIN-PROLOG} development environment will call the built-in predicate, \texttt{abort\_hook/0}. This performs various initialisations of its own, before returning control to the user via the "?-" prompt. It is a good idea to ensure that any definition of \texttt{'\?ABORT?/0} calls this hook as its final operation, as in the example above.
**abort_hook/0**

*default abort hook in development environment*

**abort_hook**

**Comments**
This predicate is called by the development environment to perform various initialisations after a program has aborted, either by a direct call to the `abort/0` predicate, or because of an unhandled error condition. It is normally only called directly from a user-defined abort hook (see `?ABORT?/0`).

**Examples**
This predicate should not normally be called directly, apart from in the very special case of a user-defined abort hook (see `?ABORT?/0`). Its sole function is to restart the development environment "supervisor loop", which displays the "?-." prompt, reads commands and shows solutions. As it is also run automatically by the development environment after a program aborts, it is easiest to demonstrate via a call to `abort/0`:

```
?- abort. <enter>
```

While you could have called `abort_hook/0` itself, this would have restarted the supervisor loop without first clearing out the execution stacks, effectively "locking up" some memory until the next call to `abort/0`.

**Notes**
The `abort_hook/0` predicate is only significant during program development, while the development environment is active. It must not be called in stand-alone applications: instead, any arity-zero predicate can be nominated for the task of recovering after an abort.
absolute_file_name/2

convert a logical file name to an absolute file name

absolute_file_name( LogFileName, AbsFileName )

+LogFileName <file_spec>
-AbsFileName <variable>

Comments

This predicate is used to convert "logical file names" to their equivalent "absolute file names". The logical file name can be anything from a simple atom defining a file, via a partially specified path, through to a complex term containing "path aliases"; the absolute file name is an atom which gives the entire device/drive/directory/file/extension of the file.

Where LogFileName is a given logical file name, AbsFileName will be bound to the given file's absolute file name, using simple rules outlined below (see "notes").

Examples

In the simplest case, an atom can be supplied as the logical file name; in this case, the current working directory is automatically added to it, together with any appropriate extension:

?- absolute_file_name( foo, A ).
A = 'c:\pro386w\foo'

Using path aliases, a term can be supplied; note that the "examples" alias is pre-defined (see file_search_path/2), and that because the file "salesman.pl" exists within this, the appropriate extension is added here:

?- absolute_file_name( examples(salesman), A ).
A = 'c:\pro386w\examples\salesman.pl'

Notes

The absolute_file_name/2 predicate is simply a default front-end to absolute_file_name/3, using a standard set of options. These options govern how the file extension is applied to the given file name, during its conversion from logical to absolute, using the following rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>If</td>
<td>an extension is specified in the logical file name, then use that extension</td>
</tr>
<tr>
<td>else if</td>
<td>an extension is not specified in the logical file name, and a Prolog source file (.PL) exists with the given name, then use this extension</td>
</tr>
</tbody>
</table>
else if an extension is not specified in the logical file name, and a Prolog object file (.PC) exists with 
the given name, then use this extension

else if an extension is not specified in the logical file name, and no file exists with the given name, 
then do not append any extension

The majority of traditional built-in file predicates (things like `see/1`, `tell/1`, etc., use logical file names, and make internal calls to `absolute_file_name/3` to convert these to absolute file names. However, some lower-level, `WIN-PROLOG` specific file predicates, namely those that reside directly in the "Prolog Kernel" (pro386w.exe) rather than the "Prolog Overlay" (pro386w.ovl), have no method of converting logical file names to absolute: these include predicates such as `dir/3`, `exec/3`, `fcreate/5`, etc.

The entries in this manual for all kernel-based file predicates make it clear that they can only accept atoms as file names; should it be necessary to mix these predicates with the traditional file predicates, it is a good idea to make explicit use of `absolute_file_name/2`, and then refer to the file’s absolute name throughout; for example:

```
delete_and_tell( LogFile ) :-
    absolute_file_name( LogFile, AbsFile ),
    close( AbsFile ),
    file( AbsFile, -1, Exists ),
    ( Exists = 0 
      -> write( 'File did not exist' ),
        nl
      ;  Exists = 1 
      -> del( AbsFile ),
        write( 'File has been deleted' ),
        nl
    ),
    tell( AbsFile ).
```

This program uses calls to `close/1` and `tell/1`, both of which are "traditional" file I/O predicates, able to take logical file names. However, it also calls `file/3` and `del/1`, which are assembler-coded predicates that live in the `WIN-PROLOG` kernel, and can only use atomic file names. By making an initial call to `absolute_file_name/2` to convert the given logical file name to absolute, and then referring to the latter thereafter, this program can safely mix both types of predicate.
absolute_file_name/3

convert a logical file name to absolute, using options

    absolute_file_name( LogFileName, Options, AbsFileName )

+LogFileName       <file_spec>
+Options           <list_of <tuple>>
-AbsFileName       <variable>

Comments  This predicate is used to convert "logical file names" to their equivalent "absolute file names", using various options. The logical file name can be anything from a simple atom defining a file, via a partially specified path, through to a complex term containing "path aliases"; the absolute file name is an atom which gives the entire device/drive/directory/file/extension of the file.

Where LogFileName is a given logical file name and Options is a list of zero or more of the following terms: ignore_underscores(Bool), extensions(Extn), file_type(Type), ignore_version(Bool), access(Mode), file_errors(Flag), solutions(Sltct), AbsFileName will be bound to the given file's absolute file name.

Examples  The following command checks whether the file "pro386w" exists in the current user directory, with one of the extensions ".exe" or ".ovl", and is readable:

    ?- absolute_file_name( pro386w, [extensions(["exe","ovl"]),access(read)], File ).  <enter>
    File = 'c:\pro386w\pro386w.exe'

To check if the Prolog file "trace.pl" exists in the "library" directory, try the command:

    ?- absolute_file_name( library(trace), [file_type(prolog),access(exist)], File ).  <enter>
    File = 'c:\pro386w\library\trace.pl'

Notes  The absolute_file_name/3 predicate works as a three phase process, in which each phase gets a possible solution from the preceding phase, and constructs one or more alternative solutions to be tested by the succeeding phases; these are:

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>syntactic rewriting</td>
</tr>
<tr>
<td>2</td>
<td>extension expansion</td>
</tr>
</tbody>
</table>
Each of the first two phases modifies the possible solution passed to it and produces alternatives that will be fed into the succeeding phases. The functionality of all phases except the first is governed by the options list. The 'access checking' phase checks if the generated file exists in the appropriate access mode, and if not asks for a new alternative from the preceding phases. 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. For a thorough description, see the description of each phase below.

Note that the logical file name argument `LogFileName` 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 the description of each phase below).

Normally `absolute_file_name/3` generates an error if a file is found that doesn't obey the access options given. This might not always be a desirable behaviour, since matching files that do obey the access options might be found later on in the search. The `file_errors(Flag)` option allows these cases to be treated as simple failure, as does setting the value of the "fileerrors" Prolog flag to "no", which can be achieved via a call to `prolog_flag/3` or `nofileerrors/0`. If failure is chosen via any of these options, the search space is guaranteed to be exhausted. If it is important to raise an error when the file doesn't exist, simply use the `access(exist)` option without any of the above failure options in place. Each option is discussed in turn in the tables that follow.

The `ignore_underscores(Bool)` option is provided for source-code compatibility with Quintus Prolog, and does nothing in the present WIN-PROLOG system:

<table>
<thead>
<tr>
<th>ignore_underscores(Bool)</th>
<th>In theory, this would remove all underscore characters (&quot;_&quot;) from the final file name; since these characters are significant in Windows file names, this option is ignored in WIN-PROLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>In theory, this would remove all underscore characters (&quot;_&quot;) from the final file name; since these characters are significant in Windows file names, this option is ignored in WIN-PROLOG</td>
</tr>
<tr>
<td>false</td>
<td>Do not delete underscores (default)</td>
</tr>
</tbody>
</table>

The `extensions(Ext)` option is ignored if `LogFileName` contains a file extension; otherwise it specifies an atom or list of atoms which define one or more file extensions to be tried in turn while constructing the file name:

<table>
<thead>
<tr>
<th>extensions(Ext)</th>
<th>Extn is an atom or a list of atoms, each atom naming an extension that should be tried in turn when constructing the absolute file name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extn</td>
<td>Extn is an atom or a list of atoms, each atom naming an extension that should be tried in turn when constructing the absolute file name.</td>
</tr>
</tbody>
</table>

The `file_type(Type)` option is ignored if `LogFileName` contains a file extension; otherwise it specifies an extension appropriate to the operating system, which makes programs using this option more portable than those relying on the `extensions(Ext)` option:
file_type(Type)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Implies extensions([]): LogFileName is a file without any extension. (Default)</td>
</tr>
<tr>
<td>prolog</td>
<td>Implies extensions([&quot;pl&quot;,&quot;pc&quot;,&quot;]): LogFileName is a Prolog source file (.pl), Prolog object file (.pc), or has no extension.</td>
</tr>
<tr>
<td>source</td>
<td>Implies extensions([&quot;pl&quot;,&quot;]): LogFileName is a Prolog source file (.pl), or has no extension.</td>
</tr>
<tr>
<td>object</td>
<td>Implies extensions([&quot;pc&quot;,&quot;]): LogFileName is a Prolog object file (.pc), or has no extension.</td>
</tr>
<tr>
<td>project</td>
<td>Implies extensions([&quot;pj&quot;,&quot;]): LogFileName is a Prolog project file (.pj), or has no extension.</td>
</tr>
<tr>
<td>overlay</td>
<td>Implies extensions([&quot;ovl&quot;,&quot;]): LogFileName is a Prolog overlay file (.ovl), or has no extension.</td>
</tr>
<tr>
<td>flex</td>
<td>Implies extensions([&quot;ksl&quot;,&quot;]): LogFileName is a flex source file (.ksl), or a has no extension.</td>
</tr>
</tbody>
</table>

The `ignore_version(Bool)` option is included simply for source code compatibility with Quintus Prolog:

<table>
<thead>
<tr>
<th>ignore_version(Bool)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>ignore incorrect file versions: this setting has no meaning, because the only WIN-PROLOG files which include version information are object and overlay files, and these are always checked for version integrity before being loaded.</td>
</tr>
<tr>
<td>false</td>
<td>raise an exception with incorrect object or overlay file versions (default).</td>
</tr>
</tbody>
</table>

The `access(Mode)` option specifies attributes that must be true of the given file:

<table>
<thead>
<tr>
<th>access(Mode)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>The file must be readable.</td>
</tr>
<tr>
<td>write</td>
<td>If the file exists, it must be writeable. If it doesn't exist, it must be possible to create it.</td>
</tr>
<tr>
<td>append</td>
<td>If the file exists, it must be writeable. If it doesn't exist, it must be possible to create.</td>
</tr>
<tr>
<td>exist</td>
<td>The file must exist.</td>
</tr>
<tr>
<td>none</td>
<td>If the extensions(Extn) option is specified, search for and return the absolute name of an actual file; otherwise, return the first possible absolute file name without checking for its existence. Note that if this option is specified, no existence exceptions can be raised (default).</td>
</tr>
</tbody>
</table>

The `file_errors(Flag)` option specifies whether to raise an error or simply to fail when the given file does not exist in the required access mode, unless the access mode is set to "none":

<table>
<thead>
<tr>
<th>file_errors(Flag)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**file_errors(Flag)**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>raise an error if the specified file does not exist in the required access mode (default)</td>
</tr>
<tr>
<td>fail</td>
<td>fail if the specified file does not exist in the required access mode</td>
</tr>
</tbody>
</table>

The solutions(Slct) option specifies whether the predicate should be deterministic, committing to the first successful file, or non-deterministic, allowing it to backtrack to alternatives:

**solutions(Slct)**

<table>
<thead>
<tr>
<th>Slct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>commit to and return the first suitable file name (default)</td>
</tr>
<tr>
<td>all</td>
<td>permit backtracking, returning alternative file names where possible</td>
</tr>
</tbody>
</table>

The syntactic rewriting phase translates the relative file specification given by LogFileName into the corresponding absolute file name.

If LogFileName is a compound term with one argument, it is interpreted as "<path_alias>(<file_spec>)": a call is made to file_search_path/2 to convert the path alias into a path specification. If file_search_path/2 has more than one solution, these are tried successively until an appropriate one is found. If no suitable match exists, an error is raised.

If LogFileName is the empty atom (''), the current working directory is returned. If absolute_file_name/3 is called from a goal in a file being loaded, the directory containing that file is considered to be the current working directory.

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Between the beginning of the file name and the end of the file name if there are no \s in the file name.</td>
</tr>
<tr>
<td>2</td>
<td>Between the beginning of the file name and the first .</td>
</tr>
<tr>
<td>3</td>
<td>Between any two successive -groups (where a -group is defined to be a sequence of one or more \s with no non-\ character interspersed.)</td>
</tr>
<tr>
<td>4</td>
<td>Between the last \ and the end of the file name.</td>
</tr>
</tbody>
</table>

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

The component ‘.’ is deleted together with the directory name syntactically preceding it. For example, ‘\a\b\..\c’ is rewritten as ‘\a\c’.

Two or more consecutive ‘\’s are replaced with one ‘\’.

When these rules have been applied, the resulting absolute file name is passed on to the next phase for further processing; if this subsequent phase fails, control backtracks into the current phase to allow it to suggest alternatives.

The extension expansion phase is now applied, following the rules outlined above in the `extensions(ext)` and `file_type(type)` options, and the resulting file name is passed on to the next phase for further processing; if this subsequent phase fails, control backtracks into the current phase to allow it to suggest alternatives.

Finally the access checking phase comes into play: again, this has been described above in the `access(mode)` option.

If an option is specified more than once the rightmost option takes precedence. This provides for a convenient way of adding default values by putting these defaults at the front of the list of options.

If `absolute_file_name/3` succeeds, and the file access option was one of "read", "write" or "append", it is guaranteed that the file can be opened with `open/2`. If the access option was "exist", the file is guaranteed to exist, but might be both read and write protected. Note that `absolute_file_name/3` signals a permission error if the specified file refers to a directory. Any such errors can be disabled by setting the "nofileerrors" flag: this causes `absolute_file_name/3` to fail, rather than raise an error, when generating file names with the wrong permissions. This feature allows backtracking through alternative file names until a match is found or the search space is exhausted.

The majority of traditional built-in file predicates (things like `see/1`, `tell/1`, etc) use logical file names, and make internal calls to `absolute_file_name/3` to convert these to absolute file names. However, some lower-level, `WIN-PROLOG` specific file predicates, namely those that reside directly in the "Prolog Kernel" (pro386w.exe) rather than the "Prolog Overlay" (pro386w.ovl), have no method of converting logical file names to absolute: these include predicates such as `dir/3`, `exec/3`, `fcreate/5`, etc. Please see the "notes" section in the entry for `absolute_file_name/2` for advice on how to work simultaneously with these two classes of predicate.
append/3

join or split arbitrary lists

append( First, Second, Whole )

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

Comments
This predicate succeeds when Whole is bound to a list consisting of the Second list appended to the First list. Any of the arguments may be fully or partially instantiated lists, or simply variables; append/3 can backtrack to generate alternative solutions where appropriate.

Examples
The following command simply joins two lists, "[1,2,3]" and "[a,b,c]", to give a new whole one:

?- append( [1,2,3], [a,b,c], W ).
W = [1,2,3,a,b,c]

Here is an example where the whole list is given, but the first and second ones are variables; on backtracking, each possible solution is offered in turn:

?- append( F, S, [1,2,3] ).
F = [],
S = [1,2,3] ;

F = [1],
S = [2,3] ;

F = [1,2],
S = [3] ;

F = [1,2,3],
S = [] ;

no
The `append/3` predicate is a classic Prolog program, and is widely used for joining and splitting lists; however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of `append/3`. In order to avoid errors, such references must be renamed or removed before loading files into WIN-PROLOG.
arg/3

find the nth argument of a term

arg(N, Term, Arg)

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

Comments This unifies Arg with the Nth argument of Term. N must be a positive integer, and Term must be a compound term. The arguments are numbered 1 upwards. The arg/3 predicate considers lists, conjunctions and disjunctions to be compound terms of arity 2.

Examples The following call returns the third argument from a term:

?- arg( 3, foo(one,two,three,four), A ).
A = three

Because traditional Prologs treat lists as arity-two compound terms of the functor ",", arg/3 can only be used to return the head or the tail of a list, and not individual elements:

?- arg( 1, [the,quick,brown,fox], H ).
H = the

?- arg( 2, [the,quick,brown,fox], T ).
T = [quick,brown,fox]

Notes In traditional Prolog systems, arg/3 was the only efficient way in which to access individual arguments in arbitrary terms. Furthermore, because in these primitive systems, lists were simply nested terms of the functor ",", this predicate could not be used to pluck out an individual list element.

The architecture of WIN-PROLOG is considerably more advanced than that of many Prolog implementations, and amongst other features has an efficient, true "list" data type. The arg/3 predicate, while useful for "Edinburgh" compatibility, is not as powerful as WIN-PROLOG's mem/3 predicate, which gives direct access to any part of any compound term, list, conjunction or disjunction, including the ability to address any element, even deep inside a nested data structure, in a single call.
test whether the given term is an arity

\[ \text{arity} \left( \text{Term} \right) \]

?Term <term>

**Comments**
This predicate succeeds if the given Term is an integer in the range \([0..128]\); otherwise it fails.

**Examples**
The following calls test various cases:

```prolog
?- \text{arity}( 123 ). <enter>
yes

?- \text{arity}( 456 ). <enter>
no

?- \text{arity}( \text{foo} ). <enter>
no

?- \text{arity}( \text{`bar`} ). <enter>
no

?- \text{arity}( X ). <enter>
no
```

**Notes**
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

In the current **WIN-PROLOG** system, a predicate may have anything between 0 and 128 arguments, and its argument count is known as its "arity". The predicate \text{arity}/1 is true of any integer in this range.

Future versions of **WIN-PROLOG** may well support predicates with more than the current maximum number of arguments. At this point, the definition of the present predicate would be extended to cover the larger integer range accordingly.
direct destructive assignment

asn( Target, Source )

?Target <term>
?Source <term>

Comments
This strictly non-logical predicate is used to perform efficient data accumulation during backtracking, avoiding the traditional uses of assert/1 and retract/1 for these purposes. If both Target and Source are variables, the former is bound to a special tuple structure, and the latter to an empty list: subsequent calls using the same Target, append copies of whatever term is in Source, to the list, even on backtracking. If Target is a single-element list containing a bound or unbound variable, this variable is assigned the term in Source.

Examples
The following initialises a list, and then appends items to it on backtracking:

?- asn( T, L ), member( X, [1,2,3] ), asn( T, X ).
T = _123() ,
L = [1] ,
X = 1 ;

T = _123() ,
L = [1,2] ,
X = 2 ;

T = _123() ,
L = [1,2,3] ,
X = 3 ;

no

The following call uses direct destructive assignment to replace the binding of an existing, bound variable:

?- X = 3.142, asn( [X], `hello world` ).
X = `hello world`
The `asn/2` violates the normal behaviour of Prolog, in that "bindings" of variables are undone on failure, providing a means to return data from failed branches of a computation, for an assortment of purposes. In most Prolog systems, the only way to pass data back from a failed call, is to assert it to some data relation, which can be checked later. An example of where you may want to do this, is when implementing a predicate like `findall/3`, which enumerates all solutions to a query using backtracking by enforced failure, and returns a list of all those force-failed solutions. Here is a pretty conventional implementation of a "find all" predicate:

```
find_all( Term, Goal, List ) :-
  dynamic( find_data/1 ),
  ( Goal,
    assert( find_data(Term) ),
    fail
  ;  find_build( List )
  ).

find_build( List ) :-
  ( retract( find_data(Head) )
    -> List = [Head|Tail],
    find_build( Tail )
  ;  List = []
  ).
```

In this example, "find_all/3" initialises a data relation, "find_data/1", then calls the given "Goal", asserting "Term" when it succeeds, before forcing it to fail and backtrack to any further solution. Once all solutions are exhausted, "find_build/1" is called to mop up all the saved solutions into a list.

While this approach works fine, and is relatively portable among Prolog implementations, it is not very efficient when you bear in mind that `assert/1` and `retract/1` are actually calls to the incremental compiler and decompiler respectively. Using `asn/2`, we can completely remove the need for a temporary data relation, and simply build the list of solutions directly in memory, on the fly:

```
find_all( Term, Goal, List ) :-
  asn( Hook, Info ),
  ( Goal,
    asn( Hook, Term ),
    fail
  ;  List = Info
  ).
```
The first call to \texttt{asn/2} binds "Hook" to a special tuple, and "Info" to an empty list. The subsequent call to \texttt{asn/2} uses information within the special tuple, "Hook", to locate and directly copy the given "Term" to the end of the list at "Info". When all solutions are complete, and the list at "Info" is fully populated, it is returned as the desired "List". The above example is effectively how the actual built-in predicate, \texttt{findall/3}, is implemented in \texttt{WIN-PROLOG}.

The other case of \texttt{asn/2}, where the first argument consists of a single-element list containing a variable, bound or otherwise, is useful for returning items such as the largest or smallest solution to a query, or perhaps even just a count of the number of solutions:

\begin{verbatim}
count( Goal, Sols ) :-
  Base = 0,
  ( Goal,
    rpn( [Base,1,+], More ),
    asn( [Base], More ),
    fail
  ; Sols = Base
  ).
\end{verbatim}

Here, "Base" is initialised to zero ("0"), and then the given "Goal" is called. If it succeeds, \texttt{rpn/2} is called to compute a value one "More" than "Base", and then \texttt{asn/2} is used to directly assign this value over "Base", before failure is forced, and backtracking allows the "Goal" to evaluate a subsequent solution. Once all solutions to "Goal" are exhausted, the value accumulated in "Base" is returned as the count of all solutions, "Sols":

\begin{verbatim}
?- count( (X = `hello` ; X = `world` ; member( X, [a,b,c] )), Sols ).
\end{verbatim}

\begin{verbatim}
X = _ ,
Sols = 5
\end{verbatim}

Again, in traditional Prolog implementation, things like counting solutions would require use of \texttt{assert/1} and \texttt{retract/1}, which would mean invoking the incremental compiler and decompiler respectively.

In all, \texttt{asn/2} should be considered a "dirty" predicate, one that violates the pure backtracking model of Prolog, but when used with care, as in when implementing predicates to accumulate or count solutions to a query, it is no more dirty than using \texttt{assert/1} and \texttt{retract/1}, or indeed writing to files or using any other method to preserve data through the failure process.
assert/1

add a clause to a dynamic predicate

\[ \text{assert( Clause )} \]

+Clause <clause>

Comments This adds a \textit{Clause} to a dynamic predicate at the end of its sequence of existing clauses, or if the predicate does not yet exist, declares it to be "dynamic", and consisting just of the given \textit{Clause}.

Examples The following calls assert the "\textit{fact}" that "pooh" likes "hunny", and then immediately test the new predicate:

\begin{verbatim}
?- assert( likes(pooh,hunny) ).
yes

?- likes( Who, What ).
Who = pooh ,
What = hunny
\end{verbatim}

Now we'll add a "\textit{rule}" to the existing predicate, saying that "brian" likes anything that "pooh" likes, and then test the predicate again:

\begin{verbatim}
?- assert( (likes(brian,X):-likes(pooh,X)) ).
X = _

?- likes( Who, What ).
Who = pooh ,
What = hunny ;

Who = brian ,
What = hunny
\end{verbatim}

Notes When adding a rule, it is necessary to parenthesise it, as shown in the second example above, in order to avoid a syntax error. This is because the "\textless-\textgreater" operator has a higher precedence (1200) than a normal argument list (1000).

Predicates can only be asserted to when they are declared "dynamic": when \texttt{assert/1} is called to add to a previously nonexistent predicate,
this declaration is performed automatically. However, if some clauses for a predicate have already been loaded from a file, the predicate may be considered "static", and any attempt to add new clauses to it will result in errors. If a source file contains the definition of a predicate which is to be further modified at run-time, this predicate should be declared dynamic explicitly with a call to \texttt{dynamic/1}; for example, consider the following file:

\begin{verbatim}
    :- dynamic foo/1.
    foo( hello ).
    bar( world ).
\end{verbatim}

If this file were to be loaded with a call to \texttt{consult/1}, it would be possible to assert additional clauses to the \texttt{foo/1} predicate, but not to \texttt{bar/1}.

It is never possible to assert to optimised predicates (those which have been compiled with the \texttt{optimising compiler}); were the above file to be optimised, for example with a call to \texttt{optimize\_files/1}, the \texttt{foo/1} predicate would be bypassed by the optimiser, leaving it dynamic.

Note that asserting a clause actually invokes a very efficient \texttt{incremental compiler}, which converts the clause into low-level Prolog virtual machine instructions. This means that even dynamically modified code is able to run at compiled code speeds.

Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, \texttt{WIN-PROLOG} immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are added to a program, they are inserted into memory immediately.

Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, \texttt{WIN-PROLOG} contains a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is a copy which is actually running. See the entry for \texttt{dynamic/1}, as well as Appendix B, to find out more about this feature.
assert/2

add a clause at the given position in a dynamic predicate

assert( Clause, Position )

+Clause <clause>
+Position <integer>

Comments
This adds a Clause to a dynamic predicate at the specified Position in its sequence of existing clauses, or if the predicate does not yet exist, declares it to be "dynamic", and consisting just of the given Clause.

Examples
The following calls assert the "fact" that "pooh" likes "hunny", and then immediately test the new predicate:

?- assert( likes(pooh,hunny), 1 ).
yes

?- likes( Who, What ).
Who = pooh ,
What = hunny

Now we'll add a "rule" at position "1" (the top) of the existing predicate, saying that "brian" likes anything that "pooh" likes, and then test the predicate again:

?- assert( (likes(brian,X):-likes(pooh,X)), 1 ).
X = _

?- likes( Who, What ).
Who = brian ,
What = hunny ;
Who = pooh ,
What = hunny

Notes
The assert/2 predicate allows clauses to be inserted into a relation, and is unique to WIN-PROLOG: other Prologs only allow clauses to be added at the beginning or end of a predicate. If the value of Position exceeds the number of clauses present, the new clause is placed at
the end of the relation; a value of zero directly references this location.

Please see the "notes" section in the entry for assert/1 for important additional information about dynamic predicates.
**asserta/1**

*add a clause at the top of a dynamic predicate*

```prolog
asserta( Clause )
+Clause <clause>
```

**Comments**

This adds a `Clause` to a dynamic predicate at the top of its sequence of existing clauses, or if the predicate does not yet exist, declares it to be "dynamic", and consisting just of the given `Clause`.

**Examples**

The following calls assert the "fact" that "pooh" likes "hunny", and then immediately test the new predicate:

```prolog
?- asserta( likes(pooh,hunny) ).
yes
?- likes( Who, What ).
Who = pooh ,
What = hunny
```

Now we'll add a "rule" to the top of the existing predicate, saying that "brian" likes anything that "pooh" likes, and then test the predicate again:

```prolog
?- asserta( (likes(brian,X):-likes(pooh,X)) ).
X = _
?- likes( Who, What ).
Who = brian ,
What = hunny ;
Who = pooh ,
What = hunny
```

**Notes**

Unlike `assert/1`, this predicate adds new clauses at the top (beginning) of a relation: this is useful when it is intended to call recently added clauses ahead of older ones. Please note that it is no more (and no less) efficient to add clauses to the top of a predicate rather than to the end: **WIN-PROLOG's incremental compiler** has direct access to both locations.
Please see the "notes" section in the entry for assert/1 for important additional information about adding clauses to and deleting them from dynamic predicates.
assertz/1

add a clause at the end of a dynamic predicate

```prolog
assertz( Clause )
+Clause <clause>
```

Comments
This adds a Clause to a dynamic predicate at the end of its sequence of existing clauses, or if the predicate does not yet exist, declares it to be "dynamic", and consisting just of the given Clause.

Examples
The following calls assert the "fact" that "pooh" likes "hunny", and then immediately test the new predicate:

```prolog
?- assertz( likes(pooh,hunny) ).
yes
?- likes( Who, What ).
  Who = pooh ,
  What = hunny
```

Now we'll add a "rule" to the end of the existing predicate, saying that "brian" likes anything that "pooh" likes, and then test the predicate again:

```prolog
?- assertz( (likes(brian,X):-likes(pooh,X)) ).
  X = _

?- likes( Who, What ).
  Who = pooh ,
  What = hunny ;
  Who = brian ,
  What = hunny
```

Notes
This predicate is identical in every respect to assert/1, and its only advantage is notational: its name makes it the natural opposite of asserta/1. Please note that it is no more (and no less) efficient to add clauses to the end of a predicate rather than to the top: WIN-PROLOG's incremental compiler has direct access to both locations.
Please see the "notes" section in the entry for assert/1 for important additional information about adding clauses to and deleting them from dynamic predicates.
at_end_of_file/0

test to see if the input stream pointer is at end-of-file

at_end_of_file

Comments
This predicate succeeds when the current input stream is at the end-of-file position, where subsequent attempts at input will normally result in failure or an error condition.

Examples
The following call checks whether the end of the input file "foo.pl" has been reached: if so, it closes the input stream; otherwise, it reads a term from the file (this example assumes the file was indeed at its end-of-file position):

?- see( foo ), (at_end_of_file -> seen ; read(X) ). <enter>
X = _

Notes
The at_end_of_file/0 predicate is normally used to preempt end-of-file errors or failure when reading from disk files. Most input predicates fail the first time they are called when the input stream is at end-of-file, and generate an error on subsequent attempts. As well as allowing programs to check for the end-of-file explicitly, at_end_of_file/0 clears the "error-next-time" flag, so a subsequent input attempt will once again fail.

In WIN-PROLOG, this predicate has an additional use. When reading from the "console window" (the "user" device), at_end_of_file/0 can be used to check whether there are any additional characters still to be read from the current command line, allowing programs to read the entire contents of a console command, but to stop short of requesting further user input.
at_end_of_line/0

test to see if the input stream pointer is at the end of a line

at_end_of_line

Comments  This predicate succeeds when the current input stream is at the end of a line, which is taken to mean either that the next character is a $<cr>$ (carriage return, character code 0Dh), or that the input stream is at end-of-file.

Examples  The following call checks whether the end of a line of input from the file "foo.pl" has been reached: if so, it closes the input stream; otherwise, it reads a term from the file (this example assumes the file was indeed at the end of a line):

   ?- see( foo ), (at_end_of_line -> seen ; read(X) ).
   X = _

Notes  The at_end_of_line/0 predicate takes a simplistic view of the meaning of "end of line", and will not, for example, recognise the end of line in a Unix file, where $<lf>$ (line feed, character code 0Ah) is used in place of the $<cr>$ character or $<cr><lf>$ pair used by the WIN-PROLOG console window ("user" device) or Windows/DOS files respectively.
atom/1

test whether the given term is an atom

atom( Term )

?Term <term>

Comments
This predicate succeeds if the given Term is a Prolog atom; otherwise it fails.

Examples
The following calls test various cases:

?- atom( 123 ).
   no <enter>  

?- atom( [1,2,3] ).
   no <enter>  

?- atom( [] ).
   yes <enter>  

?- atom( foo ).
   yes <enter>  

?- atom( `bar` ).
   no <enter>  

?- atom( X ).
   no <enter>

Notes
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

Please note that the special "empty list" atom, "[]", is not a true atom in WIN-PROLOG (see constant/1): for efficiency, this system uses a special data type to mark the ends of lists. It is therefore not possible to use "[]" to name predicates or files, and it should be noted that the following call fails:
?- [] = '[]'.
no

The `atom/1` predicate treats the empty list as an atom purely for compatibility with Quintus Prolog: in all other respects, this item should be thought of as a list with no elements.
atom_chars/2

convert between an atom and a character list

atom_chars( Atom, Chars )

?Atom <atom> or <variable>
?Chars <char_list> or <variable>

Comments
If Atom is instantiated to an atom, then Chars is unified with a list consisting of integers representing the character codes that make up the atom; otherwise, if Atom is an unbound variable and Chars is a list of character codes, a new atom is created and bound to Atom.

Examples
The following call converts an atom into a list of character codes:

?- atom_chars( foo, L ). <enter>
L = [102,111,111]

This call fills in the missing characters in a partially instantiated character list:

?- atom_chars( brian, [98,X,105|Y] ). <enter>
X = 114 ,
Y = [97,110]

An atom can be created from a fully instantiated list:

A = hello

Even if the list contains the characters of some other term, such as a number, it is an atom which is returned:

?- atom_chars( A, "123.45" ). <enter>
A = '123.45'

Notes
In all cases where the Atom is given, the Chars list may be as complete or incomplete as required; if Atom is a variable, then the Chars list must be fully instantiated: this predicate cannot generate alternative solutions based on partial information. Please note while the special "empty list" atom, "[]", is not a true atom in WIN-PROLOG, it is accepted by this predicate. See atom/0 for further information.
**atom_string/2**

*convert between an atom and a string*

```prolog
atom_string( Atom, String )
```

?Atom <atom> or <variable>

?String <string> or <variable>

**Comments**

If Atom is instantiated to an atom, then String is unified with a string consisting of the same characters; otherwise, if Atom is an unbound variable and String is instantiated to a string, a new atom is created and bound to Atom.

**Examples**

The following call converts an atom into a string:

```prolog
?- atom_string( foo, S ).
S = `foo`
```

This call checks that a string created from the given atom can be unified with the given string:

```prolog
?- atom_string( brian, `brian` ).
yes
```

An atom can be created from a string:

```prolog
?- atom_string( A, `hello` ).
A = hello
```

Even if the string contains the characters of some other term, such as a number, it is an atom which is returned:

```prolog
| ?- atom_string( A, `123.45` ).
A = '123.45'
```

**Notes**

In WIN-PROLOG, the term "string" is used to define a special text data type; in other Prologs, the same term is often used to refer to a "character list", in which text is stored a list of integer character codes. The string data type in WIN-PROLOG is not only compact and efficient, but extremely versatile. Please refer to the Windows Programming Guide for further information. Please note while the special "empty list" atom, "[]", is not a true atom in WIN-PROLOG, it is accepted by this predicate. See atom/0 for further information.
atomic/1

test whether a term is an atom, number or string

atomic( Term )

?Term <term>

Comments  This succeeds if Term is atomic, which means that it is currently bound to a number (integer or float), atom or string.

Examples  The following calls test various cases:

?- atomic( 123 ).
yes <enter>

?- atomic( [1,2,3] ).
no <enter>

?- atomic( foo ).
yes <enter>

?- atomic( `bar` ).
yes <enter>

?- atomic( X ).
no <enter>

Notes  This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

The term "atomic" means non-variable and non-compound, implying that the term is "indivisible". Just as discovered in the world of physics, atomic entities can indeed be split: in the case of Prolog, this operation has to be performed by a variety of term conversion predicates.
attrib/2

get or set file attributes

attrib( Filename, Attributes )

+Filename <atom>
?Attributes <variable> or <integer>

Comments
This predicate gets existing or sets new read-only, hidden, system and archive Attributes for a given Filename. The Attributes may be any additive combination of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'0001</td>
<td>Read-only</td>
</tr>
<tr>
<td>16'0002</td>
<td>Hidden</td>
</tr>
<tr>
<td>16'0004</td>
<td>System</td>
</tr>
<tr>
<td>16'0020</td>
<td>Archive</td>
</tr>
</tbody>
</table>

Examples
The following call sets a file, "foo", to be hidden with read/write permissions:

?- attrib( foo, 2 ).
yes

Notes
This predicate is limited to setting and checking just four file attribute bits, namely the read-only bit (16'0001), hidden bit (16'0002), system bit (16'0004) and archive bit (16'0020). It cannot be used to check or set the volume or directory bits (16'0008 and 16'0010 respectively), which are always masked, as are any more advanced attributes.

The file/3 predicate can be used to check all file attributes, but not to set them; further, the dir/3 predicate can use all attributes both for file inclusion and exclusion during its searches.

This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
bagof/3

return unsorted sets of terms in solutions of a given goal

\[ \text{bagof}( \text{Term}, \text{Goal}, \text{List} ) \]

?- Term <term>  
+Goal <goal>  
?List <variable> or <list>

Comments
This predicate returns unsorted sets of solutions that would normally be obtainable only by failing and backtracking through a query. It succeeds if \( \text{List} \) can be unified to a non-empty list of instances of \( \text{Term} \) such that \( \text{Goal} \) is true. The \( \text{Term} \) may be any Prolog term, and \( \text{Goal} \) may be any Prolog goal, with or without calls to the existential quantifier predicate, \( ^\exists/2 \), which is used to alter the partitioning of solutions (see "notes" below).

Examples
Consider the database relation:

\[
\begin{align*}
\text{foo}(1, c). \\
\text{foo}(2, a). \\
\text{foo}(1, b). \\
\text{foo}(2, a).
\end{align*}
\]

The following call returns sets of solutions of "X" in the goal "foo(Y,X)", partitioned according to different values of "Y":

?- bagof( X, foo(Y,X), Z ).
\[
\begin{align*}
X &= _ , \\
Y &= 1 , \\
Z &= [c,b] \\
X &= _ , \\
Y &= 2 , \\
Z &= [a,a] \\
\text{no}
\end{align*}
\]

If it is desired to get all solutions irrespective of the bindings of "Y", then "existential quantification" can be used:
?- bagof( X, Y^foo(Y,X), Z ).
X = _ ,
Y = _ ,
Z = [c,a,b,a]

Notes
The bagof/3 predicate is very closely related to setof/3. The only difference in behaviour is that the latter sorts the returned sets, removing duplicate solutions in the process.

The "existential quantification" feature of both bagof/3 and setof/3 often causes confusion. In general, solutions to both predicates are simply partitioned according to the bindings of those variables mentioned in the Goal but not in the Term. Consider the command:

?- bagof( (X,Y), foo(X,Y,A,B), Z ).

This would return a series of lists of (X,Y) solutions for the hypothetical predicate, foo/4, with each list corresponding to a different binding combination of (A,B). Supposing it was desired still to partition according to bindings of B, but not A, all that is required is to mention "A" in the Goal as the left operand of a call to ^/2:

?- bagof( (X,Y), A^foo(X,Y,A,B), Z ).

This time, the series of lists of (X,Y) solutions will be split only on different bindings of B: effectively, A will be ignored during the partitioning.

The "^" atom is defined as an infix, left-associate operator, so a series of variables can be mentioned simply by extending (deepening) the sequence of calls to ^/2:

?- bagof( (X,Y), A^B^foo(X,Y,A,B), Z ).

Please note that outside of its special interpretation in the bagof/3 and setof/3 predicates, ^/2 has no meaning apart from that of the goal that constitutes its second argument; because of this, the command:

?- A ^ foo(A,B).

is in every respect identical to the command:

?- call( foo(A,B) ).
One remaining solution set predicate, `findall/3`, returns all solutions to a goal, without partitioning or sorting: in this case, use of existential quantification is meaningless.
bdsbox/2

show or hide the banner display status box or record a screenshot

\[ \text{bdsbox(Text, Type)} \]

- \text{Text} \quad \text{<variable, <string> or <atom>}
- \text{Type} \quad \text{<variable, <integer> or <list>}

Comments

This predicate performs two distinct functions: when \text{Text} is a string, and \text{Type} is an integer, it shows or hides the \textsc{WIN-PROLOG} banner display status box, where the \text{Type} integer is as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>hide the banner display status box</td>
</tr>
<tr>
<td>0</td>
<td>show the the banner display status box with white text over image in fixed font</td>
</tr>
<tr>
<td>1</td>
<td>show the the banner display status box with black text beneath image in fixed font</td>
</tr>
<tr>
<td>2</td>
<td>show the the banner display status box with black text beneath image in variable font</td>
</tr>
</tbody>
</table>

When \text{Text} is an atom, and \text{Type} is a list of exactly four integers, a screenshot is saved to a \texttt{.BMP} (Microsoft Bitmap) file: the four integers define the Left, Top, Right and Bottom of a rectangle of the screen to be saved; values of zero ("0") are taken to mean the respective edge of the screen.

Examples

The following call displays the \textsc{WIN-PROLOG} welcome banner, with the words,"Hello World" displayed in the top left corner in white:

\[ \text{?- bdsbox(`Hello World`, 0).} \]

yes

The following calls pick up the current state, then hide the banner display status box:

\[ \text{?- bdsbox(Text, Type), bdsbox(``, -1).} \]

Text = `Hello World`

\text{Text} = `Hello World`,

Type = 0

The following call takes a screenshot of the entire screen, and saves it in a file called "foo.bmp":

\[ \text{?- bdsbox(``, -1).} \]
The `bdsbox/2` predicate is used by WIN-PROLOG both to display the welcome banner at start up, using `Text` set to an empty string, and `Type` set to zero ("0"), and also to display the "About Box", with `Text` initialised with all relevant text, and `Type` set to two ("2").

This predicate is further used in the preparation of documentation, both for some of the system manuals, and also for the LPA website, using a hitherto undocumented feature of the WIN-PROLOG development environment: when the "Z" switch (see `switch/2`) is set to a value in the range [0..99999], performing a right-click on any Prolog window while holding down shift and/or control, causes a bitmap file of the name, "bds?????.bmp" to be saved, where "?????" is 5 digit value of the current value of the "Z" switch, which is then incremented. There are three variations on how much of the screen is saved by this operation:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>save the entire screen</td>
</tr>
<tr>
<td>control</td>
<td>save the WIN-PROLOG main window area</td>
</tr>
<tr>
<td>shift+control</td>
<td>save the clicked window only</td>
</tr>
</tbody>
</table>

There are two ways to set the "Z" switch to enable this behaviour; either specify on the Windows command line, for example:

```c>
pro386w /z0
```

or set it up at any time with a command like this:

```?- switch( z, 0 ).
yes
```

This mechanism allows up to 100,000 screenshots to be taken during any one session, although of course it is possible to reset the counter at any time, and to any number beween 0 and 99999 inclusive, either to replace existing screenshots, or (after copying them), creating new ones.
beep/2

sound a beep of the given duration and frequency

\[ \text{beep}( \text{Frequency}, \text{Duration} ) \]

+Frequency <integer> in the range [0,20..20000]
+Duration <integer>

Comments
This predicate sounds a beep of the given Frequency (in Hz) and Duration (in ms) on the computer's internal speaker. If Frequency is set to zero, this predicate pauses silently for the specified Duration; otherwise a note is sounded.

Examples
The following call plays a 1-second note of "concert A", which is defined as having a frequency of 440Hz:

\[ \text{?} - \text{beep}( 440, 1000 ). \]

yes

Notes
The Duration is automatically truncated to a maximum of 1000 (ie, 1 second), because during the beep it is not possible to interrupt WIN-PROLOG. For backwards compatibility, values greater than this are accepted, but are treated as 1000.

The beep/2 works as described only on Windows setups without a soundcard: where one is present, the Duration and Frequency are ignored, and a standard Windows alert sound (typically a single bell chime) is played instead.
break/0

suspend the current execution and start a nested supervisor

break

Comments

This causes the currently-running program to suspend, displaying a "break" message and starting a nested copy of the "supervisor loop", allowing new commands to be entered at the usual "?-" prompt. To return to running the suspended program, the user simply presses the <escape> key.

Examples

Consider the program:

    foo :-
    write( hello ),
    nl,
    break,
    write( world ),
    nl.

If this program is run with the following command, it will display the word "hello", before suspending and displaying a break message:

    ?- foo.  <enter>
    hello
    # Break level 1
    [1]

The new "break level 1" supervisor is identified by the number "1", which appears in square brackets immediately above the traditional "?-" prompt. Any command or series of commands can now be entered, for example:

    ?- write( there ), nl.  <enter>
    there
    yes

To exit from a break loop, simply press <escape>, and the "end of file" indicator "~Z" will be shown, before the original program resumes,
printing the word "world":

?- ~Z
world
yes

Notes

The break/0 predicate is primarily used during debugging, where it allows the user to run local tests in the midst of running a larger program. For this reason, both the WIN-PROLOG source level and box model debuggers include a button, labeled "Break", which simply causes break/0 to be executed. This predicate should not be used in run-time applications.

Note that any call to the abort/0 predicate, either directly or through an unhandled error condition, causes all current break levels and suspended programs to be terminated immediately, returning control to the top-level supervisor prompt.

The break/0 predicate, together with references to "break levels", should not be confused with the similarly named predicates, break/1, 'BREAK?'/1 or break_hook/1: these latter two predicates are concerned with handling the <ctrl-del> key combination which allows users to interrupt program execution.
break/1

*get or set the keyboard break flag*

\[ \text{break( Flag )} \]

?Flag <variable> or <integer> in the domain \{0,1\}

**Comments**

This gets or sets the state of the keyboard break \( \text{Flag} \). When clear, this flag suppresses the acceptance of \(<\text{ctrl-del}>\) interrupts to Prolog programs, preventing the user from stopping a running query; when set, such interrupts are accepted and passed to any system- or user-defined break hook. The flag may have either of the following states:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>keyboard breaks not accepted</td>
</tr>
<tr>
<td>1</td>
<td>keyboard breaks accepted</td>
</tr>
</tbody>
</table>

**Examples**

The following call picks the current value of the keyboard break flag, then clears the flag (sets it to "0"), before calling some program "foo/0". Upon completion, it resets the flag to its original value:

\[ \text{- break( F ), break( 0 ), foo, break( F ).} \quad \text{<enter>} \]

\[ F = 1 \]

**Notes**

Initially, the keyboard break flag is set (has the value "1"): in this state it allows any \textit{WIN-PROLOG} program to be interrupted by typing the \(<\text{ctrl-del}>\) key combination. Any such event interrupts the current Prolog goal and calls a system- or user-defined "break hook" (see \texttt{break\_hook/1} and \texttt{?BREAK?/1} respectively). When this flag is clear ("0"), typing \(<\text{ctrl-del}>\) has no effect until the flag is next set (back to "1"): at this point, any previously-ignored keyboard break will occur.

Please note that while the keyboard break flag is disabled, there is no way to interrupt a program that is running: this predicate should only be used in fully-tested applications.

The \texttt{break/1}, \texttt{?BREAK?/1} and \texttt{break\_hook/1} predicates should not be confused with the similarly named \texttt{break/0}, together with references to "break levels": this latter predicate is concerned with handling nested supervisor prompts and suspended program execution during debugging.
user-defined hook to gain control after a break interrupt

'?BREAK?'( Goal )

?Goal <goal> or <variable>

Comments
This is not a built-in predicate, but an optional hook that can be written by the user to gain control after a "break interrupt", where a program has been interrupted by the user pressing the <ctrl-del> key combination. The interrupted goal is unified with Goal.

Examples
Suppose a program is being developed in which it is desired to allow break interrupts during calls to get/1, but not elsewhere: whenever this predicate is interrupted, the program should abort; otherwise, it should carry on regardless. The following definition of '?BREAK'/1 will achieve this:

    '?BREAK?'( get(_) ) :-
    abort.

    '?BREAK?'( Goal ) :-
    Goal.

Notes
The '?BREAK'/1 hook predicate is only significant during program development, while the development environment is active. In stand-alone applications, any arity-one predicate can be nominated for the task of recovering after a break interrupt. Because the requirements of a stand-alone application may differ from that of development-time testing, it is usually best to choose a predicate other than '?BREAK'/1 in stand-alone applications.

In the absence of a user-supplied definition of '?BREAK'/1, the WIN-PROLOG development environment will call the built-in predicate, break_hook/1. This displays a message box which gives the user the choice of whether to abort the evaluation, which would return control to the top-level supervisor prompt, or to continue running the interrupted program.

The break/1, '?BREAK'/1 and break_hook/1 predicates should not be confused with the similarly named break/0, together with references to "break levels": this latter predicate is concerned with handling nested supervisor prompts and suspended program execution during debugging.
**break_hook/1**

*default break hook in development environment*

```
break_hook( Goal )
```

+Goal <goal>

**Comments**

This predicate is called by the development environment to regain control after a "break interrupt", where program has been interrupted at the given Goal by the user pressing the <ctrl-del> key combination. It is normally only called directly from a user-defined break hook (see '?BREAK?'/1).

**Examples**

This predicate should not normally be called directly, apart from in the very special case of a user break hook (see '?BREAK?'/1). Its sole function is to display a message box which gives the user the options of aborting or continuing program execution. As it is also run automatically by the development environment after break interrupt occurs, it is easiest to demonstrate indirectly, by pressing <ctrl-del>:

```
?- <ctrl-del>
```

This will display a message box: if you press this dialog's "Yes" button, a call is made to abort/0, which has the effect of terminating all running programs and restarting the supervisor loop; otherwise if you press "No", program execution continues untouched.

**Notes**

The break_hook/1 predicate is normally only used during program development, while the development environment is active. While it can be called in stand-alone applications, most such systems have run-time requirements which differ from those at development time: any arity-one predicate can be nominated for the task of recovering after a break interrupt.

The break/1, '?BREAK?'/1 and break_hook/1 predicates should not be confused with the similarly named break/0, together with references to "break levels": this latter predicate is concerned with handling nested supervisor prompts and suspended program execution during debugging.
brsbox/2

display the "locate folder" dialog box

\[
\text{brsbox( Title, Folder )}
\]

+Title <term>
-Folder <variable>

Comments
This displays a standard system "browse box" (or "locate folder") dialog box, with the given Title, allowing the user to browse and select from the available folders. The dialog closes when the "OK" button is clicked or if <enter> is pressed, and Folder is bound to an atom denoting the user's chosen folder.

Examples
The following command displays the browser box, using the title, "Locate a Folder"; after the user has browsed the available folders and selected one, the final choice is returned (in this example, "c:\pro386w"):

\[
?- \text{brsbox( `Locate a Folder`, Folder ).} <\text{enter}>\\
\text{Folder = 'c:\pro386w'}
\]

Notes
Three common dialog box predicates are dedicated to obtaining filenames from the user: brsbox/2, opnbox/5, savbox/5. The first of these is used to return a folder name, while the latter pair share the majority of their features, but as their names suggest, one is used when opening files for loading (input), and the other when creating files for saving (output). This leads to the main differences in their respective behaviours, as detailed in the following paragraphs.

When loading, compiling or otherwise processing existing files, it is often desirable to be able to select several files at once; furthermore, it is essential that any such files are known to exist, and to be available for read access: opnbox/5 both allows multiple file selection, and performs its own existence and access permission checks on all selected files.

When "saving as" or otherwise creating new files, it is usually desirable to specify just one file at a time; furthermore, it is helpful to display a warning if a named (selected) file already exists, to avoid the user inadvertently overwriting important data: savbox/5 both limits selections to one file at a time, and displays a warning if the chosen file exists, giving the user the option to try again.

When searching for file locations other than when saving or loading, it can be handy to be able to select a folder directly, rather than one or more files contained within it: brsbox/2 provides exactly this feature. Please note that none of these predicates actually opens the file: it is up to the application to pass the resulting file names to predicates such as open/2, fcreate/5, see/1, tell/1, and so on.
**busy/1**

get or set the busy cursor and status

\[ \text{busy}( \text{Cursor} ) \]

?Cursor <variable> or <integer>

**Comments**

Get or set the Windows "busy cursor" and status. If Cursor is a variable, it will return the current busy cursor and status; otherwise, it is used to set the cursor and status. The cursor and status is set according to the following table:

<table>
<thead>
<tr>
<th>Cursor</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal cursor, user input enabled</td>
</tr>
<tr>
<td>1</td>
<td>hourglass (busy) cursor, user input disabled</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>handle of a cursor, user input disabled</td>
</tr>
</tbody>
</table>

**Examples**

The following call sets the busy cursor and status, and then performs a 1-second delay before clearing the busy cursor and status. The delay is created by calling `repeat/0` within `ms/2`, and testing that the resulting time exceeds 1000ms:

?- busy( 1 ), ms( repeat, T ), T > 1000, busy( 0 ). <enter>

\[ T = 1001 \]

**Notes**

The `busy/1` predicate is normally called to indicate period of activity during which user intervention is not required: whenever the cursor/status is set other than to zero, all user input is disabled apart from the "break interrupt" (see `?BREAK?/1` and `break_hook/1`).

Any valid cursor handle, represented as an integer, can be used to set "busy" status: these can be obtained, among other sources, from the `gfx_cursor_handle/2` predicate. User input can only be reenabled via the special zero "cursor" handle, or automatically by attempting input from the "console window" (user device).

In Windows 3.1, setting the busy cursor and status had the effect of locking input to all applications, and not just the current WIN-PROLOG session. Since Windows 95/98 and NT, the busy status has only affected the application which sets it.
internal predicate used in the expansion of grammar rules

'C'( List, Head, Tail )

?List <list> or <variable>
?Head <term> or <variable>
?Tail <list> or <variable>

Comments  The curiously-named 'C/3' predicate is not normally used directly in user programs, because all it does is match a List to its Head and Tail according to the following definition:

'C'( [Head|Tail], Head, Tail ).

Examples  The following call returns the head and tail of a list:

?- 'C'( [the,quick,brown,fox], H, T ).
H = the ,
T = [quick,brown,fox]

Notes  This predicate is used internally by Prolog when expanding "grammar rules" that contain "terminal symbols" (see the Programming Guide for information about grammar rules).
call/1

**call a goal, limiting the scope of any cut**

```prolog
call( Goal )
+Goal <goal>
```

**Comments**  
This calls the given Goal, and succeeds or fails depending upon whether the Goal itself succeeds or fails. It is defined simply as the program:

```prolog
call( Goal ) :- Goal.
```

**Examples**  
The following command invokes the member/3 predicate from within call/1:

```prolog
?- call( member(X,[1,2,3]) ).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

**Notes**  
The "meta-programming" features in **WIN-PROLOG** are extremely powerful, and any variable can be "called" directly providing it has first been bound to a valid goal:

```prolog
?- X = (write(hello),nl), X .
hello
X = (write(hello),nl)
```

It might well be asked that, given all this power, why is call/1 necessary? Apart from the need to maintain compatibility with other Prolog systems, the main reason for using call/1 is that it limits the scope of any "cut" (/0) to within the goal itself. Consider the following program, which will display all solutions to a given goal, followed by the message "That's all folks!":

```prolog
```
show_all( Goal ) :-
    Goal,
    writeq( Goal ),
    nl,
    fail.

show_all( Goal ) :-
    write( 'That~'s all folks!' ),
    nl.

To show how this program works, try the following command:

?- show_all( member(X,[1,2,3]) ).
member(1,[1,2,3])
member(2,[1,2,3])
member(3,[1,2,3])
That's all folks!
X = _

The program has displayed each of three solutions to a call to member/3. Now try the call:

?- show_all( (member(X,[1,2,3]),!) ).
member(1,[1,2,3]),!
no

The first solution has been shown as before, but because the given goal contained a cut, not only has member/3 been prevented from backtracking, but so has the show_all/1 program itself, which therefore fails, rather than show the "That's all folks!" message. The following variation of this program replaces the in-line meta-call to "Goal" with a call to the call/1 predicate:

call_all( Goal ) :-
call( Goal ),
    writeq( Goal ),
    nl,
    fail.

call_all( Goal ) :-
write( 'That~'s all folks!' ),
nl.

With this version of the program, the cut will only remove choice points within the goal being tested, and not throughout the entire program:

?- call_all( (member(X,[1,2,3]),!) ).
member(1,[1,2,3]),!
That's all folks!
X = _
call/2

call a goal, returning its exit port

call( Goal, Port )

+Goal <goal>
?Port <atom> or <variable>

Comments
This calls the given Goal, and succeeds, unifying Port to the "exit port" of the Goal. The exit port is one of three atoms as shown in the table below:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>the goal has succeeded, but has further choice points to try on backtracking</td>
</tr>
<tr>
<td>!</td>
<td>the goal has succeeded, and there are no further choice points to try</td>
</tr>
<tr>
<td>fail</td>
<td>the goal has failed: by definition all possible choice points have been exhausted</td>
</tr>
</tbody>
</table>

Examples
The following command backtracks a call to member/3, showing solutions and exit ports until the "fail" port is reached:

?- call( member(X,[1,2,3]), P ).
X = 1 ,
P = true ;
X = 2  ,
P = true ;
X = 3  ,
P = true ;
X = _ ,
P = fail

This variant of the call includes a "cut" (a call to !/0) to make member/3 deterministic, so only a single solution is reported, together with the exit port "done":

<enter>
<space>
<space>
<space>
?- call( (member(X,[1,2,3]),!), P ).
X = 1 ,
P = !

Notes

The call/2 predicate is mainly used when debugging and testing programs, and is especially useful when determining whether a given predicate is succeeding "cleanly", i.e. without leaving unnecessary choice points behind.

Although call/2 captures two types of success as well as failure, it does not pick up errors in calls. Another predicate, catch/2, can be used to perform this function.

Any cuts in the specified Goal are limited in scope to within the goal itself. Please see the "notes" entry for call/1 for more details about the scope of cut and meta-programming.

Up to version 5.000 of WIN-PROLOG, the first two atoms in the above table were "exit" (rather than "true"), "done" (rather than "!"). The names were changed in version 6.000 better to reflect their computational status, and to simplify the meta-programming in debuggers and profiler programs.
call_dialog/2

**call a modal dialog and get or check the result**

```
call_dialog( Window, Result )
```

+Window <window_handle>

?Result <term>

**Comments**

This shows the given dialog `Window`, disables all other top-level WIN-PROLOG windows, and waits for the user to complete the dialog: `Result` is unified with whatever value has been returned by the dialog's `window handler`.

**Examples**

The following program creates a very simple dialog with two buttons, labelled "Hello" and "World" respectively:

```
create_foo :-
    Dstyle = [ws_popup,ws_caption],
    Bstyle = [ws_child,ws_visible,bs_pushbutton],
    wdcreate( foo, `foo`, 100, 100, 170, 75, Dstyle ),
    wccreate( (foo,1), button, `Hello`, 10, 10, 70, 30, Bstyle ),
    wccreate( (foo,2), button, `World`, 90, 10, 70, 30, Bstyle ).
```

When compiled, and the following command is entered, the dialog "foo" is created, but not yet displayed:

```
?- create_foo.
yes
```

To show the dialog, and return the text of whichever of the two buttons was pressed, enter the following command:

```
?- call_dialog( foo, X ).
```

The dialog will appear, disabling all other WIN-PROLOG windows, and if the user presses the "Hello" button, its lowercase text will be returned as follows:

```
X = hello
```

**Notes**

The example shown above uses the default window handler, `window_handler/4`, which simply binds the lowercase text of any clicked but-
ton to the Result. It is the action of binding this variable which actually "completes" the dialog.

Window handlers are simply arity-4 programs which intercept messages destined for a window (including dialogs and their controls), allowing arbitrary actions to take place. Consider the following example:

```prolog
foo_handler( (foo,1), msg_button, _, Result ) :-
    time( _, _, _, H, M, S, _ ),
    Result = finished_at(H,M,S).

foo_handler( Window, Message, Data, Result ) :-
    window_handler( Window, Message, Data, Result ).
```

The first clause detects that a button called "(foo,1)" has been clicked, and completes the dialog by returning the structure "finished_at(H,M,S)" rather than the atom "hello". All other messages are passed to the default window handler. Once compiled, the new window handler is attached to the dialog window with a call to `window_handler/2`, as shown here:

```prolog
?- window_handler( foo, foo_handler ).
yes
```

If the dialog is invoked once again, and the "(foo,1)" button (labelled "Hello") is pressed, the time will be returned instead of the atom "hello":

```prolog
?- call_dialog( foo, X ).
X = finished_at(10,8,42)
```

Please note that "modal" and "modeless" dialogs have exactly the same window handlers: in fact, in WIN-PROLOG, these two types of dialog are one and the same thing as each other. All that determines whether a dialog behaves in a modal or modeless fashion at run-time is whether it is invoked by `call_dialog/2` (modal) or `show_dialog/1` (modeless). A third class of dialog, sometimes referred to as a "wizard", behaves semi-modally: like a modal dialog, it waits for input from the user when called with `wait_dialog/2`, but like a modeless dialog, it remains visible and in focus between successive calls, until finally hidden by invoking `hide_dialog/1`. 
**callable/1**

*test to see if a term is callable*

```prolog
callable(Term)
```

**Comments**

This predicate succeeds if the given `Term` is a Prolog atom, compound term, conjunction or disjunction; otherwise it fails.

**Examples**

The following calls test various cases:

```prolog
?- callable( 123 ).
no

?- callable( [1,2,3] ).
no

?- callable( foo ).
yes

?- callable( foo(bar) ).
yes

?- callable( X ).
no
```

**Notes**

This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

The `callable/1` predicate is simply a type test, and its success does not imply that it can be called without error. In particular, it is true of any compound term, even where the term is incomplete, or has a non-atom functor. For example, "X(A,B,C)" is "callable", but certainly cannot be called, at least until such time as "X" has been bound to an atom.

Please note that the special "empty list" atom, "[]", is not a true atom in **WIN-PROLOG**: for efficiency, this system uses a special data type to mark the ends of lists. It is therefore not possible to use "[]" to name predicates, and so `callable/1` fails if called with this atom.
**cat/3**

*join or split atoms or strings*

\[ cat( \text{Parts}, \text{Whole}, \text{Joins} ) \]

- **?Parts**  <list> or <variable>
- **?Whole**  <string> or <atom> or <variable>
- **?Joins**  <list> or <variable>

**Comments**  
This predicate can be used to join an arbitrary number of strings or atoms together, or to split an atom or string into an arbitrary number of parts.

When joining (concatenating) text items, **Parts** must be bound to a list containing atoms or strings (but not both), and **Whole** and **Joins** must be unbound variables. The predicate succeeds by returning a single atom or string in **Whole**, together with a list of integers in **Joins**, which describes where to split the resulting atom or string in order to restore the original list.

When splitting (separating) an atom or a string into a list of components, **Parts** must be an unbound variable, and **Whole** and **Joins** must be bound to an atom or string and a list of integers respectively. The predicate succeeds by returning the list of atoms or strings that is obtained by splitting the given atom or string at the specified split points.

**Examples**  
The following call concatenates a series of atoms, returning a single atom and list of split points:

\[- \text{cat( [the,quick,brown,fox], A, J ).} <\text{enter}>\]

\[ A = \text{thequickbrownfox} , \]
\[ J = [3,5,5] \]

The following call splits a string at the specified offsets:

\[- \text{cat( L, `jumpsoveralazydog`, [5,4,1,4] ).} <\text{enter}>\]

\[ L = [`jumps`,`over`,`a`,`lazy`,`dog`] \]

**Notes**  
The output text type generated by **cat/3** (ie, atom or string) is determined by the text type that was input. For this reason, and to avoid ambiguity, when concatenating text, the given list must contain either atoms or strings, but not both.
catch/2

call a goal and return its success, failure or error code

catch( Error, Goal )

?Error  <variable> or <integer>
+Goal   <goal>

Comments
This predicate calls the given Goal, and unifies Error with an integer error code, as shown in the following table:

<table>
<thead>
<tr>
<th>Error</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the goal succeeded</td>
</tr>
<tr>
<td>-1</td>
<td>the goal failed</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>the goal generated error number &quot;integer&quot;</td>
</tr>
</tbody>
</table>

Examples
The following call demonstrates the catching of the "Arithmetic Overflow Error" that is caused by dividing a number by zero in the is/2 predicate:

?- catch( Error, X is 1 / 0 ).
Error = 52 ,
X = _

The following shows that failure can be caught after all solutions to a call to member/2 have been exhausted:

?- catch( Error, member(X,[1,2,3]) ).
Error = 0 ,
X = 1 ;

Error = 0 ,
X = 2 ;

Error = 0 ,
X = 3 ;
Error = -1 ,
X = _

Notes

The catch/2 predicate always succeeds, whether its given Goal succeeds, fails or generates an error; however, please note that this behaviour means that an otherwise deterministic Goal will have at least one "extra" choicepoint upon success when called using catch/2. This can be shown easily:

?- catch( Error, true ).
Error = 0 ;

Error = -1

In order to ensure the Goal is properly deterministic, it is permissible to use "cut" (/0) in calls to catch/2:

?- catch( Error, (true,!)) .
Error = 0

When a goal fails or generates an error, the "extra" choicepoint is used up, and a cut is not necessary:

?- catch( Error, fail ).
Error = -1

The catch/2 predicate is very useful for calling programs which give rise to unpredictable errors, for example when trying to delete a file which may or may not be present at the time. Used in conjunction with throw/2, complete "catch-and-throw" error management can be achieved. Consider the program:

square_root( Arg1, Arg2 ) :-
    catch( Error, Arg2 is sqrt(Arg1) ),
    !,
    throw( Error, square_root(Arg1,Arg2) ).

Here, any error that occurs during the call to "Arg2 is sqrt(Arg1)" is "caught" by catch/2, and then "thrown" by throw/2, making it look as if any errors occurred directly inside square_root/2, rather than in is/2:

?- square_root( 123, X ).
X = 11.0905365064094
?- square_root( -123, X ).
! ----------------------------------------
! Error 53 : Arithmetic Error
! Goal : square_root(-123,_1)

Note the cut (/0) in this program, which removes the "extra" choicepoint left after a successful call to is/2.
catch/3

call a goal and return its success, failure or error code, and culprit

catch( Error, Goal, Culprit )

?Error <variable> or <integer>
+Goal <goal>
?Culprit <variable> or <term>

Comments This predicate calls the given Goal, and unifies Error with an integer error code, as shown in the following table, and unifies Culprit with the functor and arity of whatever predicates caused failure or an error:

<table>
<thead>
<tr>
<th>Error</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the goal succeeded, there is no culprit, so this argument is not bound</td>
</tr>
<tr>
<td>-1</td>
<td>the goal failed, so culprit is bound to the functor/arity of the predicate which failed</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>the goal generated error number &quot;integer&quot;, so culprit is bound to the functor/arity of the predicate which gave the error</td>
</tr>
</tbody>
</table>

Examples The following call demonstrates the catching of the "Arithmetic Overflow Error" that is caused by dividing a number by zero in the is/2 predicate:

?- catch( Error, X is 1 / 0, Culprit ).
Error = 52 ,
X = _ ,
Culprit = (is) / 2

Notice the parenthesising of "is" in the output from this example: this is a result of its being displayed as an argument to "="; which is an infix operator with the same precedence as "is" (both are "700,xfx").

Notes The catch/3 predicate always succeeds, whether its given Goal succeeds, fails or generates an error; however, please note that this behaviour means that an otherwise deterministic Goal will have at least one "extra" choicepoint upon success when called using catch/3: this is described in more detail in the "notes" section in the entry for catch/2.
char/1

test whether the given term is a character code

char( Term )

?Term <term>

Comments
This predicate succeeds if the given Term is an character code, which can be any 32-bit integer.

Examples
The following calls test various cases:

?- char( 123 ).
yes <enter>

?- char( 456 ).
yes <enter>

?- char( foo ).
no <enter>

?- char( `bar` ).
no <enter>

Notes
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

In version 4.100 and later of WIN-PROLOG, all text is represented in a compressed 32-bit encoding, which encompasses ASCII (7-bit characters in the range [00h..7Fh], ISO/IEC 8859-1 (8-bit characters in the range [00h..FFh], and Unicode (20-plus-bit characters in the range [000000h..10FFFFh]. The predicate char/1 is true of any integer in any of these ranges.
chars/1

test whether the given term is a list of character codes

chars( Term )

?Term <term>

Comments
This predicate succeeds if the given Term is a list comprised entirely of character codes, which can be any 32-bit integers; otherwise it fails.

Examples
The following calls test various cases:

?- chars( [1,2,3,4,5,999] ). <enter>
yes

?- chars( "foo" ). <enter>
yes

?- chars( 'bar' ). <enter>
no

?- chars( X ). <enter>
no

Notes
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

In version 4.100 and later of WIN-PROLOG, all text is represented in a compressed 32-bit encoding, which encompasses ASCII (7-bit characters in the range [00h..7Fh], ISO/IEC 8859-1 (8-bit characters in the range [00h..FFh], and Unicode (20-plus-bit characters in the range [000000h..10FFFFh]. The predicate chars/1 is true of any list made up entirely of integers in any of these ranges.
chdir/1

get or set the current working directory

chdir( Directory )

?Directory <atom> or <variable>

Comments

This predicate can be used to get or set the current working Directory: this is the directory relative to which WIN-PROLOG searches for a file when no absolute directory information is specified as part of the file name. When Directory is given as an atom, this forces the current working directory to change to whatever directory this atoms specifies; otherwise, when Directory is an unbound variable, it returns an atom which names the existing current working directory.

Examples

Supposing that "c:\pro386w" is the current working directory, this command picks it up:

?- chdir( D ).
D = 'C:\PRO386W'

In order to change to a new current working directory, say "c:\temp", the following command can be entered:

?- chdir( 'c:\temp' ).
yes

Notes

This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
get or check the body of a clause given its head

clause( Head, Body )

+Head       <atom> or <compound_term>
?Body       <goal> or <variable>

Comments
This predicate returns a clause whose head unifies with Head and whose body unifies with Body, and can backtrack to find alternative solutions. Where a clause has no body, a placeholder body of "true" is returned. If the predicate does not yet exist, it is declared as "dynamic", and consisting of no clauses.

Examples
Consider the following program:

:- dynamic likes/2.
likes( brian, prolog ).
likes( pooh, hunny ).
likes( brian, X ) :-
    likes( pooh, X ).

The following command picks up a clause whose head matches "likes(brian,X)", and returns further matches on backtracking:

?- clause( likes(brian,X), B ). <enter>
X = prolog ,
B = true ; <space>
X = _ ,
B = likes(pooh,X) ; <space>
no

Notes
When querying predicates, clause/2 requires that the target predicate's name and arity can be deduced from the given Head. This means,
minimally, that the head must be an atom or compound term whose principle functor is an atom. The arguments of any such compound term may be specified as unbound variables or anything else that will match the desired clauses; similarly, the Body may be a variable, or a partially or fully instantiated term. It is not possible to retrieve clauses "in reverse" by specifying the body and not the head.

Clauses can only be retrieved from predicates which are declared "dynamic": when clause/2 is called to query a previously nonexistent predicate, this declaration is performed automatically. However, if a predicate has been loaded from a file, the predicate may be considered "static", and any attempt to retrieve its clauses will result in errors. If a source file contains the definition of a predicate which is to be retrieved at run-time, this predicate should be declared dynamic explicitly with a call to dynamic/1; for example, consider the following file:

```prolog
:- dynamic foo/1.
  foo(hello).
  bar(world).
```

If this file were to be loaded with a call to consult/1, it would be possible to retrieve clauses from the foo/1 predicate, but not from bar/1.

It is never possible to retrieve clauses from optimised predicates (those which have been compiled with the optimising compiler); were the above file to be optimised, for example with a call to optimize_files/1, the foo/1 predicate would be bypassed by the optimiser, leaving it dynamic.
**clause/3**

_get or check the body and position of a clause given its head_

```prolog
clause( Head, Body, Position )
```

+Head <atom> or <compound_term>

?Body <goal>

?Position <integer> or <variable>

**Comments**

This predicate returns a clause whose head unifies with `Head`, whose body unifies with `Body`, and whose position within the relation unifies with `Position`, and can backtrack to find alternative solutions. Where a clause has no body, a placeholder body of "true" is returned. If the predicate does not yet exist, it is declared as "dynamic", and consisting of no clauses.

**Examples**

Consider the following program:

```prolog
:- dynamic likes/2.

likes( brian, prolog ).

likes( pooh, hunny ).

likes( brian, X ) :-
    likes( pooh, X ).
```

The following command picks up a clause whose head matches "likes(brian,X)"., and returns further matches on backtracking:

```prolog
?- clause( likes(brian,X), B, P ).
X = prolog ,
B = true ,
P = 1 ;

X = _ ,
B = likes(pooh,X) ,
P = 3 ;
```
Notes

This predicate is very similar to clause/2, the only difference being that it matches on head, body and position rather than just head and body: it is very useful for accessing specific clauses within a relation. Please see the "notes" entry for clause/2 for additional information about clause retrieval.
clauses/2

*return all clauses matching the given head*

```
clauses( Head, Clauses )
```
+Head <atom> or <compound_term>
?Clauses <list> or <variable>

**Comments**
This predicate returns a list of clauses whose heads unify with `Head`, unifying this with `Clauses`. It cannot backtrack to find alternative solutions, since it returns all possible matches at once. Where a clause has no body, a placeholder body of "true" is returned. If the predicate does not yet exist, it is declared as "dynamic", and consisting of no clauses.

**Examples**
Consider the following program:

```prolog
:- dynamic likes/2.

likes( brian, prolog ).

likes( pooh, hunny ).

likes( brian, X ) :-
    likes( pooh, X ).
```

The following command picks up a clause whose head matches "likes(brian,X)", and returns further matches on backtracking:

```
?- clauses( likes(brian,X), L ).
```

```
X = _ ,
L = [(likes(brian,prolog) :- true),(likes(brian,_1) :- likes(pooh,_1))]
```

**Notes**
Unlike `clause/2`, which matches clauses one at a time in response to backtracking, `clauses/2` returns all candidate clauses in a single, deterministic call.

Please see the "notes" entry for `clause/2` for additional information about clause retrieval.
close/1

close the named file

close( FileSpec )

+FileSpec <file_spec>

Comments
If FileSpec refers to the name of a currently open file, any unsaved information in its internal buffer is written to disk and the file is closed; if FileSpec is not the name of an open file, this predicate simply succeeds.

Examples
Assuming the file "prolog(data)" had previously been opened, using any of see/1, tell/1 or open/2, the following command will close it:

?- close( prolog(data) ).
eyes

Notes
When seen/0 is used to suspend input from a file previously opened with see/1, or told/0 is used to suspend output to a file previously opened with tell/1, the file concerned is not closed; this means that subsequent calls to see/1 or tell/1 continue input or output respectively at the point in the file where they previously left off. Calling close/1 after seen/0 or told/0 does close the file, so that subsequent calls to see/1 or tell/1 will once again commence from the top of the appropriate file.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used. This automatic expansion of file names can be shown by the following command:

?- open( foo, write ), fdict( 0, X ), close( foo ). <enter>
X = ['c:\pro386w\foo']

Both open/2 and close/1 take logical file names; here "foo" is automatically expanded to "c:\pro386w\foo", as shown in the value of "X" as returned by fdict/2.

Please see the "notes" section of absolute_file_name/2 for further discussion about file names.
**cmp/3**

*get or test the standard ordering of two terms numerically*

```prolog
cmp( Order, Term1, Term2 )
```

- ?Order <integer> in the range [-1..1] or <variable>
- +Term1 <term>
- +Term2 <term>

**Comments**
This predicate compares two terms, Term1 and Term2, according to the "standard ordering of terms", and unifies Order an integer to show their relationship, as described in the following table:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Term1 is less than Term2</td>
</tr>
<tr>
<td>0</td>
<td>Term1 is equal to Term2</td>
</tr>
<tr>
<td>1</td>
<td>Term1 is greater than Term2</td>
</tr>
</tbody>
</table>

**Examples**
The following command checks that "3" is less than "three" according to the standard ordering of terms:

```prolog
?- cmp( -1, 3, three ).
yes
```

The following command returns an integer that shows the term "foo(bar)" is greater than "bar(foo)":

```prolog
?- cmp( X, foo(bar), bar(foo) ).
X = 1
```

**Notes**
This predicate is very similar to `compare/3`: the main difference is that while the latter uses the atoms ",<", ",=" and ",>" respectively to describe "less than", "equal to" and "greater than", `cmp/3` uses the integers ",-1", ",0" and ",1".

The standard ordering of terms uniquely defines the relationship between any two terms: full details of standard ordering can be found in *Appendix A*, but here is a brief summary:

If two terms are of different types, comparison is simply defined by the relationship between these types (atoms are "less" than strings, etc).
If two terms have the same type, and are atomic (numbers, atoms, etc), comparison is based on their value. If two terms have the same type, but are compound (terms, lists, etc), comparison is based recursively first on their respective heads, and then on their tails.

Please note that the \texttt{cmp/3} predicate does not evaluate numerical expressions, but rather compares the actual structures representing these expressions.
compare/3

*get or test the standard ordering of two terms symbolically*

\[
\text{compare( Order, Term1, Term2 )}
\]

?Order <atom> or <variable>
+Term1 <term>
+Term2 <term>

**Comments**
This predicate compares two terms, Term1 and Term2, according to the "standard ordering of terms", and unifies Order to an atom to show their relationship, as described in the following table:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Term1 is less than Term2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Term1 is equal to or less than Term2</td>
</tr>
<tr>
<td>=</td>
<td>Term1 is equal to Term2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Term1 is greater than or equal to Term2</td>
</tr>
<tr>
<td>&gt;</td>
<td>Term1 is greater than Term2</td>
</tr>
</tbody>
</table>

**Examples**
The following command checks that "3" is equal to or less than "three" according to the standard ordering of terms:

?- compare( =<, 3, three ).
yes

The following command returns an atom that shows the term "foo(bar)" is greater than "bar(foo)":

?- compare( X, foo(bar), bar(foo) ).
X = >

**Notes**
When getting the relationship between two terms (Order is an unbound variable at the time of call), the result will always be one of the three atoms, "<", "=" or ">". The atoms "<=" and ">=" are never generated by compare/3, but can be used when testing a relationship (Order is an atom at the time of call).

This predicate is very similar to cmp/3: the main difference is that while the latter uses the integers "-1", "0" and "1" respectively to describe
"less than", "equal to" and "greater than", compare/3 uses the atoms "<", "=" and ">".

The choice of atoms in compare/3 is somewhat unfortunate, because they are used to name a series of predicates that evaluate arithmetic expressions and then compare their solutions (</2, >/2, etc). For example, consider these commands:

?- compare( <, 3, 2-1 ). yes
?- 3 < 2-1. no
?- 3 @< 2-1. yes

Clearly, for consistency with the other comparison predicates, compare/3 should have used the atoms "@<", "@=<", "@=", "@>=" and "@>" rather than "<", "=<", "=" and ">".

The standard ordering of terms uniquely defines the relationship between any two terms: full details of standard ordering can be found in Appendix A, but here is a brief summary:

If two terms are of different types, comparison is simply defined by the relationship between these types (atoms are "less" than strings, etc). If two terms have the same type, and are atomic (numbers, atoms, etc), comparison is based on their value. If two terms have the same type, but are compound (terms, lists, etc), comparison is based recursively first on their respective heads, and then on their tails.

Please note that the compare/3 predicate does not evaluate numerical expressions, but rather compares the actual structures representing these expressions.
**compile/1**

Load source files using the optimising compiler

```
compile( FileSpec )
```

+FileSpec <file_specs>

**Comments**
This predicate loads the programs contained in the disk file or files specified in FileSpec, using the optimising compiler. If the file has previously been loaded, the existing copy is abolished before reloading.

**Examples**
The following call loads the example program, "BENCHMRK.PL", into memory:

```
?- compile( examples(benchmrk) ).
```

# 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
yes

To see that it was compiled with the optimising compiler, type the call:

```
?- predicate_property( suite, P ).
P = optimized ;
P = static ;
```

no

**Notes**
This predicate is very similar to consult/1, the only difference being that the latter uses the incremental compiler instead of the optimising compiler. Code loaded with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed.

When predicates are loaded from a file, they retain an association with that file: if the file is reloaded, these predicates are automatically abolished. This provides a convenient method for "cleaning up" between successive runs of a system, especially where dynamic predicates have been defined in the source file. Any commands present in the source file are executed on the fly, as they are encountered by the loader.

If FileSpec does not specify a file extension, the "source" extension (normally '.PL') is assumed.
compound/1

test whether a term is a compound term

compound( Term )

?Term <term>

Comments
This succeeds if Term is compound, which means that it is currently bound to a tuple, list, conjunction or disjunction.

Examples
The following calls test various cases:

?- compound( 123 ). no <enter>

?- compound( [1,2,3] ). yes <enter>

?- compound( foo ). no <enter>

?- compound( bar(sux) ). yes <enter>

?- compound( X ). no <enter>

Notes
The term "compound" means non-variable and non-atomic, implying that the term is "divisible", without the use of the special term conversion predicates that are required to split atomic data types.
constant/1

test whether the given term is a true atom

constant( Term )

?Term <term>

Comments
This predicate succeeds if the given Term is a true atom; otherwise it fails.

Examples
The following calls test various cases:

?- constant( 123 ). <enter>
  no

?- constant( [1,2,3] ). <enter>
  no

?- constant( [] ). <enter>
  no

?- constant( foo ). <enter>
  yes

?- constant( `bar` ). <enter>
  no

?- constant( X ). <enter>
  no

Notes
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

Please note that the special "empty list" atom, "[]", is not a true atom in WIN-PROLOG: for efficiency, this system uses a special data type to mark the ends of lists. It is therefore not possible to use "[]" to name predicates or files, and it should be noted that the following call fails:
?- [] = '[]'.
no

The `atom/1` predicate treats the empty list as an atom purely for compatibility with Quintus Prolog: in all other respects, this item should be thought of as a list with no elements.
consult/1

load source files using the incremental compiler

    consult( FileSpec )

    +FileSpec  <file_specs>

Comments
This predicate loads the programs contained in the disk file or files specified in FileSpec, using the incremental compiler. If the file has previously been loaded, the existing copy is abolished before reloading.

Examples
The following call loads the example program, "BENCHMRK.PL", into memory:

    ?- consult( examples(benchmrk) ).
    # 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
    yes

To see that it was compiled with the incremental compiler, type the call:

    ?- predicate_property( suite, P ).
    P = compiled ;
    P = static ;
    no

Notes
This predicate is very similar to compile/1, the only difference being that the latter uses the optimising compiler instead of the incremental compiler. Code loaded with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed.

When predicates are loaded from a file, they retain an association with that file: if the file is reloaded, these predicates are automatically abolished. This provides a convenient method for "cleaning up" between successive runs of a system, especially where dynamic predicates have been defined in the source file. Any commands present in the source file are executed on the fly, as they are encountered by the loader.

If FileSpec does not specify a file extension, the "source" extension (normally '.PL') is assumed.
copy/2

copy data from current input to current output

\text{copy}( \text{Wanted}, \text{Copied} )

\begin{align*}
+\text{Wanted} & \quad \langle \text{integer} \rangle \\
-\text{Copied} & \quad \langle \text{variable} \rangle
\end{align*}

Comments
This copies up to the number of characters \text{Wanted}, from the current input stream to the current output stream, and returns the number of characters successfully \text{Copied}.

Examples
The following command copies 5 characters out of one string, and place them in another:

\begin{verbatim}
?- \text{copy}(\text{5},\text{C}) \leftarrow `Hello World` \rightarrow \text{S}.
\end{verbatim}

\begin{align*}
\text{C} = 5, \\
\text{S} = `Hello`
\end{align*}

Here is a simple program that will make a complete copy of an existing file, and return its size:

\begin{verbatim}
\text{copy_file}( \text{Source}, \text{Target}, \text{Size} ) :-
  \text{fcreate}(\text{i},\text{Source},0,0,0),
  \text{fcreate}(\text{o},\text{Target},-2,0,0),
  \text{input}(\text{I}),
  \text{output}(\text{O}),
  \text{input}(\text{i}),
  \text{output}(\text{o}),
  \text{copy}(\text{-1},\text{Size}),
  \text{input}(\text{I}),
  \text{output}(\text{O}),
  \text{fclose}(\text{i}),
  \text{fclose}(\text{o}).
\end{verbatim}

After compiling this program, it can be used in a call such as:

\begin{verbatim}
?- \text{copy_file}(\text{examples\benchmrk.pl},\text{temp},\text{D}).
\end{verbatim}
D = 18142

Notes

Internally, copy/2 uses unsigned integer arithmetic to count characters, so "-1" is a convenient way of specifying the copying of files up to just under 4Gb in size. Because this predicate copies directly from input to output, without using Prolog’s internal data structures, it can handle large amounts of data very efficiently.
copy_term/2

*make a new copy of a term*

```
copy_term( Term, Copy )
```

+Term   <term>
?Copy   <variable> or <term>

**Comments**
This predicate makes a duplicate of the given Term, and then unifies this with Copy. All variables in the original Term are replaced with new ones in the Copy.

**Examples**
The following call makes a copy of a term, before binding variables in the original:

```
?- copy_term( foo(A,B,A), C ), A = a, B = b.
```

A = a,
B = b,
C = foo(_1,_2,_1)

**Notes**
The ability to copy a term allows some fairly sophisticated programs to be written. Consider the problem of implementing a "for" loop in Prolog, so that, for example, the call:

```
?- for( X, 1, 9, write(X) ).
```

would result in the output, "123456789". A naive (and non-working) attempt at a solution would be to write:

```
for( _, Count, Limit, _ ) :-
    Count > Limit,
    !.

for( X, Count, Limit, Goal ) :-
    X = Count,
    Goal,
    Next is Count + 1,
    for( X, Next, Limit, Goal ).
```
The problem with this program is that at the first iteration, "X" and "Count" are bound, which prevents further iterations from being able to unify "X" with successive counts:

\[
\text{?- for( X, 1, 9, write(X) ), nl.} \quad <\text{enter>}
\]

1no

By duplicating the calls "X = Count, Goal" into a new term, "NewGoal", which has completely new variables, and calling this instead, it is possible to overcome this problem:

\[
\text{for( _, Count, Limit, _ ) :-}
\]
\[
\text{Count > Limit, !.}
\]
\[
\text{for( X, Count, Limit, Goal ) :-}
\]
\[
\text{copy_term( (X = Count,Goal), NewGoal ), NewGoal,}
\]
\[
\text{Next is Count + 1, for( X, Next, Limit, Goal ).}
\]

Once compiled, this "for" loop works as intended:

\[
\text{?- for( X, 1, 9, write(X) ), nl.} \quad <\text{enter>}
\]

123456789
X = _

Term duplication is generally useful in any case where it is desired to call a goal without binding its variables.
perform a cyclic redundancy check on data from current input

\[
\text{crc} \begin{array}{ll}
\text{Wanted} & \text{<integer>} \\
\text{Checked} & \text{<variable>} \\
\text{Result} & \text{<variable>}
\end{array}
\]

**Comments**
This checks up to the number of characters *Wanted*, from the current 8-bit input stream, computes the 32-bit cyclic redundancy check (CRC32) on the data, and returns the number of characters successfully *Checked* as well as the *Result* CRC32 value as a 32-bit integer.

**Examples**
The following command checks all characters in a string, returning its length and CRC32:

\[
?\text{-} \text{crc} \begin{array}{l}
\text{\(-1, Count, CRC\)} \end{array} \text{<~ `Hello World`.}
\]

\[
\begin{array}{l}
\text{Count = 11 ,} \\
\text{CRC = 1243066710}
\end{array}
\]

**Notes**
An input value of "-1" interpreted is interpreted as "process the entire file", using a 53-bit (X86) or 64-bit (X64) counter. Because this predicate digests directly from input, without using Prolog's internal data structures, it can handle large amounts of data very efficiently.

There are three data checking predicates in **WIN-PROLOG**, providing differing levels of security. The first is *crc/3*, which performs the 32-bit "Cyclic Redundancy Check", or "CRC32", a reliable checksum a given data stream. The other two predicates, *mdf/3* and *sha/3*, perform the 128-bit "Message Digest Five", or "MD5", and 256-bit "Secure Hash Algorithm", or "SHA-256" message digests respectively.

While CRC32 is fine for quick checks, it is not "secure": there is a \(2^{-32}\) chance of two data streams having the same CRC32, and it is easy to create files with a given CRC32 value. Meanwhile, MD5 is relatively secure, with only a \(2^{-128}\) chance of a clash: it is also not thought possible to design a file with a specific MD5, although recent attacks are making it look less than ideal. For maximum security, use SHA-256, which has only a \(2^{-256}\) chance of a collision, and is considered computationally impossible to crack.

Note that *crc/3* computes the CRC-32 cyclic redundancy check as a single 32-bit integer: normally, such checksums are represented as an 8-character hexadecimal string, comprising the big-endian hexadecimal representation of this integers. Consider the following program:

\[
\text{crc( File ) :-}
\]

\[
\text{fcreate( input, File, 0, 0, 0 ),}
\]
input( Current ),
input( input ),
crc( -1, _, Int ),
input( Current ),
fclose( input ),
fwrite( r, 8, 16, Int ),
nl.

This takes as its input the name of a single file, which is opened in "ISO" (8-bit, ISO/IEC 8859-1) mode, and then checked using \texttt{crc/3}; the integer in the resulting checksum is then written to the current output stream in big-endian hexadecimal format by \texttt{fwrite/4}. Suppose we had a text file, "HELLO.TXT", containing just the characters, "hello world—M—J"; once the above program is compiled, the following call will display the conventional CRC-32 signature of the file:

\begin{verbatim}
?- crc( 'hello.txt' ).
38E6C41A
yes
\end{verbatim}

Note that both CRC-32 and SHA-256 values are traditionally output in big-endian format, making direct use of \texttt{fwrite/4} possible when displaying the results of \texttt{crc/3} and \texttt{sha/3} respectively, but MD5 values are conventionally output in little-endian format, requiring some "bit twiddling" to display the output of \texttt{mdf/3} correctly.
current_atom/1

test whether the given term is an atom, or return atoms

    current_atom( Term )

    ?Term   <term>

Comments  This predicate succeeds if the given Term is a Prolog atom or can be bound to an atom; otherwise it fails.

Examples  The following calls test various cases:

    ?- current_atom( 123 ).
    no

    ?- current_atom( [1,2,3] ).
    no

    ?- current_atom( foo ).
    yes

    ?- current_atom( `bar` ).
    no

    ?- current_atom( X ).
    X = (~>) ;
    X = (<=) ;
    X = >> ;

Notes  This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

    The current_atom/1 predicate is similar to atom/1: when called with a non-variable argument, it simply succeeds or fails depending upon whether or not that argument is an atom. Unlike other type tests, however, current_atom/1 succeeds when called with a variable argu-
ment, returning successive known atoms on backtracking.

Please see the "Notes" section in the entry for `atom/1` for information about the special "empty list" atom, `[]`, which, for efficiency reasons, is not a true atom in **WIN-PROLOG**.
current_op/3

check or get the definition of a current operator

current_op( Precedence, Type, Name )

?Precedence       <integer> or <variable>
?Type             <atom> or <variable>
?Name             <atom> or <variable>

Comments This predicate can be used to test or retrieve information about the Precedence, Type and/or Name of a current operator, and will return successive solutions on backtracking.

Examples The following call returns definitions of the "+" operator:

?- current_op( P, T, + ).
    P = 500 ,
    T = fx ;

    P = 500 ,
    T = yfx ;

    no

Notes Operators are critical components of Prolog syntax, and define how terms are assembled internally during calls to read/1, and how they are output by write/1, writeq/1 and so on. In expressions with mixed operators, those with lower precedences bind more tightly than those with higher ones.

For example, the "+" operator has an infix precedence of 500, as shown above; the precedence of "*" is lower, at 400. With this relationship, the term "(1 * 2 + 3)" is translated to "+((1,2),3)" during input. Suppose, however, that the following call was made to the op/3 predicate, to increase the infix precedence of "*" to 600:

?- op( 600, yfx, * ).
    yes

Because this call has given "*" a higher precedence than "+", the term "(1 * 2 + 3)" will now be translated to "*(1,+(2,3))" during input.
current_predicate/1

check or get a current user predicate

\[ \text{current_predicate( Pred )} \]

?Pred \hspace{1cm} <\text{pred_spec}> \text{ or } <\text{variable}> 

Comments
This predicate can be used to test or retrieve information about user-defined predicates. If Pred is fully bound to a predicate specification, then the call will succeed if that predicate exists; otherwise, if any part of Pred is an unbound variable, details about successive user predicates are returned on backtracking.

Examples
The following calls create a single clause for each of a pair of dynamic predicates, \textit{foo/2} and \textit{bar/1}, and then return these predicates successively on backtracking:

\[ \text{?- assert( foo(hello,there) ), assert( bar(world) ).} \] <enter>

yes

\[ \text{?- current_predicate(P).} \] <enter>

P = foo / 2 ;

P = bar / 1 ;

no

Notes
The \texttt{current_predicate/1} predicate is only true of user predicates, or of those loaded from files at run-time: it does not return or succeed for system or built-in predicates. However, please note that public predicates contained within plug-in modules, such as the cross referencer (XREFS.PC) and demand-loaded modules, such as the source level debugger (SRCBUG.PC), will be true of \texttt{current_predicate/1}.

This predicate is closely related to \texttt{current_predicate/2}, and simply provides an alternative notation for specifying user-defined predicates.
**current_predicate/2**

*check or get a current user predicate and call template*

```prolog
current_predicate( Name, Term )
```

?Name <atom> or <variable>
?Term <term> or <variable>

**Comments**

This predicate can be used to test or retrieve information about user-defined predicates. If `Name` is bound to the name of a user predicate, then `Term` will be unified with the most general representation of a call to any definition of that predicate; otherwise, if `Name` is an unbound variable, details about successive user predicates are returned on backtracking.

**Examples**

The following calls create a single clause for each of a pair of dynamic predicates, `foo/2` and `bar/1`, and then return these predicates successively on backtracking (please note that the returned arguments are arbitrary dummy variables):

```prolog
?- assert( foo(hello,there) ), assert( bar(world) ).
```

yes

```prolog
?- current_predicate(N,C).
```

```prolog
N = foo ,
C = foo(_1,_2) ;
```

```prolog
N = bar ,
C = bar(_3) ;
```

no

**Notes**

The `current_predicate/2` predicate is only true of user predicates, or of those loaded from files at run-time: it does not return or succeed for system or built-in predicates. However, please note that public predicates contained within plug-in modules, such as the cross referencer (XREFS.PC) and demand-loaded modules, such as the source level debugger (SRCBUG.PC), will be true of `current_predicate/2`.

This predicate is closely related to `current_predicate/1`, and simply provides an alternative notation for specifying user-defined predicates.
**debug/0**

set the debugging mode to "debug"

```
debug
```

**Comments**

This predicate sets the debugging mode to "debug", which will result in the debugger being invoked if a predicate is called for which a spypoint is currently set.

**Examples**

The following call sets a spypoint on a predicate "foo/1", and then sets the debugging mode to "debug" so that the debugger will take over if and when foo/1 is called:

```
?- spy( foo/1 ).
# Spypoint placed on predicate foo / 1
yes

?- debug.
yes
```

**Notes**

The debugging mode has three settings: "off", "debug", and "trace". In the first case, the debugger is never invoked, while in the second, it is invoked if a spied predicate is called. In the third case, the debugger is invoked immediately a query is entered at the console prompt.

The three debugging modes are mutually exclusive, but are somewhat confusingly set by four predicates, as shown in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>nodebug/0 or notrace/0</td>
</tr>
<tr>
<td>debug</td>
<td>debug/0</td>
</tr>
<tr>
<td>trace</td>
<td>trace/0</td>
</tr>
</tbody>
</table>

The `nodebug/0` and `notrace/0` predicates are effectively identical, and either can be used to cancel the effects of either `debug/0` or `trace/0`. Please note that WIN-PROLOG's default debugging mode is "debug".
'?DEBUG?'/1

user-defined hook to gain control at a spypoint

'?DEBUG?'( Goal )

?Goal <goal> or <variable>

Comments
This is not a built-in predicate, but an optional hook that can be written by the user to gain control after a program has called a predicate for which a spypoint is currently set, and when the debugging mode is one of "debug" or "trace".

Examples
The following program provides a simple debugger which displays calls for any spied predicates which are called while the debugging mode is set to one of "debug" or "trace":

'?DEBUG?'( Goal ) :-
    output( Current ),
    output( 0 ),
    write( 'Calling: ' ),
    writeq( Goal ),
    write( '~M~J~Z' ),
    output( Current ),
    force( Goal ).

Notes
The '?DEBUG?'/1 hook predicate is intended for use only during program development, although unlike other hooks, its mechanism is still active within stand-alone applications.

In order to maintain program integrity, it is essential that the given Goal is executed: however, please note that because '?DEBUG?'/1 is only invoked in the first place because a spypoint was set on the predicate which Goal tried to call, simply calling Goal directly will fire another spypoint, resulting in an endless loop. One solution would be set the debugging mode to "off" while the call is made, but this will prevent any other spypoints from firing during the call. Instead, as shown above, the Goal should be called using the force/1 predicate, which behaves just like call/1, but disables spypoints just long enough to set up the call to the Goal.

In the absence of a user-supplied definition of '?DEBUG?'/1, the WIN-PROLOG system will call the built-in predicate, debug_hook/1. This normally invokes the currently selected debugger to handle the goal being called.
**debug_hook/1**

*default debug hook*

```prolog
debug_hook( Goal )
+Goal <goal>
```

**Comments**
This predicate is called by the **WIN-PROLOG** system to invoke the currently selected debugger to handle the given `Goal`. It is normally only called directly from a user-defined debug hook (see `?DEBUG?/1`).

**Examples**
This predicate can be called directly to debug any goal, irrespective of the current debugging mode. For example, the following call will enter the debugger to trace the execution of `file_search_path/2`, ultimately displaying its results as normal:

```
?- debug_hook( file_search_path(Name, Path) ). <enter>
Name = system ,
Path = prolog('system\') ; <space>
Name = examples ,
Path = prolog('examples\') ; <space>
Name = library ,
Path = prolog('library\') ; <space>
Name = temp ,
Path = prolog(\.)
```

**Notes**
The `debug_hook/1` predicate is normally only used during program development, while the development environment is active. While it can be called stand-alone applications, it is not normal for such systems to include Prolog-level debugging facilities. Instead, applications should perform diagnostics using simple calls to predicates such as `write/1`, `msgbox/4` and other output or display predicates.


**debugging/**0  

`write debugger status to the current output stream`

**debugging**

**Comments**  
This predicate displays information about the **debugging mode**, **leashed ports** and active spypoints, writing a summary to the current output stream.

**Examples**  
Suppose that a predicate, `foo/1`, exists and has been "spied" by a call to `spy/1`. With the debugging mode set to "debug" and all standard ports "leashed", the following call will generate the response shown here:

```
?- debugging.  
[State : debug]  
[Head  : call exit redo fail ]  
[Body  : call exit redo fail ]  
[Spied : foo/1 ]
```

**Notes**  
The debugging mode is referred to as "State" by `debugging/0`. It is changed by any of the predicates, `debug/0`, `trace/0`, `nodebug/0` and `notrace/0`; the latter two predicates are effectively identical. The leashed ports can be set by `leash/2`, and checked or obtained by `leashed/2`. Spypoints are set by `spy/1`, and cleared with calls to `nospy/1` or `nospyall/0`. 
**decode/2**

*decode data from current input to current output*

```
decode( Password, Raw )
```

<table>
<thead>
<tr>
<th>+Password</th>
<th>&lt;string&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Raw</td>
<td>&lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**

This copies data from the current input stream to the current output stream, decoding (decrypting) it along the way, using the given `Password`. It terminates when the end of an encrypted input record is encountered. The raw character count is returned in `Raw`.

**Examples**

The following simple programs can be used to create or unpack encrypted archives of multiple files, using a secret password:

```
encode_archive( Password, Archive, Files ) :-
  open( Archive, write ),
  tell( Archive ),
  forall( member( File, Files ),
    (  writeq( File ),
      write( '.' ),
      see( File ),
      encode( -1, Password, _ ),
      seen
    )
  ),
  told.

decode_archive( Password, Archive ) :-
  see( Archive ),
  repeat,
  read( File ),
  tell( File ),
  decode( Password, _ ),
  told,
  at_end_of_file,
!,
Once compiled, this program can be used to create an encoded (encrypted) archive of multiple files; the following call, for example, will create an archive called "foo" which includes encrypted copies of two other files, "bar" and "sux", using the secret password, "abc":

```
?- encode_archive( `abc`, foo, [bar, sux] ).
yes
```

These files can be recovered from the archive on a subsequent occasion, provided the password is known, with the call:

```
?- decode_archive( `abc`, foo ).
yes
```

**Notes**

Because `decode/2` reads a single stuffed record, rather than an entire file, more than one encrypted sequence can be encoded into a single file. The MZSS algorithm, as used to encrypt and decrypt data in the `encode/3` and `decode/2` predicates respectively, is a highly-secure 1185-bit single key encryption system. It uses a password string of up to 148 characters is to seed a sophisticated "Marsaglia Zaman" pseudo random number generator ("PRANG"), which is completely independent from the "linear congruential" PRANG used by `is/2` and other arithmethic predicates. A comb-filtering process uses only randomly selected numbers from the Marsaglia Zaman data stream, avoiding any chance of using a knowledge of recent history to try to predict future numbers.

One common attack on an encrypted document is to use a second document for which both "plaintext" and "cyphertext" versions are available, and which is thought to have been encoded with the same password as the document under attack. MZSS is completely immune to this, because a completely different random sequence is guaranteed to be generated every time, even for a given password. The current system date and time are combined with the seed generated from the user's password, and "published" in the cyphertext file to enable subsequent decryption to use the same "sequence variation". See Appendix K for more information about MZSS encryption.
decompile/3

*decompile* an incrementally compiled predicate

```
decompile( Name, Arity, Code )
```

+Name <atom>
+Arity <integer>
?Code <variable> or <list>

**Comments**

This "decompiles" an entire incrementally compiled predicate of the given *Name* and *Arity*, binding a list containing its clauses and ancillary information to *Code*. Each entry in the *Code* list is of the form:

```
clause( Posn, Vars, Head, Body )
```

where *Posn* is an integer identifying the clause position, *Vars* is a list of (Name,Var) pairs that links any variables to atoms containing their original names, *Head* is the clause head and *Body* is its body. The latter element is simply bound to the atom "true" when a given clause is a fact.

**Examples**

Consider the following simple program:

```
foo( a ).

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

Once loaded with consult/1 or otherwise compiled by the "incremental compiler", its entire definition can be retrieved by the following call:

```
?- decompile( foo, 1, C ).
```

```
C = [clause(1,[],foo(a),true),clause(2,['X',_1],foo(_1),write(_1))]
```

**Notes**

Unlike the clause/2 predicate, decompile/3 can retrieve the code of any incrementally compiled predicate, and not just those that have been declared "dynamic" (see dynamic/1). In addition, the present predicate returns a "varlist" of each clause, giving the names that were assigned to each of its variables at the time the code was compiled. This enables program listings and debugger output to contain original variable names when required (see ewrite/2 and eprint/2).
def/3

test existence of a predicate and check or return its type

\[ \text{def( Name, Arity, Type )} \]

+Name <functor>
+Arity <arity>
?Type <variable> or <integer> in the range [0..5]

Comments

This succeeds if a predicate of the given \text{Name} and \text{Arity} exists, and its type matches \text{Type}, as listed in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No definition, but one or more predicate flags</td>
</tr>
<tr>
<td>1</td>
<td>Written in Prolog, and incrementally compiled</td>
</tr>
<tr>
<td>2</td>
<td>Written in Prolog, and hashed</td>
</tr>
<tr>
<td>3</td>
<td>Written in Prolog, and optimised</td>
</tr>
<tr>
<td>4</td>
<td>Written in Assembler, and built-in</td>
</tr>
<tr>
<td>5</td>
<td>Written in Assembler, and dynamically loaded</td>
</tr>
</tbody>
</table>

Examples

The following call adds a single clause to create the incrementally compiled predicate, \text{foo/1}, and adds the dynamic predicate flag to an otherwise undefined predicate, \text{bar/1}:

\[ ?- \text{assert( foo(_) ), dynamic( bar/1 ).} \]

yes

Given these definitions, and assuming there are no other user predicates, the following calls return various predicate types:

\[ ?- \text{def( exit, 1, T ).} \]
\[ T = 3 \]

\[ ?- \text{def( halt, 1, T ).} \]
\[ T = 2 \]
?- def( foo, 1, T ).
    T = 1  <enter>

?- def( bar, 1, T ).
    T = 0  <enter>

The following call fails, because there is no definition of any kind for sux/0:

?- def( sux, 1, T ).
   no  <enter>

Notes

The `def/3` predicate can be used as a "guard" prior to calling a predicate which may not exist, and therefore which would otherwise generate an error; it can also be used to determine which predicates can be processed using features which are limited to working on certain predicate types. For example, it is not possible to generate a program listing for an optimised predicate or one which is assembler-coded, so the following definition of `my_listing/1` might be useful:

```
my_listing( Name / Arity ) :-
    (  def( Name, Arity, 0 )
        -> write( 'No clauses for ' ),
              writeq( Name / Arity ),
              nl
        ;  (  def( Name, Arity, 1 )
              ;  def( Name, Arity, 2 )
        )
        -> listing( Name / Arity )
    ;  def( Name, Arity, Type )
        -> write( 'Cannot list predicates of type ' ),
              write( Type ),
              nl
        ;  writeq( Name / Arity ),
              write( ' does not exist' ),
              nl
    ).
```

Predicate types are also used in `pdict/4`, which returns lists of predicates matching a given type and flag mask; these flags are set by various predicates, such as `spy/1` and `dynamic/1`. A list of arities for a given predicate name can be retrieved with the `defs/2` predicate.
**defs/2**

return all arities for a given predicate name

```
defs( Name, Arities )
```

+Name               <functor>
-Arities            <variable>

Comments  This returns a list of Arities for any predicates of the given Name. If no predicates exist with the given Name, Arities is bound to an empty list.

Examples  The following call shows that there are two predicates that share the name "sort":

```
?- defs( sort, A ).
A = [2,3]
```

Notes  Together with the def/3 predicate, defs/2 is useful in programs which need to find out about defined programs; pdict/4 can also be used to return lists of predicates as (name,arity) pairs.
del/1

delete a file

\[ \text{del( File )} \]

+File <atom>

Comments

This deletes the named File from the disk. If the named file does not exist, an error is generated.

Examples

Assuming that there is a file called "foo" in the current working directory, the following call deletes it:

\[ ?- \text{del( foo ).} \]

yes

Notes

This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.

Please note that del/1 attempts to delete files immediately, but cannot check whether the file concerned is currently open for input and/or output. Depending upon the version of Windows being run, it might be possible to delete a file which is still active: in such cases, disk space will need to be recovered subsequently by calls to the CHKDSK or SCANDISK utilities.
**dict/2**

return a list of all visible and/or hidden atoms

\[
\text{dict( Flag, Atoms )}
\]

+Flag <integer> in the domain \{-1,0,1\}
-Atoms <variable>

**Comments**
This predicate returns an unordered list comprised of true Atoms currently known to **WIN-PROLOG**, as specified in the given Flag, as listed in the following table:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Return only visible atoms</td>
</tr>
<tr>
<td>1</td>
<td>Return only hidden atoms</td>
</tr>
<tr>
<td>-1</td>
<td>Return all atoms, both visible and hidden</td>
</tr>
</tbody>
</table>

**Examples**
The following call returns the entire collection of visible true atoms as a list, as specified by the given flag together with its length (please note that this list has been abbreviated in the example below):

```
?- dict( 0, D ), len( D, L ).
D = [~>,<~, .. ,wm_nclbuttonup,=] ,
L = 1499
```

**Notes**
Please note that the special "empty list" atom, "[]", is not a true atom in **WIN-PROLOG**: for efficiency, this system uses a special data type to mark the ends of lists. It is therefore not possible to use "[]" to name predicates or files, and it should be noted that the following call fails:

```
?- [] = '[]'.
no
```

The dict/2 predicate only returns true atoms, and so does not include the empty list among its members.

It should be noted that hidden atoms have a number of potentially confusing properties. For example, they cannot be unified with any atom newly created during input, even if they have the same spelling; similarly, it is quite possible for two or more identically spelt atoms
to occur on the hidden dictionary, while the main dictionary guarantees that any atom will only ever appear once. The `hide/2` predicate can be used to transfer atoms between the visible and hidden portions of the dictionary. Please treat this predicate with caution, and refer to its entry for further information.
get a matching list of files from a single directory

\[ \text{dir( Pattern, Mask, Files )} \]

+Pattern <atom>
+Mask <integer>
-Files <variable>

Comments    This predicate returns the list of atoms naming all those files from a single directory that match a given Pattern and Mask. The Mask encodes those file attributes to include or exclude in a search, by adding together the values in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000001</td>
<td>include read-only files</td>
</tr>
<tr>
<td>16'00000002</td>
<td>include hidden files</td>
</tr>
<tr>
<td>16'00000004</td>
<td>include system files</td>
</tr>
<tr>
<td>16'00000010</td>
<td>include directories</td>
</tr>
<tr>
<td>16'00000020</td>
<td>include archive files</td>
</tr>
<tr>
<td>16'00010000</td>
<td>exclude read-only files</td>
</tr>
<tr>
<td>16'00020000</td>
<td>exclude hidden files</td>
</tr>
<tr>
<td>16'00040000</td>
<td>exclude system files</td>
</tr>
<tr>
<td>16'00100000</td>
<td>exclude directories</td>
</tr>
<tr>
<td>16'00200000</td>
<td>exclude archive files</td>
</tr>
</tbody>
</table>

Examples    The following call returns a list of all hidden files in the root of the "C:" drive, excluding directories and system files:

\[ \text{?- dir( 'c:/*.\*', 16'00140002, F ).} \]

F = ['c:\COMMAND.COM', 'c:\AUTOEXEC.BAT', 'c:\CONFIG.SYS']

Notes    This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms; there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
The attribute mask used by `dir/3` and `dir/4` is a 32-bit integer that really comprises two separate 16-bit masks. The low-order 16 bits define any DOS/Win32s file attributes which must be present in order for a file to qualify for inclusion; the high-order 16 bits define any such attributes which must not be present in order for a file to qualify. The more bits that are set, the more specific will be the file selection; conversely, if no bits are set, all files will qualify. In order to see all files matching the given Pattern, simply specify a Mask of zero ("0").

An error will be raised if any given bits are set both in the low and high halves of the mask, because it is not possible both to include and exclude files based on the same attribute.

The `dir/3` and `dir/4` predicates return lists of file names, but no information (date/time stamps, size, etc.) about the files: to retrieve such information about any one file, use the `file/3` predicate.
get a matching list of files from a nested tree of directories

\[
dir( \text{Root}, \text{Pattern}, \text{Mask}, \text{Files} )
\]

+Root <atom>
+Pattern <atom>
+Mask <integer>
-Files <variable>

Comments
This predicate returns the list of atoms naming all those files in a recursive, nested tree of directories, starting at the given Root directory, that match a given Pattern and Mask. The Mask encodes those file attributes to include or exclude in a search, by adding together the values in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000001</td>
<td>include read-only files</td>
</tr>
<tr>
<td>16'00000002</td>
<td>include hidden files</td>
</tr>
<tr>
<td>16'00000004</td>
<td>include system files</td>
</tr>
<tr>
<td>16'00000010</td>
<td>include directories</td>
</tr>
<tr>
<td>16'00000020</td>
<td>include archive files</td>
</tr>
<tr>
<td>16'00010000</td>
<td>exclude read-only files</td>
</tr>
<tr>
<td>16'00020000</td>
<td>exclude hidden files</td>
</tr>
<tr>
<td>16'00040000</td>
<td>exclude system files</td>
</tr>
<tr>
<td>16'00100000</td>
<td>exclude directories</td>
</tr>
<tr>
<td>16'00200000</td>
<td>exclude archive files</td>
</tr>
</tbody>
</table>

Examples
The following call returns a list of all hidden files in the root of the "C:" drive, excluding directories:

?- F = ['c:\PRO386\PRO386W.BMP','c:\PRO386\PRO386W.EXE','c:\PRO386\PRO386W.OVL']

Notes
Unlike most other operating system predicates, \textit{dir/4} is not implemented directly in the \textit{WIN-PROLOG "kernel"}, a low-level, assembler-coded part of the system; however, it does use the same filename conventions as those predicates that are implemented in this way,
such as dir/3. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.

The attribute mask used by dir/3 and dir/4 is a 32-bit integer that really comprises two separate 16-bit masks. The low-order 16 bits define any DOS/Win32s file attributes which must be present in order for a file to qualify for inclusion; the high-order 16 bits define any such attributes which must not be present in order for a file to qualify. The more bits that are set, the more specific will be the file selection; conversely, if no bits are set, all files will qualify. In order to see all files matching the given Pattern, simply specify a Mask of zero ("0").

An error will be raised if any given bits are set both in the low and high halves of the mask, because it is not possible both to include and exclude files based on the same attribute.

The dir/3 and dir/4 predicates return lists of file names, but no information (date/time stamps, size, etc.) about the files: to retrieve such information about any one file, use the file/3 predicate.

Please note that in recent versions of WIN-PROLOG prior to 5.000, the Root argument was a list of atoms, rather than a single atom, to enable multiple independent directories to be listed simultaneously; this feature proved to be problematic in some instances, and it was felt preferable to revert this predicate to its former state, that existed prior to WIN-PROLOG version 4.800.
**display/1**

*write a term to the user output stream in prefix notation*

```
display( Term )
```

**Comments**
This writes the given `Term` in quoted, prefix (operator-free) notation, directly to the *user* output stream.

**Examples**
The following call demonstrates that terms are output in prefix notation, ignoring all operator declarations:

```
?- display( X is 2 + 2 ), nl.

is(_1,+(2,2))
X = _
```

This call shows both that terms are output quoted, and that output is always forced to the user output stream:

```
?- display( 'Atom'(`String`) ) ~> S, nl.

'Atom'(`String``)
S = ``
```

**Notes**
One of the family of term output predicates, `display/1` is a particularly quirky relic from Quintus Prolog, included here as part of the compatibility legacy. For a start, it is the only term output predicate which ignores the setting of the current output stream, although this can be seen as useful for debugging purposes. Similarly useful is its avoidance of operators, because if these are redefined in an application, debugger output can become hard to interpret.

In versions of *WIN-PROLOG* prior to 5.000, `display/1` used unquoted output, making the predicate pretty much useless for its purported purpose, namely displaying terms in an unambiguous canonical form. For example, in *WIN-PROLOG* 4.920, the second example above would have behaved like this, making it impossible to distinguish the data types of the atom from the string on output:

```
?- display( 'Atom'(`String`) ) ~> S, nl.

Atom(String)
S = ``
```

The new, corrected behaviour (as shown above in "Examples"), is in line with other, modern Prolog implementations.
dos/0

spawn a DOS shell

**dos**

**Comments**  
This predicate locates and executes a copy of the *command processor* (COMMAND.COM or CMD.EXE).

**Examples**  
The following command initiates a DOS session:

```
?- dos.
yes
```

**Notes**  
The *dos/0* predicate is really a left-over from **DOS-PROLOG**, which ran under the single-tasking MS-DOS and where such a "shell" predicate was the only way to temporarily switch from the Prolog environment back to the operating system. Under Windows, it requires no more than the click of a mouse to run one or more "DOS Boxes" alongside Prolog, and this renders the present predicate somewhat vestigial.

Please note that because Windows is a multitasking operating system, any shell spawned by *dos/0* will run in parallel with **WIN-PROLOG**: control may well return to Prolog before the shell has completed its operations.
**dos/1**

*spawn a DOS command*

\[\text{dos}(\text{Command})\]

+Command <atom>

**Comments**

This predicate locates a copy of the command processor (COMMAND.COM or CMD.EXE), and directs it to execute the given command.

**Examples**

The following command runs a DOS command to delete all backup files in the directory "C:\TEMP":

\[?- \text{dos}( 'del c:\temp\*.bak' ).\]

yes

**Notes**

The dos/1 predicate allows arbitrary DOS commands to be executed, including those that are built into the command processor and those defined as batch files. Conversely, the related predicate, exec/3, only allows the spawning of executable programs (.COM and .EXE files).

Please note that because Windows is a multitasking operating system, any commands spawned either by dos/1 or exec/3 run in parallel with WIN-PROLOG: control may well return to Prolog before the command has completed its operations. As such, it is important to write synchronisation code if the Prolog application depends upon the results of a spawned command. For example, such a command could delete an agreed temporary file as its last operation, and Prolog could sit in an idle loop waiting for this to happen before proceeding.
**dup/2**

*make a duplicate of a term*

\[ \text{dup( Term, Copy )} \]

+Term \ <term>  
-Copy \ <variable>

**Comments**

This predicate makes a duplicate of the given *Term*, and then binds this to *Copy*. All variables in the original *Term* are replaced with new ones in the *Copy*.

**Examples**

The following call makes a copy of a term, before binding variables in the original:

```
?- dup( foo(A,B,A), C ), A = a, B = b.  
A = a ,  
B = b ,  
C = foo(_1,_2,_1)  
```

**Notes**

Term duplication is generally useful in any case where it is desired to call a goal without binding its variables; for example, consider the following definition of a "for" loop:

```
for( _, Count, Limit, _ ) :-  
    Count > Limit,  
    !.  

for( X, Count, Limit, Goal ) :-  
    dup( (X = Count,Goal), NewGoal ),  
    NewGoal,  
    Next is Count + 1,  
    for( X, Next, Limit, Goal ).  
```

By duplicating the calls "X = Count, Goal" into a new term, "NewGoal", which has completely new variables, it is possible to bind successive integer values to (a copy of) "X". Once compiled, this "for" loop can be called as follows:

```
?- for( X, 1, 9, write(X) ), nl.  
```
X = _

The `dup/2` predicate is similar to, but slightly faster than the predicate, `copy_term/2`: the only difference is that the latter unifies, rather than binds, its second argument.
**dynamic/1**

*initialise all predicates specified by the given argument*

\[
dynamic( \text{Preds} )
\]

+Preds \hspace{1cm} <pred_specs>

**Comments**
This removes all clauses of any predicates specified in \textit{Preds}, freeing up memory resources for other programs. If the specified program(s) do(es) not exist, it simply succeeds; in both cases, it marks the predicates as "dynamic".

**Examples**
The following call ensures that any predicate called \textit{foo/1} has been deleted and marked dynamic:

\[
?- \text{dynamic}( \text{foo/1} ).
\]

yes

A number of predicates can be initialised in a single call, using either a list or a conjunction:

\[
?- \text{dynamic}( [\text{bar/2,sux/3,you/4}] ).
\]

yes

\[
?- \text{dynamic}( (\text{bar/2,sux/3,you/4}) ).
\]

yes

Because "dynamic" is defined as a prefix operator with a very high precedence, it is also possible to initialise predicates using the following syntax:

\[
?- \text{dynamic} \text{ foo/1} .
\]

yes

\[
?- \text{dynamic} \text{ bar/2, sux/3, you/4}. 
\]

yes

**Notes**
Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, \textit{WIN-PROLOG} immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are added to a program, they are inserted into memory immediately.
Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, **WIN-PROLOG** contains a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is a copy which is actually running. Consider the program:

```prolog
:- dynamic bar/1.

foo :-
    bar( X ),
    ( X = 1
     -> retract( bar(2) ),
             fail
     ;  X = 2
    ).

  bar( 1 ).
  bar( 2 ).
```

A call is set up to "bar(X)", and this will initially match "X" with "1", leaving a choicepoint set to the second clause. This second clause is now retracted, and failure is forced: Prolog will try to backtrack to the deleted clause, which will result in a serious error, most probably a "general protection fault" (GPF). By replacing the direct call to bar/1 with an indirect one using the dynamic_call/1 predicate, this danger is bypassed:

```prolog
:- dynamic bar/1.

foo :-
    dynamic_call( bar( X )
                 ),
    ( X = 1
     -> retract( bar(2) ),
             fail
     ;  X = 2
    ).
```
After compiling this version, the `foo/0` predicate will succeed on first call, and fail on second and subsequent ones:

```prolog
?- foo. yes ?- foo. no
```

This is the behaviour that we should expect: when first compiled, `bar/1` has two clauses, and as `dynamic_call/1` allows the `foo/1` program to execute a frozen copy of `bar/1` while deleting clauses from the real version, backtracking allows the "X = 2" line to succeed. On the second run (without first recompiling), there is only one clause for `bar/1`, and both the attempt to retract "bar(2)" and check that "X = 2" will fail. See the Appendix B to find out more about this feature.
**dynamic_call/1**

*call a dynamic predicate safely, limiting the scope of any cut*

\[
\text{dynamic_call( Goal )}
\]

+Goal  <goal>

**Comments**
This creates a frozen copy of a dynamic predicate, calls the given Goal with respect to this copy, and succeeds or fails depending upon whether the Goal itself succeeds or fails.

**Examples**
The following command asserts two clauses for a predicate, `foo/1` (this predicate will be automatically declared dynamic):

\[
\text{?- assert( foo(1) ), assert( foo(2) ).}
\]

yes

Now we will make a dynamic call to `foo/1`, and abolish it. Despite having deleted its definition after the first solution, it is still able to backtrack and generate the second:

\[
\text{?- dynamic_call( foo(X) ), abolish( foo/1 ).}
\]

\[
\text{X = 1 ;}
\]

\[
\text{X = 2}
\]

**Notes**
Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, **WIN-PROLOG** immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are added to a program, they are inserted into memory immediately.

Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, **dynamic_call/1** provides **WIN-PROLOG** with a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is the copy which is actually running. See the entry for **dynamic/1**, as well as Appendix B, to find out more.
elex/1

set, reset or get the Edinburgh language extensions flag

elex( Flag )

?Flag <variable> or <integer> in the domain {0,1}

Comments
This sets, resets or gets the current value of the Edinburgh language extensions Flag, which determines whether or not certain WIN-PROLOG-specific features are enabled.

Examples
The following call picks up the current setting of the Edinburgh language extensions flag, in this case showing that it is set:

?- elex( F ).
F = 1

Notes
WIN-PROLOG has a number of built-in extensions to the standard Edinburgh syntax: for example, it uses the backward-quote character (`) to denote strings, and "bar notation" can be used to denote compound terms of unknown arity:

?- `foo`(1,2,3) = X(1|Y).
X = `foo`
Y = [2,3]

When loading legacy source code from other Prolog implementations, it is possible that these notational extensions could cause syntax errors: for this reason, it can be desirable to switch them off while loading such files. The elex/1 predicate allows you to do just this:

?- elex( C ), elex( 0 ), consult( old_file ), elex( C ).
# 0.000 seconds to consult old_file.pl [c:\pro386w\]
C = 1

Please note that the elex/1 only affects the reading (input) of terms, and not their processing. Even if the flag is turned off, so that, for example, WIN-PROLOG strings are not recognised, programs which create and manipulate strings will still work. In particular, such terms are output without regard to the flag:

?- elex( 0 ), atom_string( 'hello world', S ).
S = `hello world`
empty/0

empty the current input stream

empty

Comments  
This input predicate is used to empty remaining characters from the current line in the input stream, and is used mainly in error handlers in response to mistyped user input.

Examples  
The following call performs two reads from the console, and even supplies the pair of terms ("a" and "b") to be read. However, because of the call to empty/0 after the first read, the second term is lost and a fresh read prompt is displayed, inviting the user to type in a term ("c" in this case):

```
?- read( Q ), empty, read( W ). a. b. <enter>
|: c. <enter>
Q = a ,
W = c
```

Notes  
When an error is handled by the system "error hook", a call is made to empty/0, after switching input back to the user console, in order to ensure that any unprocessed characters are discarded: were this not done, additional errors might result when the partially processed command line was re-read. Thus, when typing a line containing multiple syntax errors, such as the following, only one error is reported before the remaining input is emptied:

```
?- foo). bar(. sux :-) <enter>
! ----------------------------------------
! Error 42 : Syntax Error
! Goal     : ered(_1,_2)
```

Versions of WIN-PROLOG before 5.000 included this feature, but driven by the misnamed predicate, flush/0. From WIN-PROLOG 5.000 onwards, the latter predicate performs the expected flushing of buffered output.
encode/3

encode data from current input to current output

\[
\text{encode}( \text{Wanted, Password, Raw} ) \]

+Wanted <integer>
+Password <string>
-Raw <variable>

Comments

This copies up to the number of characters Wanted, from the current input stream to the current output stream, encoding (encrypting) them along the way, using the given Password. It terminates when the given number of characters has been processed, or end of file is encountered in the input stream. The raw character count is returned in Raw.

Examples

The following simple programs can be used to create or unpack encrypted archives of multiple files, using a secret password:

\[
\begin{align*}
\text{encode_archive}( \text{Password, Archive, Files} ) :&= \\
&
\text{open}( \text{Archive, write} ), \\
&\text{tell}( \text{Archive} ), \\
&\forall( \text{member( File, Files )}, \\
&\left( \text{writeq( File )}, \\
&\text{write( ' ' )}, \\
&\text{see( File )}, \\
&\text{encode( -1, Password, _ )}, \\
&\text{seen}
&\right) ) \\
&\text{told},
\end{align*}
\]

\[
\begin{align*}
\text{decode_archive}( \text{Password, Archive} ) :&= \\
&\text{see}( \text{Archive} ), \\
&\text{repeat}, \\
&\text{read( File )}, \\
&\text{tell( File )}, \\
&\text{decode( Password, _ )}, \\
&\text{told},
\end{align*}
\]
Once compiled, this program can be used to create an encoded (encrypted) archive of multiple files; the following call, for example, will create an archive called "foo" which includes encrypted copies of two other files, "bar" and "sux", using the secret password, "abc":

```prolog
?- encode_archive( `abc`, foo, [bar,sux] ). yes
```

These files can be recovered from the archive on a subsequent occasion, provided the password is known, with the call:

```prolog
?- decode_archive( `abc`, foo ). yes
```

Notes

The MZSS algorithm, as used to encrypt and decrypt data in the `encode/3` and `decode/2` predicates respectively, is a highly-secure 1185-bit single key encryption system. It uses a password string of up to 148 characters is to seed a sophisticated "Marsaglia Zaman" pseudo random number generator ("PRANG"), which is completely independent from the "linear congruential" PRANG used by `is/2` and other arithmetic predicates. A comb-filtering process uses only randomly selected numbers from the Marsaglia Zaman data stream, avoiding any chance of using a knowledge of recent history to try to predict future numbers.

One common attack on an encrypted document is to use a second document for which both "plaintext" and "cyphertext" versions are available, and which is thought to have been encoded with the same password as the document under attack. MZSS is completely immune to this, because a completely different random sequence is guaranteed to be generated every time, even for a given password. The current system date and time are combined with the seed generated from the user's password, and "published" in the cyphertext file to enable subsequent decryption to use the same "sequence variation". See Appendix K for more information about MZSS encryption.
ensure_loaded/1

ensure the most recent versions of files are loaded

\[\text{ensure} \text{ loaded}( \text{FileSpec} )\]

+FileSpec \text{<file_specs>}

Comments
This predicate loads the programs contained in the disk file or files specified in FileSpec, unless they have already been loaded and have not been modified since. Any source files are loaded with the incremental compiler; any object files are loaded directly. If the file has previously been loaded, and since modified, the existing copy is abolished before reloading.

Examples
The following call loads the example program, "BENCHMRK.PL", into memory:

?\- ensure_loaded( examples(benchmrk) ).
# 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
yes

Running the same query a second time simply succeeds, because the file has not changed since being loaded:

?\- ensure_loaded( examples(benchmrk) ).
yes

Notes
This predicate is very similar to load_files/1, the only difference being that the latter always loads files, whether or not they have been loaded previously. Where both source (.PL) and object (.PC) files exist for the same program, both predicates load the most recent, based on the files' timestamps.

Code loaded with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed. See the entries for compile/1 and consult/1 for further information about loading source files, and optimize/1 and optimize_files/1 for information about how to optimise code.

When predicates are loaded from a file, they retain an association with that file: if the file is reloaded, these predicates are automatically abolished. This provides a convenient method for "cleaning up" between successive runs of a system, especially where dynamic predicates have been defined in the source file. Any commands present in the source file are executed on the fly, as they are encountered by the loader.
If `FileSpec` does not specify a file extension, one of the "source" and "object" extensions (normally `.PL` or `.PC` respectively) is assumed, the choice depending upon which of these names the most recent, existing file.

Please note that the current method of maintaining the relationship between a source file and its object file is based purely on the source file's name, minus its extension. If a source file is compiled, and its object file subsequently renamed, then `ensure_loaded/1` and `load_files/1` will be unable to determine that they are two versions of the same file. Similarly, if source or object files are given non-standard extensions, neither of these predicates will know how to load the "latest" version.
**env/1**

*get a list of all environment variables*

```
env( Block )
```

- Block <variable>

**Comments**

This predicate returns a list comprising the entire `Block` of "environment variables" that were inherited by the currently executing instance of WIN-PROLOG; each entry in the list is a conjunction of an uppercase atom and a mixed case string, respectively corresponding to a variable's name and value.

**Examples**

The following call returns the environment variables as a list of atom/string pairs:

```
?- env( Block ).
```

```
Block = [['COMSPEC', 'C:\COMMAND.COM'], ['PATH', ...]]
```

**Notes**

The environment variable table is a set of strings, each one of the form "name=value", which are maintained by the operating system and passed to an application when it is executed. In most cases, the name of a variable is in uppercase, but newer versions of Windows often have one or more lowercase or mixed case variables: to avoid confusion, `env/1` always returns variable names in uppercase, although it does not modify the case of any characters their values.

Versions of WIN-PROLOG before 4.100 had a different predicate, `env/2`, which took a given variable name and returned its value (if any), as a list of atoms parsed and split at any embedded semicolons. There were two problems with this old predicate: firstly, the parsing at semicolons, while ideal for the "PATH" variable, was a problem in most other cases, and secondly, it provided no way to access environment variables unless their names were known in advance. The new `env/1` predicate solves both these problems.

One feature present in the old `env/2`, but absent in `env/1`, was the ability to pick up the full file name of the currently executing instance of WIN-PROLOG: the `fname/4` predicate has been extended to provide this service.
eof/0

test to see if the input stream is at end of file

eof

Comments
This predicate succeeds when the current input stream is at the end of file position, where subsequent attempts at input will normally result in failure or an error condition.

Examples
The following call checks whether the end of the input file "foo.pl" has been reached: if so, it closes the input stream; otherwise, it reads a term from the file. The following example assumes the file was indeed at its end-of-file position:

?- see( foo ), (eof -> seen ; read(X) ).
X = _

Notes
The eof/0 predicate is identical to at_end_of_file/0: the latter is simply implemented as a call to the former, for compatibility with Quintus Prolog:

at_end_of_file :-
eof.

Most input predicates fail the first time they are called when the input stream is at end-of-file, and generate an error on subsequent attempts. As well as allowing programs to check for the end-of-file explicitly, eof/0 clears the "error-next-time" flag, so a subsequent input attempt will once again fail.

In WIN-PROLOG, this predicate has an additional use. When reading from the console window (the "user" device), eof/0 can be used to check whether there are any additional characters still to be read from the current command line, allowing programs to read the entire contents of a console command, but to stop short of requesting further user input.
**eprint/1**

`write a quoted term to the current output stream`

```prolog
    eprint( Term )
```

*Comments*  
This writes the given `Term` to the current output stream, in quoted Edinburgh syntax.

*Examples*  
The following command outputs the given term with quotes where necessary:

```prolog
    {?- eprint( 'String'+Atom=Var ), nl.
      'String' + 'Atom' = _1
      Var = _
    }
```

*Notes*  
This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (`'`), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes (''). Finally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation:

```prolog
    {?- eprint( 'hello-M~Jworld' ), nl.
      'hello-M~Jworld'
      yes
    }
```

Terms output to a file in quoted syntax are readable provided they are followed by a period (.) and at least one space or control character. This is not provided automatically, and must therefore be output explicitly if readability is required:

```prolog
    {?- eprint( foo(X) ), ewrite( '. ' ), nl.
      foo(_1).
      X = _
    }
```

The `eprint/1` predicate is entirely equivalent to the standard Edinburgh predicate, `writeq/1`. Unlike the latter, however, `eprint/1` forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output with correct bracketing with respect to any operator precedences (see `eprint/2` and `eprint/3` respectively).
\textbf{eprint/2}

\textit{write a quoted term with named variables to current output}

\begin{verbatim}
eprint( Term, Vars )
\end{verbatim}

\begin{verbatim}
?Term <term>
+Vars <varlist>
\end{verbatim}

\textbf{Comments} \par This writes the given \textit{Term} to the current output stream, in quoted Edinburgh syntax, naming any variables listed in \textit{Vars} with their corresponding names.

\textbf{Examples} \par The following command outputs the given term with quotes where necessary, using the given named variable:

\begin{verbatim}
?- eprint( `String`+'Atom'=Var, [('X',Var)] ), nl.
`String` + 'Atom' = X
Var = _
\end{verbatim}

\textbf{Notes} \par This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (`), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Finally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see \texttt{eprint/1}).

The ability to preserve or create variable names during input and output is a key feature of \textbf{WIN-PROLOG}: predicates such as \texttt{eread/2} return the list of variable names used in a term, and these can then be used in predicates like \texttt{eprint/2}:

\begin{verbatim}
?- eread( T, V ), eprint( T, V ), nl.
|:- foo(VAR1,VAR2).
foo(VAR1,VAR2)
T = foo(_1,_2) ,
V = [('VAR1',_1),('VAR2',_2)]
\end{verbatim}

It is this feature that allows \textbf{WIN-PROLOG} to display program listings with the variable names, and even to display the results of queries to the console! A special predicate, \texttt{vars/2}, can be used to identify and name all variables in a term using the sequence, [A..Z], [A1..Z1], [A2..Z2], and so forth, with all single-use variables being given the "anonymous variable" name of "_" (underscore). This list can be used to name variables neatly during output.
eprint/3

write a quoted term with named variables and precedence or truncated output

\[ eprint( \text{Term}, \text{Vars}, \text{Precedence} ) \]

?Term <term>
+Vars <varlist>
+Precedence <integer> in the range \([-1..1200]\) or <integer> with other negative value

Comments
This writes the given \text{Term} to the current output stream, in quoted Edinburgh syntax, naming any variables listed in \text{Vars} with their corresponding names, using the given \text{Precedence}, or at the standard precedence of 1200, with automatic truncation, by specifying a negative \text{Precedence} other than -1 whose absolute value represents the maximum number of characters to output.

Examples
The following command outputs the given term with quotes where necessary, using the given operator precedence:

\[
\text{?- eprint( `String`+'Atom'=Var, [], 300 ), nl.}
\]

\`
(`String` + 'Atom' = _1)
Var = _
\`

By using a "precedence" of -1, use of operators is suppressed:

\[
\text{?- eprint( `String`+'Atom'=Var, [], -1 ), nl.}
\]

\`
=(+(`String`,'Atom'),_1)
Var = _
\`

The following command outputs a term, automatically truncated to just 9 characters:

\[
\text{?- eprint( `the quick brown fox`, [], -10 ).}
\]

\`
`the quickno
\`

Notice that \text{eprint/3} has "failed" in this instance, resulting in "no" being displayed by the supervisor: see "Notes" below for an explanation.

Notes
This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (`), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see \text{eprint/1}).
As well as allowing variables to be named (see `eprint/2`), the `eprint/3` predicate allows terms to be output recursively, with automatic bracketing of subterms where necessary. If the given `Term` is named by an operator whose precedence exceeds the given `Precedence`, parentheses are output around the term; otherwise, they are not. When outputting the argument of an operator, the `Precedence` should be obtained by calling the `current_op/3` predicate. To write a term in "canonical form", the `Precedence` should be set to "-1".

Truncated output is a new feature in **WIN-PROLOG** 5.000, and was introduced to handle the output of long or infinite terms more gracefully than before. For example, prior to version 5.000, the following command generates an error (and a misnomer!):

```
?- X = [foo|X].
```

Term too deep

In version 5.000 and later, this is handled elegantly, with the supervisor showing just the first 1024 characters of the infinite solution (shown here further truncated for convenience):

```
?- X = [foo|X].
X = [foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,foo,...
```

The `eprint/3` truncation feature is designed to fail the predicate if truncation occurs, and to succeed if not, giving calling programs the chance to handle truncation. For example, consider this program:

```
truncate( Term, Size ) :-
    Prec is - Size,
    ( eprint( Term, [], Prec ) -> true
    ;   write( `<---TRUNCATED!'` )
    ).
```

The following calls show how the failure can be used to display a truncation warning message:

```
?- truncate( hello_world, 11 ), nl.
hello_world
yes

?- truncate( hello_world, 10 ), nl.
```

hello_world<---TRUNCATED!

yes
eqv/2

test the equivalence of two terms

    eqv( Term1, Term2 )

?Term1 <term>
?Term2 <term>

Comments  This tests the equivalence of two terms, Term1 and Term2, which indicates whether or not they have the same structure.

Examples  The following commands test some cases:

?- eqv( foo(X), foo(Y) ).            <enter>
    X = _ ,
    Y = _

| ?- eqv( [X,Y], [Z,Z] ).            <enter>
    X = _ ,
    Y = _ ,
    Z = _

| ?- eqv( foo(X), foo(123) ).        <enter>
    no

Notes   Two terms are considered to be "equivalent" if their structures are identical, and every number, string and atom in one term is matched by an identical one in the other; however, the values of unbound variables are ignored.

When eqv/2 succeeds with two terms, it is always the case that these two terms can be unified; when it fails, however, it is still possible that the two terms might unify:

| ?- T1 = foo(X), T2 = foo(123), eqv( T1, T2 ). <enter>
    no

| ?- T1 = foo(X), T2 = foo(123), T1 = T2. <enter>
    T1 = T2 = foo(123) ,
\[ X = 123 \]

The `eqv/2` predicate is similar in many respects to `cmp/3`: the main difference is that the latter does take account of the values of unbound variables, while the former does not.
eread/1

read an Edinburgh term from the current input stream

eread( Term )

-Term <variable>

Comments
This reads an Edinburgh syntax Term from the current input stream.

Examples
The following command reads a term from the console:

?- eread( T ).
<enter>
|: foo(VAR1,VAR2).
<enter>
T = foo(_1,_2)

Notes
This predicate inputs terms that have been written in "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (``), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes (' ) (see eprint/1). Terms must be terminated by a period (.) and at least one space or control character.

The eread/1 predicate is more or less equivalent to the standard Edinburgh predicate, read/1. Unlike the latter, however, eread/1 forms part of a family of more advanced term input predicates that allow variable names to be preserved on input and named during subsequent output (see eprint/2).

Error handling in the "eread" family is simpler than in read/1: firstly, if the input stream is at end of file, a call to eread/1 or eread/2 simply fails; read/1 returns the atom, "end_of_file", for which programs must explicitly test; secondly, if a syntax error is encountered during input, the "eread" predicates report this directly using the standard error handler, while the read/1 predicate attempts to process the error internally, depending upon the settings of assorted Prolog flags.
eread/2

read an Edinburgh term and variables from current input

\[ \text{eread( Term, Vars )} \]

- Term \text{<variable>}
- Vars \text{<variable>}

Comments This reads an Edinburgh syntax Term from the current input stream, returning a list of any variables with their corresponding names in Vars.

Examples The following command reads a term and variable list:

\[
?\text{- eread( T, V ).}\\
|: \text{foo(VAR1,VAR2).}\\
T = \text{foo(_1,_2)},\\
V = [(\text{VAR1}',_1),(\text{VAR2}',_2)]
\]

Notes This predicate inputs terms that have been written in "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes ("), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes (') (see eprint/1). Terms must be terminated by a period (.) and at least one space or control character.

The ability to preserve or create variable names during input and output is a key feature of \text{WIN-PROLOG}: predicates such as eread/2 return the list of variable names used in a term, and these can then be used in predicates like eprint/2:

\[
?\text{- eread( T, V ), eprint( T, V ), nl.}\\
|: \text{foo(VAR1,VAR2).}\\
\text{foo(VAR1,VAR2)}\\
T = \text{foo(_1,_2)},\\
V = [(\text{VAR1}',_1),(\text{VAR2}',_2)]
\]

It is this feature that allows \text{WIN-PROLOG} to display program listings with the variable names, and even to display the results of queries to the console! A special predicate, vars/2, can be used to identify and name all variables in a term using the sequence, [A..Z], [A1..Z1], [A2..Z2], and so forth, with all single-use variables being given the "anonymous variable" name of "_" (underscore). This list can be used to name variables neatly during output.
errmsg/2

return the error message text for a given error number

```
errmsg( Number, Text )
```

+Number  <integer>
-Text    <variable>

Comments  This takes a given error Number, and returns an atom consisting of the appropriate error's Text. Any nonexistent error number returns the text "Unknown Error".

Examples  The following command returns the text message associated with error 10:

```
?- errmsg( 10, T ).
T = 'Window Handling Error'
```

Notes  The errmsg/2 predicate is primarily of use in error handlers written either with the ’?ERROR?’/2 hook, or with catch-and-throw (see catch/2, catch/3 and throw/2). It retrieves messages coded directly into the WIN-PROLOG kernel, but not those added by toolkits and plug-ins: these, together with the kernel messages, can be obtained using error_message/2.
'?ERROR?'/2

user-defined hook to gain control after an error

'?ERROR?'( Number, Goal )

?Number  <integer> or <variable>
?Goal  <goal> or <variable>

Comments This is not a built-in predicate, but an optional hook that can be written by the user to gain control after an error. The error number is unified with Number, and the faulty goal is unified with Goal.

Examples The following program will report all errors with a simple message to the console, before aborting execution:

```
'?ERROR?'( Number, Goal ) :-
   errmsg( Number, Message ),
   output( 0 ),
   writeq( error(Number) - Message - Goal ),
   nl,
   abort.
```

After compiling this program, the error handler can be tested:

```
?- foo(bar).
error(20) - 'Predicate Not Defined' - foo(bar)
```

Notes The '?ERROR?'/2 hook predicate is only significant during program development, while the development environment is active. In stand-alone applications, any arity-two predicate can be nominated for the task of recovering after an error. Because the requirements of a stand-alone application may differ from that of development-time testing, it is usually best to choose a predicate other than '?ERROR?'/2 in applications.

In the absence of a user-supplied definition of '?ERROR?'/2, the WIN-PROLOG development environment will call the built-in predicate, error_hook/2. This displays a diagnostic message box at the console, before aborting execution of the program.

Please note that WIN-PROLOG includes a powerful catch-and-throw error handler in addition to the '?ERROR?'/2 hook mechanism (see catch/2, catch/3 and throw/2).
error_hook/2

default error hook in development environment

error_hook( Number, Goal )

+Number       <integer>
+Goal         <goal>

Comments  This predicate is called by the development environment to report when an error with the given Number has occurred during execution of the given Goal. It is normally only called directly from a user-defined error hook (see '?ERROR?'/2).

Examples  This predicate should not normally be called directly, apart from in the very special case of a user-defined "error hook" (see '?ERROR?'/2). Its sole function is to display a diagnostic message box at the console, before aborting program execution. As it is also run automatically by the development environment after an error occurs, it is easiest to demonstrate indirectly, by typing a goal containing an error:

?- PI is TWENTY_TWO / SEVEN .
! ----------------------------------------
! Error 22 : Instantiation Error
! Goal     : _1 is _2 / _3

Notes  The error_hook/2 predicate is normally only used during program development, while the development environment is active. While it can be called in stand-alone applications, most such systems have run-time requirements which differ from those at development time: any arity-two predicate can be nominated for the task of recovering from an error.

Please note that WIN-PROLOG includes a powerful catch-and-throw error handler in addition to the error_hook/2 predicate (see catch/2, catch/3 and throw/2).
error_message/2

return the error name for a given error number

   error_message( Number, Text )

+Number          <integer>
-Text            <variable>

Comments  This take a given error Number, and returns an atom consisting of the appropriate error's Text, including errors installed by toolkits and plug-ins. Any nonexistent error number returns the text "Unknown Error".

Examples  The following command returns the text message associated with error 10:

   ?- error_message( 10, T ).
   T = 'Window Handling Error'

Notes  The error_message/2 predicate is primarily of use in error handlers written either with the '?ERROR?/2 hook, or with catch-and-throw (see catch/2, catch/3 and throw/2). It retrieves messages coded directly into the WIN-PROLOG kernel as well as any added by toolkits and plug-ins: sometimes it might be desirable to restrict error messages to those from within the kernel, and these can be obtained using errmsg/2.

etoks/1

read an Edinburgh token list from the current input stream

    etoks( Tokens )

-Tokens <variable>

Comments This reads a sequence of Edinburgh syntax Tokens, representing a term, from the current input stream. The tokens are returned as a list of conjunctions, each having the form (type, size, token), where type is an integer that qualifies the token, size is the number of characters in the token. The different types are shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>variable</td>
</tr>
<tr>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>2</td>
<td>floating point number</td>
</tr>
<tr>
<td>3</td>
<td>atom (quoted or unquoted)</td>
</tr>
<tr>
<td>4</td>
<td>string (&quot;...&quot;)</td>
</tr>
<tr>
<td>5</td>
<td>empty char list (&quot;&quot;)</td>
</tr>
<tr>
<td>6</td>
<td>char list (&quot;...&quot;)</td>
</tr>
<tr>
<td>7</td>
<td>punctuation character, space or comment</td>
</tr>
<tr>
<td>8</td>
<td>incomplete token because of end-of-file</td>
</tr>
</tbody>
</table>

The penultimate category, 7, distinguishes between its cases by returning the following tokens:

<table>
<thead>
<tr>
<th>Token</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>white space</td>
</tr>
<tr>
<td>'%'</td>
<td>comment</td>
</tr>
<tr>
<td>'.'</td>
<td>end of term</td>
</tr>
<tr>
<td>&lt;other&gt;</td>
<td>single character token</td>
</tr>
</tbody>
</table>

The final category, 8, is used to return the type of token that was being parsed at the point an end of file condition occurred:
Token | Indicates
---|---
 '_' | unquoted token
 '%' | comment token
 '~' | quoted atom token
 `'' | string token
 | char list token

<table>
<thead>
<tr>
<th>Token</th>
<th>Indicates</th>
</tr>
</thead>
</table>

### Examples

The following command reads a token list from the console:

```prolog
?- etoks( T ).
|: foo(VAR1,VAR2).
T = [(3,3,foo),(7,1,'('),(0,4,_1),(7,1,(')'),'(0,4,_2),(7,1,'')),(7,2,'')]```

### Notes

This predicate inputs lists of tokens from terms that have been written in "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes ('), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('') (see eprint/1). Terms need not be terminated by a period (.) and at least one space or control character, but if they are, this will be returned as the final token in the list.

Strictly speaking, etoks/1 reads token lists according to the Edinburgh lexical rules, rather than its syntactic rules. This can be seen by reading tokens from a syntactically incorrect term which would generate errors in eread/1 or read/1:

```prolog
?- etoks( T ).
|: foo(VAR,{}).
T = [(3,3,foo),(7,1,'('),(0,3,_1),(7,1,(')')), (7,1,{}),(7,1,'{}'), (7,2,'')]```

All tokens are read up to the terminating period and white space, and this terminator is returned as the final token. If end-of-file is encountered before this <dot><space> delimiter, the absence of this final token in the returned list will indicate the situation.
etoks/2

read an Edinburgh token list and variables from input

\[ \text{etoks} \left( \text{Tokens, Vars} \right) \]

- \text{Tokens} <variable>
- \text{Vars} <variable>

Comments
This reads a sequence of Edinburgh syntax \text{Tokens}, representing a term, from the current input stream, returning a list of any variables with their corresponding names in \text{Vars}. The tokens are returned as a list of conjunctions, each having the form (type,size,token), as described in the entry for \text{etoks/1}.

Examples
The following command reads a token list and variable list:

\[ ?- \text{etoks} \left( T, V \right). \]
\[ \text{\textbf{|}}: \ \text{foo}(\text{VAR1}, \text{VAR2}). \]
\[ T = \[(3,3,\text{foo}),(7,1,\text{'}\text{'\text{'}),(0,4,\text{'}_1),(7,1,\text{'\text{'}\text{'\text{'}),(0,4,\text{'}_2),(7,1,\text{'\text{'}\text{'\text{'}}),(7,2,\text{'\text{'}})\]}
\[ V = [(\text{'}\text{\text{'}VAR1'},\text{'}_1),(\text{'}\text{\text{'}VAR2'},\text{'}_2)] \]

Notes
This predicate inputs lists of tokens from terms that have been written in "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (\text{'}), and any atoms beginning with the underscore (\text{'_}), uppercase letters or containing a mix of lexical types are delimited by single quotes (\text{'}) (see \text{eprint/1}). Terms need not be terminated by a period (\text{'}.) and at least one space or control character, but if they are, this will be returned as the final token in the list.

Strictly speaking, the \text{etoks/2} predicate reads token lists according to the Edinburgh lexical rules, rather than its syntactic rules: see the entry for \text{etoks/1} for further details.

The ability to preserve or create variable names during input and output is a key feature of \text{WIN-PROLOG}: predicates such as \text{etoks/2} return the list of variable names used in a term, and these can then be used in predicates like \text{eprint/2}; see the entry for \text{eread/2} for more information and an example.
**ewrite/1**

*write an unquoted term to the current output stream*

```
ewrite(Term)
```

**Comments**

This writes the given *Term* to the current output stream, in unquoted Edinburgh syntax.

**Examples**

The following command outputs the given term without quotes, so that atoms and strings appear the same:

```
?- ewrite( `String`+'Atom'=Var ), nl.
String + Atom = _1
Var = _
```

**Notes**

This predicate outputs terms using "unquoted" Edinburgh syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally:

```
?- ewrite( 'hello~M~Jworld' ), nl.
hello
world
yes
```

Terms output to a file in unquoted syntax are not generally readable, even if they are followed by a period (.) and at least one space or control character, because unquoted strings and atoms can be confused with each other and even with variables, or may contain characters which give rise to syntax errors.

The *ewrite/1* predicate is entirely equivalent to the standard Edinburgh predicate, *write/1*. Unlike the latter, however, *ewrite/1* forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output with correct bracketing with respect to any operator precedences (see *ewrite/2*, *eprint/2* and *eprint/3* respectively).
**ewrite/2**

*write an unquoted term with named variables to output*

```
   rewrite( Term, Vars )
```

**Comments**

This writes the given `Term` to the current output stream, in unquoted Edinburgh syntax, naming any variables listed in `Vars` with their corresponding names.

**Examples**

The following command outputs the given term without quotes, so that atoms and strings appear the same, using the given named variable:

```
?- rewrite(`String`+'Atom'=Var, [('X',Var)]), nl.
```

```
String + Atom = X
Var = _
```

**Notes**

This predicate outputs terms using "unquoted" Edinburgh syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally (see `ewrite/1`).

The ability to preserve or create variable names during input and output is a key feature of **WIN-PROLOG**: predicates such as `eread/2` return the list of variable names used in a term, and these can then be used in predicates like `ewrite/2`:

```
?- eread( T, V ), rewrite( T, V ), nl.
```

```
|: foo(VAR1,VAR2).
foo(VAR1,VAR2)
T = foo(_1,_2),
V = [('VAR1',_1),('VAR2',_2)]
```

It is this feature that allows **WIN-PROLOG** to display program listings with the variable names, and even to display the results of queries to the console! A special predicate, `vars/2`, can be used to identify and name all variables in a term using the sequence, [A..Z], [A1..21], [A2..22], and so forth, with all single-use variables being given the "anonymous variable" name of "_" (underscore). This list can be used to name variables neatly during output.
exec/3

spawn an executable file

\[
\text{exec( Program, Command, Code )}
\]

+Program <atom>
+Command <atom>
-Code <variable>

Comments
This predicate directly executes the named Program, passing it the given Command line, and returning its exit Code.

Examples
The following command runs a copy of "NOTEPAD.EXE", passing it the name of a file to edit:

\[
?- \text{exec('notepad.exe','c:\pro386w\temp.pl',X).} <\text{enter>}
\]

X = 0

Notes
The exec/3 predicate only allows the spawning of executable programs (.COM and .EXE files). Conversely, the related predicate, dos/1, allows arbitrary DOS commands to be executed, including those that are built into the command processor and those defined as batch files.

Please note that because Windows is a multitasking operating system, any commands spawned either by dos/1 or exec/3 run in parallel with WIN-PROLOG: control may well return to Prolog before the command has completed its operations. As such, it is important to write synchronisation code if the Prolog application depends upon the results of a spawned command. For example, such a command could delete an agreed temporary file as its last operation, and Prolog could sit in an idle loop waiting for this to happen before proceeding.
exit/1

exit directly to or shut down the operating system

    exit( Code )

+Code <integer>

Comments
This predicate immediately terminates the current Prolog session, directly exiting to the operating system and passing it the given exit Code, and optionally shutting the system down.

Examples
If you wanted to terminate the current Prolog session abruptly, you could type the following command; however, please note that you will lose any unsaved program files:

    ?- exit( 0 ).

Notes
The exit/1 predicate performs much the same job as either halt/0 or halt/1, the only difference being that it is a little more direct. Both the halt/n predicates perform an assortment of checks before finally closing a Prolog session: within the development environment, for example, they both check for unsaved program files, and offer the chance to save them. As another example, when testing applications with the “Run/Application” menu item (Developer Edition only), both halt/n predicates are intercepted, so that your application "returns" to the development environment upon completion.

In effect, the two halt/n predicates are written in terms of each other and of exit/1 much as follows:

    halt( Code ) :-
        { ... check if testing application ... 
          -> true
          ; ... see if any files need saving ... 
          -> system_menu( 1, file, save_all ),
              exit( Code )
          ; exit( Code )
        }.

Because exit/1 always exits directly, without offering to save files, it should be used with care: halt/0 and halt/1 are recommended for more general usage.
There are three special values for `Code` which enable `exit/1` to force or resume a shutdown of the current Windows session, as shown in the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>force Windows to shut down current session</td>
</tr>
<tr>
<td>-2</td>
<td>force Windows to log off current user</td>
</tr>
<tr>
<td>-3</td>
<td>force Windows to shut down and reboot machine</td>
</tr>
</tbody>
</table>

These special exit codes should be reserved for special situations, such as when a user-initiated shutdown sequence has been intercepted by `WIN-PROLOG`: when the user attempts to shut down, log off from, or reboot Windows, a message called "WM_QUERYENDSESSION" is sent to each running process. Normally, processes simply return the value "TRUE" to Windows, indicating that they are ready to be closed, but in the event that they need time to save files or perform other housekeeping, they have the option of returning "FALSE", thereby terminating the shutdown sequence.

In `WIN-PROLOG`, a special "shutdown flag" can be used to determine the response to this message (see `safe/1`). When clear (the default), Windows is allowed to continue with shutdown, but when set, Windows is prevented from shutting down, and the "main window" ("0") receives a special "msg_close" message, containing a mode parameter which indicates which type of shutdown is being attempted (these relate directly to the codes shown in the table above). In response to this message, a window handler should perform any necessary housekeeping, before calling `exit/1` with this mode parameter as the `Code` argument, which has the effect of resuming the appropriate Windows shutdown sequence.
expand_dcg/2

*convert grammar rules to Prolog directly*

\[
\text{expand_dcg( Term1, Term2 )}
\]

+Term1 <term>

?Term2 <term> or <variable>

**Comments**

If Term1 is a grammar rule then Term2 is the corresponding Prolog representation of that rule using default expansion; otherwise the two terms are identical.

**Examples**

The following command expands a simple definition of a sentence "s" as consisting of a verb phrase "vp" and a noun phrase "np", returning it expressed as a Prolog rule:

\[
?- \text{expand_dcg((s --> vp, np), X).}
\]

\[
X = (s(_1,_2) :- vp(_1,_3),np(_3,_2))
\]

**Notes**

The expand_dcg/2 predicate is very closely related to expand_term/2; both these predicates perform term expansion in exactly the same way unless another predicate, term_expansion/2, has been defined by the user. This hook allows special grammar rules to be added to the default grammar rule processing: however, such grammar rule extensions are recognized only by expand_term/2, and not by expand_dcg/2, which still performs default term expansion.
**expand_term/2**

*convert grammar rules to Prolog indirectly*

```
expand_term( Term1, Term2 )
```

+Term1 <term>

?Term2 <term> or <variable>

**Comments**

If Term1 is a grammar rule then Term2 is the corresponding Prolog representation of that rule using default or user-defined expansion; otherwise the two terms are identical.

**Examples**

The following command expands a simple definition of a sentence "s" as consisting of a verb phrase "vp" and a noun phrase "np", returning it expressed as a Prolog rule:

```
?- expand_term((s --> vp, np), X).
<enter>
X = (s(_1,_2) :- vp(_1,_3),np(_3,_2))
```

**Notes**

The `expand_term/2` predicate is very closely related to `expand_dcg/2`; both these predicates perform term expansion in exactly the same way unless another predicate, `term_expansion/2`, has been defined by the user. This hook allows special grammar rules to be added to the default grammar rule processing; however, such grammar rule extensions are recognized only by `expand_term/2`, and not by `expand_dcg/2`, which still performs default term expansion.
fail/0

force failure

fail

Comments This predicate simply fails, and is used to force a program to backtrack into its most recent choicepoint.

Examples Consider the following program:

```prolog
foo( 1 ).
foo( 2 ).
foo( 3 ).
```

By forcing failure, it is possible to display all solutions:

```prolog
?- (foo( X ), write( X ), nl, fail) ; true.
1
2
3
X = _
```

Notes The fail/0 predicate is one of a family of control predicates that can be used to drive Prolog programs in ways which are not purely logical. In conjunction with the cut (!/0), it permits implementation of features such as negation (\+/0 and not/0) and solution-set predicates (bagof/3, setof/3 and findall/3).

One particular predicate, forall/2, provides an alternative method of backtracking through the full set of choicepoints, in a way that is considered "cleaner" than using explicit failure:

```prolog
?- forall( foo(X), (write(X), nl) ).
1
2
3
X = _
```
false/0

alternative force failure

false

Comments  This predicate simply fails, and is used to force a program to backtrack into its most recent choicepoint.

Examples  Consider the following program:

    foo( 1 ).
    foo( 2 ).
    foo( 3 ).

By forcing failure, it is possible to display all solutions:

    ?- (foo( X ), write( X ), nl, false) ; true.  
        1
        2
        3
        X = _

Notes  Please note that this predicate, false/0, is in fact identical to fail/0; and is provided as a synonym for compatibility with programs written for some earlier Prolog systems; new applications should use the latter predicate.
fclose/1

close a disk or memory file

fclose( Name )

+Name <atom>

Comments
This closes the disk or memory file referenced by the given Name, flushing any pending output to disk, freeing its resources and resetting input and output streams if necessary.

Examples
The following call closes the file "foo", freeing its resources:

?- fclose( foo ).
yes

Notes
The fclose/1 predicate is used to close disk or memory files created by fcreate/5; the name it uses is that supplied as a first argument handle to the latter, and this need not be related in any other way to the actual file name: please see the entry for fcreate/5 for further information.

Up to 64 separate files, whether disk or memory based, may be simultaneously open. In many cases, there will be no need to close them during the run of an application: WIN-PROLOG automatically closes and flushes disk files upon halting.

This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
fcreate/5

create or open a disk or memory file

\[ \text{fcreate( Name, File, Mode, Size, Code )} \]

+Name <atom>
+File <atom>, <address> or <emptylist>
+Mode <integer> in the range \([-2..2]\]
+Size <integer>
+Code <integer> in the range \([-5..5]\]

Comments

This creates or opens a disk or memory file using the given Name and File identifier, in the given Mode, with the given buffer Size and with the character encoding specified by Code. The Mode determines whether a new file is created or an existing one opened, and also determines its access permissions, as defined in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>create a new file in READ/WRITE mode</td>
</tr>
<tr>
<td>-1</td>
<td>create a new file in WRITE-ONLY mode</td>
</tr>
<tr>
<td>0</td>
<td>open an existing file in READ-ONLY mode</td>
</tr>
<tr>
<td>1</td>
<td>open an existing file in WRITE-ONLY mode</td>
</tr>
<tr>
<td>2</td>
<td>open an existing file in READ/WRITE mode</td>
</tr>
</tbody>
</table>

The Code determines which of eleven character encodings to use to access data in the file, as defined in the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>32-bit integer raw-32le format</td>
</tr>
<tr>
<td>-4</td>
<td>16-bit integer raw-16le format</td>
</tr>
<tr>
<td>-3</td>
<td>unicode utf-32le format</td>
</tr>
<tr>
<td>-2</td>
<td>unicode utf-16le format</td>
</tr>
<tr>
<td>-1</td>
<td>7-bit ascii utf-8 format</td>
</tr>
<tr>
<td>0</td>
<td>8-bit iso/iec 8859-1 format</td>
</tr>
<tr>
<td>1</td>
<td>unicode utf-8 format</td>
</tr>
</tbody>
</table>
See Appendix L for more information about character sets, encodings and formats.

If the File identifier is specified as an atom, it is treated as the name of a disk file. If File is an empty list ([]), then a memory buffer is allocated and treated as a virtual file. Finally, if File is an address, the area of memory specified is used as a virtual file.

Examples

The following call creates a new disk file called "c:\foo.txt", with write-only permissions, giving it the Prolog name "foo":

```
?- fcreate( foo, 'c:\foo.txt', -1, 0, 0 ).
yes
```

Here we use output/1 to select this file for output, write a message to the file, and then reset output to the console window ("0"):

```
?- output( foo ), write( `prolog` ), nl, output( 0 ).
```

This time using input/1 and fread/4, the data in the file is read back in:

```
?- input( foo ), fread( a, 0, 0, A ), input( 0 ).
A = 'prolog-M~J'
```

The following call closes the disk file once more, this time creating and initialising a 1024-byte memory file for read/write access in UTF-16LE format:

```
?- fcreate( foo, [], -2, 1024, -2 ).
yes
```
The resulting memory file can be used for input and output just like any disk file, except that it has no actual disk file associated with it:

```
?- output( foo ), write( `prolog.` ), nl, output( 0 ).
yes

?- input( foo ), inpos( 6 ), read( T ), input( 0 ).
T = log
```

**Notes**

When used to open disk files, `fcreate/5` sets their input and/or output pointers to the top (beginning) of the file when read-only or read/write permission is requested, but to the end of the file if write-only mode is requested, because such files are most commonly open for extension (appending), rather than overwriting. If it is desired to rewrite a file from the beginning, a call should be made to `outpos/1` to reset the pointer:

```
?- output( foo ), outpos( 0 ), output( 0 ).
yes
```

When used with memory files, which cannot expand beyond the specified size, input and/or output pointers are always initially set to the top of file.

When used with disk files, the `Size` parameter refers to the amount of memory used to buffer the file's data. If given as zero, **WIN-PROLOG** uses a default setting of 64kb (65536 bytes) for the buffer. Any size can be specified: in general, for largely sequential access, the 64kb default is ideal, even though larger settings can sometimes make file processing slightly faster. In random-access applications, such as when processing databases, a far smaller setting, preferably one approximating to the typical size of a "record", will yield an excellent response. At the other extreme, a buffer size which exceeds the size of the file will perform brilliantly in all applications, subject to there being sufficient memory available.

With disk files, the choice of buffer size is simply an efficiency consideration: in general, the default of 64kb (specified by a size of zero (0)) is fine. It has no bearing whatsoever on the maximum size the file may attain. With memory files, the specified size is a fixed upper limit beyond which input and output is not possible: this is enforced to protect applications from the General Protection Faults (GPFs) that would occur if memory bounds were inadvertently exceeded.

Certain Windows API functions, accessed via `winapi/4`, return memory addresses (also called pointers), and any such address can be defined as a file simply by using it as the `File` parameter:

```
?- winapi( ..., ..., ..., Addr ), fcreate( myfile, Addr, 2, 1024, 0 ).
```
When the "create" modes (-1 and -2) are used with memory files, the specified memory areas are initialised to zero; when used with the "open" modes (0, 1 and 2), memory is not initialised. If using an address returned by \textit{winapi/4}, it is generally best to use mode 0 (open read-only) or possibly mode 2 (open read/write) if any attempt is going to be made to read the data stored at the given address.

The \textit{fclose/1} predicate is used to close disk or memory files created by \textit{fcreate/5}; the name it uses is that supplied as a first argument handle to the latter, and this need not be related in any other way to the actual file name: please see the entry for \textit{fcreate/5} for further information.

While \textbf{WIN-PROLOG} handles arbitrary Unicode data sources automatically and transparently, through its \textit{fcreate/5} and \textit{open/3} predicates, using any of the encodings listed above, there are still times when it may be necessary to perform explicit conversions. One such example might be when implementing Windows Sockets (Winsock) code, such as a web server or agent, where data may be transmitted in UTF-8 or any other of the standard Unicode formats. A string conversion predicate, \textit{strutf/3}, provides a way to interpret such data correctly without having to write it to a file that had been opened in 8-bit ISO/IEC 8859-1 format, and then reopen the file in the appropriate Unicode encoding, before reading the data back, or to reverse the process in when preparing data to send back across the network.

This is one of a series of file handling and operating system interface predicates which are implemented directly in the \textbf{WIN-PROLOG} "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for \textit{absolute_file_name/2} for further discussion about file names.
fdata/5

retrieve information about an open disk or memory file

\[ \text{fdata( Name, File, Mode, Size, Code )} \]

+Name  \text{<atom>}
-File  \text{<variable>}
-Mode  \text{<variable>}
-Size  \text{<variable>}
-Code  \text{<variable>}

**Comments**

This returns the disk or memory *File* name or address, access *Mode*, buffer *Size* and character encoding *Code* for the given *Name*. The file *Mode* defines its access permissions, as defined in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>file is open in READ-ONLY mode</td>
</tr>
<tr>
<td>1</td>
<td>file is open in WRITE-ONLY mode</td>
</tr>
<tr>
<td>2</td>
<td>file is open in READ/WRITE mode</td>
</tr>
</tbody>
</table>

See fcreate/5 for a list of character encoding *Code* values and Appendix L for more information about character sets, encodings and formats.

**Examples**

The following call creates a new disk file called "c:\foo.txt", with write-only permissions, giving it the Prolog name "foo":

```
? - fcreate( foo, 'c:\foo.txt', -1, 0, 0 ).
yes
```

Now we'll check on the file's details to find its name, access mode and buffer size:

```
N = 'c:\foo.txt' ,
M = 1  ,
S = 65536  ,
C = 0
```
Notes

Notice that the "-1" in the initial call to fcreate/5 has been returned as "1" by fdata/5: the sign of the Mode argument is used by the former predicate simply to determine whether to create a new file or open an existing one. Note also that the returned buffer Size is 65536, the default used by fcreate/5 when given a zero (0) as its fourth argument.

Now let's create a memory file using the UTF-8 character encoding, and compare the information that is returned with that from the above:

```prolog
?- fcreate(foo, [], -2, 1024, 1).
yes

?- fdata(foo, N, M, S, C).
N = (-1813834816, 9031),
M = 2,
S = 1024,
C = 1
```

Once again, the mode argument has been returned as a positive value, but this time the buffer size is smaller than the default 64kb, because it was specified in the call to fcreate/5. More significant, though, is the address value returned in place of the file name. This comprises a pair of integers, the first being a linear offset (treated as an unsigned integer by Windows), and the second is a segment selector. Under Windows, the latter integer will always coincide with WIN-PROLOG's data segment; the former integer can be treated as a pointer, perhaps to be passed to a Windows API function or external DLL with the help of winapi/4. Note, however, that winapi/4 automatically supplies this address when a memory file name is passed in its argument list, so this address is not usually required by WIN-PROLOG programs.
fdict/2

return a list of currently open disk and memory files

\[ \text{fdict}( \text{Flag}, \text{Dict} ) \]

+Flag <integer> in the domain \{-1, 0, 1\}
-Dict <variable>

Comments

This returns a list of all the currently open disk and memory files, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples

The following call creates a new disk file called "c:\foo.txt", with write-only permissions, giving it the Prolog name "foo":

\[ \text{?- fcreate( foo, 'c:\foo.txt', -1, 0, 0 ).} \]
no

Now we'll check for a list of all currently open files (assuming that "foo" is the only file created so far):

\[ \text{?- fdict( 0, D ).} \]
D = [foo]

The following call can be used to close all files:

\[ \text{?- fdict( 0, D ), forall( member(F,D), fclose(F) ).} \]
D = [foo]
F = _

Notes

The names returned by fdict/2 are those used as the first argument to calls to fcreate/5: they need not relate in any other way to the physical names of disk files; indeed, memory files have no name apart from the one assigned by the latter predicate. In order to determine the physical file name for a disk file, a call can be made to fdata/5: this predicate can also be used to distinguish between disk and memory files.

This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
The predicate `file/1` is used to test whether the given term is a potential file name.

**Syntax**

```
file( Term )
```

**Comments**

This predicate succeeds if the given `Term` is a "file name", which means it is either Prolog atom or a tuple whose head is an atom and whose single argument is a file name.

**Examples**

The following calls test various cases:

```
?- file( 123 ).
   no

?- file( foo(bar) ).
   yes

?- file( foo(123) ).
   no

?- file( foo ).
   yes

?- file( [] ).
   no

?- file( `bar` ).
   no

?- file( X ).
   no
```

**Notes**

This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.
The success of this predicate only indicates the syntactic legality of a file name: it does not confirm the existence or otherwise of such a file, nor even whether it can successfully be mapped onto an absolute file name (see `file_search_path/2` and `absolute_file_name/2`). Please note that the special "empty list" atom, 
\[
\text{[]} \n\]
, is not a true atom in **WIN-PROLOG**, and therefore cannot be used as a component of a file name.
**File/3**

*return information about a file's attributes*

```prolog
file( File, Flag, Info )
```

- **File** <atom>
- **Flag** <integer> in the range [-1..6]
- **Info** <variable>

**Comments**

This is used to return information about a given *File*. Where **Flag** is set to one of the following values, the appropriate piece of information is bound to **Info**:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Type</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>&lt;integer&gt;</td>
<td>file existence: a return value of &quot;0&quot; indicates file does not exist, and &quot;1&quot; that it does</td>
</tr>
<tr>
<td>0</td>
<td>&lt;integer&gt;</td>
<td>file attributes</td>
</tr>
<tr>
<td>1</td>
<td>&lt;datetime&gt;</td>
<td>file creation time: note that under Windows, this is the same as last write time</td>
</tr>
<tr>
<td>2</td>
<td>&lt;datetime&gt;</td>
<td>file last access time: note that under Windows, this is the same as last write time, and than under Win9x it is only accurate to the nearest day (if enabled at all)</td>
</tr>
<tr>
<td>3</td>
<td>&lt;datetime&gt;</td>
<td>file last write time: note that under Windows, this is the only time that is maintained</td>
</tr>
<tr>
<td>4</td>
<td>&lt;integer&gt; or &lt;float&gt;</td>
<td>file size: note that from WIN-PROLOG 6.000, file sizes of up to just over 9 petabytes are directly supported</td>
</tr>
<tr>
<td>5</td>
<td>&lt;atom&gt;</td>
<td>file long (normal) name</td>
</tr>
<tr>
<td>6</td>
<td>&lt;atom&gt;</td>
<td>file short (DOS) name</td>
</tr>
</tbody>
</table>

**Examples**

The following call returns a <datetime> structure containing the creation time of the WIN-PROLOG kernel, "pro386w.exe" (this example assumes that the file's date is 01-APR-1999, and its time is 10:08:42):

```prolog
?- file( 'pro386w.exe', 1, D ).
D = (145822,36522000)
```

WIN-PROLOG 7.0 - Technical Reference
The `file/3` predicate provides detailed access to the information that Windows stores for any given file, and in conjunction with `dir/3` can be used to implement powerful file management functions. File information can only be tested by this predicate: in order to set a file time stamp, for example, `stamp/2` must be used (this sets all three timestamps concurrently), while the `attrib/2` predicate can be used to set certain file attributes and files can be renamed with `ren/2`. Please note that in all cases except where `Flag` is set to "-1" (file existence check), an error will be generated if `File` does not name an existing file.

This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for `absolute_file_name/2` for further discussion about file names.

This is also one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date_time" data type provides a uniform structure for handling, processing and storing dates and times throughout WIN-PROLOG. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, `time/4` and `time/5`, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, `time/2`, is used to return current local or elapsed time.

Starting with WIN-PROLOG 6.000, file sizes are no longer limited to $2^{32}-1$ bytes in size: "Fat" files of up to $2^{53}-1$ bytes can be created, and randomly positioned using `inpos/1` and `outpos/1` for reading and writing respectively. This number equates to 9,007,199,254,740,991 bytes, or just over 9 petabytes (9000 terabytes) in size, and is the current limit since this is the biggest whole number than can be stored and computed with integer precision in a 64-bit floating point number.
**file_search_path/2**

*dynamic predicate defining logical file names*

```prolog
file_search_path( Name, Path )

?Name       <path_alias>
?Path       <file_spec>
```

**Comments**
This dynamic (and therefore user-modifiable) predicate contains a series of clauses, each of which relates a `Name` to a logical file `Path`. These names and the paths to which they refer are used by `absolute_file_name/2` and `absolute_file_name/3` to expand logical file names into absolute ones.

**Examples**
You can see the default definition of `file_search_path/2` simply by listing it with `listing/1`:

```prolog
?- listing( file_search_path ).
```

```prolog
% file_search_path/2

file_search_path( system, prolog('system\') ).

file_search_path( examples, prolog('examples\') ).

file_search_path( files, prolog('files\') ).

file_search_path( library, A ) :-
    library_directory( A ).

file_search_path( temp, A ) :-
    ( env( B ),
      member( ('TEMP',C), B )
    -> stratm( C, A )
    ;   A = prolog(\.)
    ).

yes
```
Notes

Programs do not normally call `file_search_path/2` directly; rather, its definition is used indirectly when `logical file names` are used, either in calls to `absolute_file_name/2` and `absolute_file_name/3`, or to any of the file handling predicates which in turn call these two predicates. By adding your own clauses to `file_search_path/2`, you can extend the "vocabulary" used by all such predicates. Please see the entries for `absolute_file_name/2` and `absolute_file_name/3` for further information about logical and `absolute file names`. 
**fileerrors/0**

enable the reporting of file error messages

**fileerrors**

**Comments**
This sets the `fileerrors` flag to "on", so that an error is generated by predicates such as `see/1`, `tell/1` and `open/2` if a specified file cannot be opened.

**Examples**
The following pair of calls set the fileerrors flag, and then attempts to "see" a nonexistent file, resulting in an error:

```prolog
?- fileerrors.
yes

?- see( foo ).  <enter>
! ----------------------------------------
! Error 31 : File Not Found
! Goal     : see(foo)
```

**Notes**
The fileerrors flag is set by default, so that, like the above example, errors normally occur when an attempt is made to open a nonexistent or prohibited file. The `fileerrors/0` predicate has an opposite, `nofileerrors/0`, which is used to clear the fileerrors flag. This, and many other flags, can also be managed via the `prolog_flag/2` and `prolog_flag/3` predicates.
find/3

find or copy up to a string in the current input stream

\[ \text{find( Find, Mode, Found )} \]

+Find <string>
+Mode <integer> in the range [0..3]
-Found <variable>

Comments

This searches the current input stream for the given \textit{Find} string, either ignoring or exactly matching the case and optionally sending mismatched text to the current output stream, according to the given \textit{Mode}. The result of the find operation is returned as a string in \textit{Found}. Mode may have any of the following settings:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Match exact case, perform no output</td>
</tr>
<tr>
<td>1</td>
<td>Ignore case, perform no output</td>
</tr>
<tr>
<td>2</td>
<td>Match exact case, copy mismatch to output</td>
</tr>
<tr>
<td>3</td>
<td>Ignore case, copy mismatch to output</td>
</tr>
</tbody>
</table>

If a match is found in the input stream, \textit{Found} is bound to a string representing the exact text of that match. In modes 1 and 3, this may contain letters of different cases to the original \textit{Find} string. If no match is found, an empty string (``) is returned. If \textit{Find} is specified as an empty string (``), \textit{find/3} will search for the first non-white-space character in the current input stream, returning it (if found) as a single-character string.

Examples

Assume that a file called "foo" exists in the current directory, containing the text:

\[ \text{the quick brown fox jumps over a lazy dog} \]

This file can be opened for reading by the call:

?\- fcreate( i, foo, 0, 0, 0 ). <enter>

yes

Now we will use \textit{find/3} to search for the word "fox", returning a string that indicates a match:
?- input(i), find(`fox`, 0, F), input(0).
F = `Fox`

All characters in the file, up to and including those matched by the Find string, are treated as having been read, as can be shown by the following call to inpos/1:

?- input(i), inpos(P), input(0).
P = 19

Using modes 2 or 3, find/3 can output mismatched characters along the way, as shown by the next example:

?- input(i), find(`LaZy`, 3, F), nl, input(0).
jumps over a
F = `lazy`

Here, all characters have been output, from the end of "fox" and up to, but not including the beginning of "lazy". Because the chosen Mode (3) also includes case insensitivity, the lowercase word "lazy" was found, even though the Find string was a mix of cases, "LaZy". Now we will search for the next non-white-space character:

?- input(i), find(``, 0, F), input(0).
F = `d`

The beginning character of "dog" has been found and returned, but not consumed: as the Find string was of zero length, zero characters are consumed when the next non-white-space token is found. Finally, let's search for something that is not in the file, outputting as we go:

?- input(i), find(`cat`, 2, F), nl, input(0).
dog
F = ``

The remainder of the file, in this case the word "dog", is output, but the Found string returns as an empty string (``). This string can only occur when the item being searched for is not present on the input stream.

**Notes**

The find/3 predicate is very versatile, effectively performing several functions. At its simplest, it can search for a string in a file or another string, or strip unwanted white space prior to a token on input. Thanks to its output modes, it can also be used to copy from current input to current output up to a specified termination string. This is especially useful when writing search/replace algorithms, for example as follows:
replace( From, To, Input, Output ) :-
    replace( From, To ) <~ Input ~> Output.

replace( From, To ) :-
    find( From, 2, Found ),
    Found = ` `, !.

replace( From, To ) :-
    write( To ),
    replace( From, To ).

Because the call to find/3 uses an output mode (2), all non-matched characters are sent directly to current output. If the Found string is empty (` `), the input stream has been fully processed; otherwise, the "To" string is sent and a recursive call is made. Thanks to WIN-PROLOG's powerful string input and output, this simple program can even perform substring replacements, as shown in this call:

?- replace( `dog`, `cat`, `dogs are dogmatic`, X ).
   X = `cats are catmatic`

Because the Found string returns a match in its original case, it is easy to write intelligent search/replace procedures that preserve the mix of cases in the substituted text.

There are two text search predicates in WIN-PROLOG; find/3 and scan/3: the former searches for a match for a single specified string, while the latter scans for the first, best instance of one string in a given list. Where only one string is the target of the search, find/3 provides a slightly faster solution; additionally, it has a special feature which allows for fast scanning for non-white text, but in all other respects, scan/3 provides a considerably more flexible solution to text searching applications.
findall/3

return unsorted list of terms in solutions of a given goal

\[\text{findall( Term, Goal, List )}\]

Comments
This predicate returns an unsorted list of solutions that would normally be obtainable only by failing and backtracking through a query. It succeeds if \(\text{List}\) can be unified to a list of all instances of \(\text{Term}\) such that \(\text{Goal}\) is true. The \(\text{Term}\) may be any Prolog term, and \(\text{Goal}\) may be any Prolog goal, with or without calls to the existential quantifier predicate, \(^/2\), which is ignored in the present predicate (see "notes" below).

Examples
Consider the database relation:

\[
\begin{align*}
\text{foo( 1, c ).} \\
\text{foo( 2, a ).} \\
\text{foo( 1, b ).} \\
\text{foo( 2, a ).}
\end{align*}
\]

The following call returns a list of all solutions of "X" in the goal "foo(Y,X)", without partitioning on different values of "Y":

\[\begin{align*}
?\text{- findall( X, foo(Y,X), Z ).} \\
X = _ , \\
Y = _ , \\
Z = [c,a,b,a]
\end{align*}\]

Notes
Unlike the bagof/3 and setof/3 predicates, findall/3 does not perform any partitioning of results. It therefore has no concept of "existential quantification", and as such treats any calls to \(^/2\) just as if the appropriate goal were being called directly. For further information about existential quantification, see the entries for bagof/3 and setof/3.
flag/1

get or set the Windows message flag

\[ \text{flag( Flag )} \]

?Flag \quad \text{<variable> or <integer> in the domain \{-1,0,1\}}

Comments

This gets or sets the state of the Windows message \text{Flag}. When clear, this flag suppresses the acceptance of messages to Prolog programs, disabling all user-defined message, menu or dialog handlers; when set, such messages are accepted and passed to any system- or user-defined handlers. The flag may have either of the following states:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Message loop disabled, Prolog does not respond to Windows messages or timer events</td>
</tr>
<tr>
<td>0</td>
<td>Message loop enabled, Prolog queues Windows messages and dispatches timer events</td>
</tr>
<tr>
<td>1</td>
<td>Message loop enabled, Prolog dispatches Windows messages and timer events</td>
</tr>
</tbody>
</table>

Examples

The following call picks the current value of the Windows message flag, then clears the flag (sets it to "0"), before calling some program "foo/0". Upon completion, it resets the flag to its original value:

\[ \text{?- flag( F ), flag( 0 ), foo, flag( F ).} \quad \text{<enter> F = 1} \]

Notes

Initially, the Windows message flag is set (has the value "1"): in this state it allows any Windows message to interrupt the current Prolog goal and call the system "message hook" (see \text{message_hook/4} and \text{'?MESSAGE?'/4}). When a message arrives, the flag is automatically cleared (set to "0"), thereby preventing any further messages from interrupting the current one. The final job that \text{'?MESSAGE?'/4} should perform is to set the message flag back to "1", and then call the interrupted goal:

\[ \text{'?MESSAGE?(' Window, Number, Data, Goal ) :-} \]
\[ \quad \ldots \]
\[ \quad \text{flag( 1 ),} \]
\[ \quad \text{Goal.} \]

If \text{message_hook/4} is called, this attempts to call a suitable window handler or similar, before finally performing the last two steps above.
float/1

`test whether a term is a floating point number`

```prolog
float( Term )
```

Term term

**Comments**  
This succeeds if `Term` is a floating point number.

**Examples**  
The following calls test various cases:

```prolog
?- float( 123 ).
no

?- float( 123.45 ).
yes

?- float( 123.00 ).
no

?- float( 9876543210 ).
yes

?- float( X ).
no
```

**Notes**  
The arithmetic evaluator in **WIN-PROLOG** treats all numbers simply as such: numbers. It automatically converts integers to floats when required, and always attempts to represent the final result as an integer, for example:

```prolog
?- X is 67.5 * 1.2.
X = 81
```

Notice that the result, "81", is an integer, and not a float. Conversely, it is possible to generate floats when computing with integers:

```prolog
?- X is 22 / 7.
```
\[
X = 3.14285714285714
\]

Except when calling built-in predicates, most of which require true integers as their numeric inputs, there is no need to distinguish between integers and floats in WIN-PROLOG.
decompress data from current input to current output

\[\text{fluff}( \text{Size}, \text{Raw}, \text{Comp} )\]

- Size \(<\text{variable}>\)
- Raw \(<\text{variable}>\)
- Comp \(<\text{variable}>\)

**Comments**
This copies data from the current compressed input stream to the current output stream, decompressing it along the way. It terminates when the end of a compressed input record is encountered. The window size, raw and compressed character counts are returned in \(\text{Size}, \text{Raw}\) and \(\text{Comp}\) respectively.

**Examples**
The following simple programs can be used to create or unpack compressed archives of multiple files:

\[
\text{stuff_archive}( \text{Archive}, \text{Files} ) :-
\text{open}( \text{Archive}, \text{write} ),
\text{tell}( \text{Archive} ),
\forall ( \text{member}(\text{File}, \text{Files} ),
( \text{writeq}(\text{File} ),
\text{write}(\text{'. '}),
\text{see}(\text{File} ),
\text{stuff}( -1, 2, _, _ ),
\text{seen}
),
\text{told}.
\]

\[
\text{fluff_archive}( \text{Archive} ) :-
\text{see}( \text{Archive} ),
\text{repeat},
\text{read}(\text{File} ),
\text{tell}(\text{File} ),
\text{fluff}( _, _, _ ),
\text{told},
\]

Once compiled, this program can be used to create a stuffed (compressed) archive of multiple files; the following call, for example, will create an archive called "foo" which includes compressed copies of two other files, "bar" and "sux":

?- stuff_archive( foo, [bar,sux] ).  
yes

These files can be recovered from the archive on a subsequent occasion with the call:

?- fluff_archive( foo ).  
yes

Notes

Because fluff/3 reads a single stuffed record, rather than an entire file, more than one compressed sequence can be stuffed into a single file. The LZSS algorithm, as used to compress and decompress data in the stuff/4 and fluff/3 predicates respectively, uses a "sliding window" to maintain a memory of recently processed data, while a "look-ahead" 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 optimum setting for whichever type of data you want to compress. The possible values of Size and their meanings are listed in the following table:

<table>
<thead>
<tr>
<th>Size</th>
<th>Bit Settings</th>
<th>Window</th>
<th>Look-ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9/7</td>
<td>512</td>
<td>128</td>
</tr>
<tr>
<td>1</td>
<td>10/6</td>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>11/5</td>
<td>2048</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>12/4</td>
<td>4096</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>13/3</td>
<td>8192</td>
<td>8</td>
</tr>
</tbody>
</table>

For highly repetitive files, such as Windows NT logfiles, the smallest window (Size=“0”) will probably give the best compression, with a theoretical maximum ratio of 64:1; this relies on repeating elements in the file being within 512 bytes of each other. For sparsely repetitive files, lists of names and addresses, a larger window will help; with the largest window (Size=“4”), repeating elements can be as far as 8192 bytes apart, although the maximum compression ratio is reduced to 4:1. For most data streams, a Size of 2 or 3 will give the best compression.
flush/0

flush the current output stream

flush

Comments

This output predicate is used to flush remaining buffered characters in the output stream, and is used mainly in console output routines to ensure that printed information is refreshed to the console window.

Examples

The following call performs writes to the console, before outputting a new line and pausing for 5 seconds; the call to flush/0 between the output of "hello" and the output of "world", ensures that the former is displayed immediately; the latter (and the new line) will only be displayed when the 5 second delay is up, and WIN-PROLOG is waiting for console input:

?- write( hello ), flush, write( world ), nl, pause( 5000 ).
   helloworld

Notes

Versions of WIN-PROLOG before 5.000 contained a predicate called flush/0, but this was a misnomer for a feature designed to clear current input. From WIN-PROLOG 5.000 onwards, a new predicate, empty/0, performs the expected flushing of buffered input.
**fname/4**

split a file into its path, name and extension

```
fname( File, Path, Name, Extension )
```

+File  <atom>
-Path  <variable>
-Name  <variable>
-Extension  <variable>

**Comments**

This splits a given `File` name into its `Path`, `Name` and `Extension` components. The `Path` is defined as all characters up to and including the final slash, backslash or colon symbol ("/", ",", or ":"), and the `Extension` is defined as all characters after and including the final period ("."), after the `Path`. The file `Name` is the sequence of characters following the `Path` and preceding the `Extension`. As a special case, if `File` is specified as the empty list, this returns the `Path`, `Name` and `Extension` of the currently executing instance of **WIN-PROLOG**.

**Examples**

The following call demonstrates how a file name is parsed and split by this predicate:

```
?- fname( 'c:\foo\bar.sux', P, N, E ).
P = 'c:\foo\',
N = bar ,
E = '.sux'
```

The following call demonstrates how to retrieve the full name of the currently executing instance of **WIN-PROLOG**:

```
?- fname( [], P, N, E ).
P = 'C:\PRO386W\',
N = 'PRO386W' ,
E = '.EXE'
```

**Notes**

Except when retrieving the name of the current instance of **WIN-PROLOG**, `fname/4` is not a file handling predicate as such, but rather a text manipulation predicate. It makes no attempt to check that the given `File` exists or even that its syntax is legal. Its purpose is to enable programs to split file names obtained from the user or other sources, and then attach other extensions or paths with the help of the `cat/3` predicate, for example as in the following program:

```
get_backup_name( File, Back ) :-
```
fname( File, Path, Name, _ ),
cat( [Path,Name,'.bak'], Back, _ ).

Once compiled, this program can quickly and efficiently generate the "bak" (backup) version of a file name:

?- get_backup_name( 'c:\foo\bar.sux', B ).
B = 'c:\foo\bar.bak'

Similarly, the following program can be used to generate the name of a "INI" file that resides with, and shares its filename with, the current instance of WIN-PROLOG:

get_init_name( Init ) :-
  fname( [], Path, Name, _ ),
  cat( [Path,Name,'.INI'], Init, _ ).

Once compiled, and assuming that WIN-PROLOG is installed in "C:\PRO386W", then this program will generate the following output:

?- get_init_name( I ).
I = 'C:\PRO386W\PRO386W.INI'

Note that any slashes ("/") in the given File name are replaced with backslashes ("\") in the returned Path.
**fntbox/3**

*display the "choose font" dialog box*

```
fntbox( Title, Initial, Final )
```

**+Title**  
<term>

**+Initial**  
<list_of(<atom>,<integer>,<integer>)>

**?Final**  
<variable> or <list>

**Comments**  
This displays a standard system "font box" (or "choose font") dialog box, with the given Title and Initial font, allowing the user to browse and select from the available fonts. The dialog closes when the "OK" button is clicked or if <enter> is pressed, and the user's chosen font is unified with Final. The Initial font must be specified as a list of the following form:

```
[Face,Size,Style]
```

This initialises the dialog with a font created with the given Face, point Size and Style; the latter is chosen from this table:

<table>
<thead>
<tr>
<th>Style</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal</td>
</tr>
<tr>
<td>1</td>
<td>italic</td>
</tr>
<tr>
<td>2</td>
<td>bold</td>
</tr>
<tr>
<td>3</td>
<td>bolditalic</td>
</tr>
</tbody>
</table>

The user's chosen font will be represented by a similar structure, which is unified with Final.

**Examples**  
The following command displays the font box, using the title, "choose(a,font)", and initialising it with Arial (normal) at 24pt; after the user has browsed the available fonts and selected one, the final choice is returned (in this example, Times New Roman Italic at 36pt):

```
?- fntbox( choose(a,font), [arial,24,0], F ).
F = ['Times New Roman',36,1]
```

**Notes**  
The fntbox/3 predicate provides a convenient method for allowing users to chose a font for use in an application; it does not permanently create the chosen font, but simply returns its specification as a list of three elements returned which may then be passed to wfccreate/4 to create the font itself. Unlike certain other programming environment dialogs, such as those accessed through prnbox/4, opnbox/5 and
savbox5, the present predicate is not a true "common dialog", being implemented directly in terms of WIN-PROLOG’s window handling predicates.

When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window (see the above example) or graphics device context. Please note that fonts created with wfcreate/4 are in fact identical to those created by gfx_font_create/4: the wf/*n window-oriented predicates are completely interchangeable with the newer, gfx_font*/n "Grafix" predicates.
fnthdl/2

convert between a font and its handle

\[ \text{fnthdl( Font, Handle )} \]

?Font          <object_handle> or <variable>
?Handle        <integer> or <variable>

Comments  This converts between a font and its raw (integer) handle. The \text{Font} parameter may be a named font, a handle of the form, "font(Raw)", where "Raw" is the raw handle.

Examples  The following call will return the raw handle for a font called "foo", assuming that such a font has previously been created:

\[
\text{?- fnthdl( foo, H ).} \quad <\text{enter}> \\
\text{H = 65535}
\]

Notes  This is one of a family of \textsc{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \text{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, and its helper predicate, \text{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
fonts/1

return a list of available typeface names

    fonts( Fonts )

    -Fonts <variable>

Comments

This binds Fonts to a list of atoms representing the names of all the typefaces available in the operating system.

Examples

The following call returns the list of typefaces present on a particular Windows system (please note that this list has been abbreviated in the example below):

    |- fonts( F ).
    F = ['AmericanTypewriter Cond', ... , 'Tamburo']

Notes

Strictly speaking, the fonts/1 predicate is misnamed, because it returns a list of typeface names, rather than fonts: a font is a specific instance of a typeface, rendered at a specific size and style (bold, italic, or whatever). In the Windows world, however, typefaces are usually described, incorrectly, as fonts.
forall/2

test that a given goal is true for all cases of another goal

forall( Goal1, Goal2 )

+Goal1   <goal>
+Goal2   <goal>

Comments
This succeeds if for all solutions of Goal1, Goal2 is also true.

Examples
The following call uses member/2 and atom/2 to check that all members of the given list are atoms:

?- forall( member(X,[a,b,c]), atom(X) ).
X = _

Conversely, the following call fails because one of the list elements is an integer, rather than an atom:

?- forall( member(X,[a,b,3]), atom(X) ).
no

Notes
Although forall/2 is usually described as a logical test, as shown in the two examples above, its main use is as a loop construct in procedural programming. Consider the program:

    nameof( 1, one ).
    nameof( 2, two ).
    nameof( 3, three ).
    nameof( 4, four ).
    nameof( 5, five ).
    nameof( 6, six ).
    nameof( 7, seven ).
    nameof( 8, eight ).
    nameof( 9, nine ).
    nameof( 10, ten ).

    evens :-
forall( (  nameof( Number, Name ),
  0 is Number mod 2 ),
  (  write( Name ),
      write( ` is even` ),
      nl
  )
).

This comprises a data relation, nameof/2, which lists the values and names of the integers 1..10, and a program, evens/0, which displays the even ones. It simply says that "for all instances of "nameof( Number, Name )", where "Number mod 2" is zero, write out a messages stating that the "Name" is even. Once compiled, here's how it looks:

?- evens.
  two is even
  four is even
  six is even
  eight is even
  ten is even
  yes

Although using exactly the same mechanism as the previous examples, this use of forall/2 is more akin to: "forall cases of Goal1, execute Goal2". Such a reading is often referred to as the "procedural reading": conversely, the description under "comments" above is usually called the "logical reading".

As a footnote, it might be interesting to remark that forall/2 is implemented through a simple "double negation":

forall( Goal1, Goal2 ) :-
  \+ ( Goal1,
      \+ Goal2
  ).

The body of this program can be read literally as, "there is not an instance of Goal1 for which there is not also an instance of Goal2": in other words, for all instances of Goal1, there must also be at least one instance of Goal2.
**force/1**

*call a goal for which a spypoint is currently set*

```prolog
force( Goal )
```

**Comments**

This calls the given Goal, bypassing the spypoint mechanism, and succeeds or fails depending upon whether the Goal itself succeeds or fails.

**Examples**

The following command invokes the `member/3` predicate from within `force/1`:

```prolog
?- force( member(X,[1,2,3]) ).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

**Notes**

In nearly every respect, this predicate is identical to `call/1`: the only difference occurs when the Goal being invoked is a predicate for which a spypoint has been set (see `spy/1`), and while debugging mode is operational (see `debug/1`). Normally, calling such a goal causes a spypoint to fire, invoking the currently selected debugger.

By calling the Goal via `force/1`, the debugger can be bypassed. Note that this is not the same as turning debugging mode off, or even removing the spypoint from the called predicate: with `force/1`, spypoints are only disabled momentarily while the precise call is set up. Any spied calls during the execution of Goal, including recursive calls to the initial predicate itself, will still invoke the debugger, unless, of course, these calls are themselves made through the `force/1` predicate.
**compute a reverse polish notation floating point expression**

```prolog
fpn( Expression, Result )
+Expression <list>
?Result <number> or <variable>
```

**Comments**
This evaluates the given reverse polish notation (RPN) arithmetic *Expression* and unifies the solution with *Result*. The *Expression* comprises a list of numbers and functions, which are processed from left to right in order to compute the result. The *Expression* list may contain anything from a single number to large sequence of numbers and functions.

The functions can be split into three groups: binary general number functions, unary general number functions, and binary integer bit manipulation functions. Binary functions are simply those that take two arguments, while unary functions take just one.

The following table shows the general binary functions:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, +</td>
<td>adds X to Y</td>
</tr>
<tr>
<td>X, Y, -</td>
<td>subtracts Y from X</td>
</tr>
<tr>
<td>X, Y, *</td>
<td>multiplies X by Y</td>
</tr>
<tr>
<td>X, Y, /</td>
<td>divides X by Y</td>
</tr>
<tr>
<td>X, Y, //</td>
<td>performs integer division of X by Y, truncating the result towards zero</td>
</tr>
<tr>
<td>X, Y, \</td>
<td>computes X modulo Y, where the result has the same sign as Y</td>
</tr>
<tr>
<td>X, Y, ^</td>
<td>raises X to the power of Y</td>
</tr>
<tr>
<td>X, Y, max</td>
<td>computes the maximum of X and Y (the nearest to +infinity)</td>
</tr>
<tr>
<td>X, Y, min</td>
<td>computes the minimum of X and Y (the nearest to -infinity)</td>
</tr>
<tr>
<td>X, Y, mod</td>
<td>computes X modulo Y, where the result has the same sign as Y</td>
</tr>
</tbody>
</table>

The following table shows the general unary functions:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>X, ?</td>
<td>computes a linear congruential pseudo random floating point number between zero and X</td>
</tr>
<tr>
<td>X, @</td>
<td>computes a Marsaglia Zaman pseudo random floating point number between zero and X</td>
</tr>
<tr>
<td>X, abs</td>
<td>computes the absolute value of X</td>
</tr>
<tr>
<td>X, acos</td>
<td>computes the arccosine of X (degrees)</td>
</tr>
<tr>
<td>X, aln</td>
<td>computes the natural antilogarithm of X</td>
</tr>
<tr>
<td>X, alog</td>
<td>computes the common antilogarithm of X</td>
</tr>
<tr>
<td>X, asin</td>
<td>computes the arcsine of X (degrees)</td>
</tr>
<tr>
<td>X, atan</td>
<td>computes the arctangent of X (degrees)</td>
</tr>
<tr>
<td>X, cos</td>
<td>computes the cosine of X (degrees)</td>
</tr>
<tr>
<td>X, fp</td>
<td>computes the fractional part of X (this has the same sign as X)</td>
</tr>
<tr>
<td>X, int</td>
<td>computes the nearest integer less than or equal to X (truncates towards -infinity)</td>
</tr>
<tr>
<td>X, ip</td>
<td>computes the integer part of X (truncated towards zero; this has the same sign as X)</td>
</tr>
<tr>
<td>X, ln</td>
<td>computes the natural logarithm of X</td>
</tr>
<tr>
<td>X, log</td>
<td>computes the common logarithm of X</td>
</tr>
<tr>
<td>X, pi</td>
<td>computes the value of X multiplied by the mathematical constant, pi</td>
</tr>
<tr>
<td>X, rand</td>
<td>computes a linear congruential pseudo random floating point number between zero and X</td>
</tr>
<tr>
<td>X, sign</td>
<td>computes the sign of X (-1, 0, or 1 for negative, zero or positive respectively)</td>
</tr>
<tr>
<td>X, sin</td>
<td>computes the sine of X (degrees)</td>
</tr>
<tr>
<td>X, sq</td>
<td>computes the square of X</td>
</tr>
<tr>
<td>X, sqrt</td>
<td>computes the square root of X</td>
</tr>
<tr>
<td>X, tan</td>
<td>computes the tangent of X (degrees)</td>
</tr>
</tbody>
</table>

The following table shows the bit manipulation operators, all of which are specific to 32-bit (X86) or 64-bit (X64) signed integers, in the ranges \(-(2^{31}) \ldots (2^{31})-1\) and \(-(2^{63}) \ldots (2^{63})-1\) respectively:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>I, J, &amp;</td>
<td>computes the bitwise &quot;and&quot; I and J</td>
</tr>
<tr>
<td>I, J, V</td>
<td>computes the bitwise &quot;inclusive or&quot; I and J</td>
</tr>
<tr>
<td>I, J, &lt;&lt;</td>
<td>computes the arithmetic shift left of I by J bits (vacated bits are cleared to zero)</td>
</tr>
<tr>
<td>I, J, &gt;&gt;</td>
<td>computes the arithmetic shift right of I by J bits (msb is propagated into the vacated bits)</td>
</tr>
<tr>
<td>I, J, a</td>
<td>computes the bitwise &quot;and&quot; I and J (this is the same as &quot;I &amp; J&quot;)</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>I, J, l</td>
<td>computes the left rotation of the I by J bits</td>
</tr>
<tr>
<td>I, J, o</td>
<td>computes the bitwise &quot;inclusive or&quot; I and J (this is the same as &quot;I | J&quot;)</td>
</tr>
<tr>
<td>I, J, r</td>
<td>computes the right rotation of the I by J bits</td>
</tr>
<tr>
<td>I, J, x</td>
<td>computes the bitwise &quot;exclusive or&quot; I and J</td>
</tr>
</tbody>
</table>

### Examples

The following calls show a variety of expressions being evaluated or tested:

```prolog
?- fpn( [2,2,+], X ).
X = 4
<br>
?- fpn( [10,2,\], 5 ).
yes
<br>
?- fpn( [22,7,\], X ).
X = 3.142857142857143
<br>
?- fpn( [45,sin,60,cos,*,75,tan,\/,asin], X ).
X = 5.436029972077099
```

### Notes

The `fpn/2` predicate is the low-level floating point "maths engine" in **WIN-PROLOG**, and is used internally by the better known `is/2` predicate to perform the actual calculations. Indeed, `is/2` is implemented, in effect, as follows:

```prolog
is( Expr, Result ) :-
    convert_to_rpn( Expr, RPN ),
    fpn( RPN, Result ).
```

except that the whole process is taken care of in assembler code. In addition to the basic set of standard arithmetic operators, **WIN-PROLOG** includes a full set of "scientific calculator" functions, covering logarithms, linear trigonometry, truncation and rounding functions. For simulation work it also includes a carefully-researched pseudo random number generator (see `seed/1` for further information), and finally a set of bit-oriented integer manipulation operators.

The reverse polish notation model is very simple to understand, and entirely unambiguous. The given list is processed from left to right, and if a number is found, it is pushed onto an internal stack; if an atom is found, it is executed as a function. A binary function will attempt to pop two arguments from the internal stack, operate on them, and then push the result back on the stack. A unary function will pop
just one value off the stack, process it, and then push the result back on the stack. The computation succeeds when the full list has been processed, and exactly one result is left on the internal stack: this result is unified with the predicate's second argument.

Conventional "infix" maths expressions have two problems, one human, the other computational. Ask someone to calculate the following expression in their heads, or even with the aid of a basic calculator, and many a time you will be given the wrong answer:

\[ 2 + 3 \times 4 = ??? \]

A lot of people will give you the wrong answer, "20", because first they will compute, "2 + 3 = 5", and then "5 \times 4 = 20". The correct answer is, of course, "14", because the in arithmetic, you perform multiplications before additions: "3 \times 4 = 12, 12 + 2 = 14. Just as you get with Prolog:

```
?- X is 2 + 3 * 4
X = 14
```

The \( \text{is} /2 \) predicate itself has no knowledge of the rules of arithmetic: the above works only because "\(+\)" has been declared as an operator with a higher numerical precedence than "\(*\)" (500 yfx vs 400 yfx). If you were to redefine "\(+\)" as a lower precedence, with a call to \( \text{op}/3 \), then \( \text{is} /2 \) will appear to return the same, wrong result, as many ordinary folks do:

```
?- \text{op}( 400, \text{yfx}, + ).
yes
?- X is 2 + 3 * 4
X = 20
```

Again, it should be stressed that this is not a bug in \( \text{is} /2 \): it is simply processing a Prolog term that has been created by \( \text{read}/1 \); before the call to \( \text{op}/3 \), "2+3\(4\)" is read as the term, \((2+(3\times4))\); after the call, it is read as \((2+3)\times4\). The beauty of reverse polish notation, is that it is not only unambiguous, but it cannot be affected by operator definitions:

```
?- \text{fpn}( [2,3,4,*,+] , X ).
X = 14
?- \text{fpn}( [2,3,+,4,*] , X ).
X = 20
```

There are two reverse polish predicates in \text{WIN-PROLOG}: \text{fpn}/2 (for all numbers) and \text{rpn}/2 (for integers only).
fread/4

read a formatted term

\[
\text{fread( Format, Width, Modifier, Term )}
\]

+Format \ <atom> in the domain \{a,b,f,i,n,r,s\}.  
+Width \ <integer> in the range \{-8192..8192\}  
+Modifier \ <integer> in the range \{-8192..8192\}  
-Term \ <variable>

Comments

This reads a Term from the current input stream using the given Format, field Width and Modifier. The Format is a single-character atom, defining the type of Term to be read:

<table>
<thead>
<tr>
<th>Format</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>atom</td>
</tr>
<tr>
<td>c</td>
<td>char list</td>
</tr>
<tr>
<td>f</td>
<td>fixed point number</td>
</tr>
<tr>
<td>i</td>
<td>integer</td>
</tr>
<tr>
<td>n</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>r</td>
<td>arbitrary radix</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
</tbody>
</table>

The field Width may be any positive or negative number, specifying the number of characters to read from the input stream, or the special value zero (0), which allows "free format" input (see below). The Modifier is an integer which further defines the treatment of several formats (again, see below).

Examples

The following call reads an integer from the first five characters of the next line of input at the console. Note that a call to the flush/0 predicate has been added: this is to dispose of any excess characters ("67890" in this example):

\[
\text{?- fread( i, 5, 0, I ), flush.} <\text{enter>}
\]

| I     | 1234567890 <\text{enter>}
|-------|-----------------
| I     | 12345 <\text{enter>}

\[
\text{I = 12345}<\text{enter>}
\]
Here, the same five characters are input as an atom:

```prolog
?- fread( a, 5, 0, A ), flush. <enter>
|: 1234567890 <enter>
A = '12345'
```

Notes
As the above examples show, `fread/4` is a normal input predicate that can be used with any input stream. For the sake of clarity, however, all remaining examples in this section will use string input, via the `<~>/2` predicate. This approach allows us to examine the exact relationships between input data, the format specification, and the resulting term.

There are, broadly, three main groups of formats, respectively handling text, decimal number and arbitrary radix number input. Furthermore, there are two basic modes of formatted input: fixed and free. In this section, we will examine each group and each mode in turn.

The text group of formats comprises three separate cases which input text as atoms ("a"), char lists ("c") or strings ("s"). When used in fixed format mode, the Modifier is ignored, and the exact number of characters specified by Width is read into a text term of the appropriate type:

```prolog
?- fread( a, 10, 0, A ) <~ `hello-M-Jworld`. <enter>
A = 'hello-M-Jwor'

?- fread( b, 10, 0, B ) <~ `hello-M-Jworld`. <enter>

?- fread( s, 10, 0, S ) <~ `hello-M-Jworld`. <enter>
S = `hello-M-Jwor`
```

When used in free format mode (specified by a Width of zero), an entire line of input, up to and including the terminating "<cr><lf>" pair, is read. In this case, the sign of the Modifier (but not its value) determines whether the input should be post-processed: if the Modifier is negative, "<tab>" characters are expanded into spaces, the terminating "<cr><lf>" characters are discarded, and any other control characters are replaced by ordinary spaces:

```prolog
?- fread( a, 0, -1, A ) <~ `foo-Ibar-[sux-M-J`. <enter>
A = 'foo     bar sux'

?- fread( s, 0, -1, S ) <~ `foo-Ibar-[sux-M-J`. <enter>
B = [102,111,111,32,32,32,32,32,98,97,114,32,115,117,120]
```
?- fread( s, 0, -1, S ) <= `foo-Ibar-[sux-M-J`.
S = `foo     bar sux`

Where the Modifier is zero or positive, no post-processing is undertaken and input is returned literally:

?- fread( a, 0, 0, A ) <= `foo-Ibar-[sux-M-J`.
A = 'foo-Ibar-[sux-M-J`

?- fread( b, 0, 0, B ) <= `foo-Ibar-[sux-M-J`.
B = [102,111,111,9,98,97,114,27,115,117,120,13,10]

?- fread( s, 0, 0, S ) <= `foo-Ibar-[sux-M-J`.
S = `foo-Ibar-[sux-M-J`

The number group of formats provides input of decimal numbers as signed ("i") and unsigned ("n") integers, or as floating point numbers ("f"). When used in fixed format mode, all three cause the given number of characters to be read in: leading and trailing spaces are then stripped, and an attempt is made to parse all remaining characters as a single number. The Modifier is ignored in the "i" and "n" cases, but specifies the number of decimal places expected in the "f" case:

?- fread( i, 5, 0, I ) <= `1234567890`.
I = 12345

?- fread( n, 5, 0, N ) <= `1234567890`.
N = 12345

?- fread( f, 5, 3, F ) <= `1.234567890`.
F = 1.234

Bearing in mind that WIN-PROLOG treats all integers as signed during internal arithmetic, and furthermore automatically converts floating point numbers to integers where possible, this trio of formats should be thought of more as data import filters, rather than type specifiers.

This can be shown by the following example, which returns an integer:

?- fread( f, 5, 3, F ) <= `1.000234`.
F = 1
The only difference between the "i" and "n" formats is that the former reads integers that have been written in the range $[-2^{31}..2^{31}]$ while the latter handles numbers that have been written in the range $[0..2^{32}-1]$. When used in free format mode, all three decimal number input formats simply read the next non-white-space token from the input stream, and attempt to interpret it as a number:

?- fread( i, 0, 0, I ) <- ` -12345 `.  
I = -12345

?- fread( n, 0, 0, N ) <- ` 4294967295 `.  
N = -1

?- fread( f, 0, 3, F ) <- ` 123.456 `.  
F = 123.456

The arbitrary radix group comprises a single format ("r") which is used to input numbers in bases other than 10. All such numbers are expected to have leading zeros, rather than spaces: when used in fixed format mode, the number of characters specified in Width are input, and assumed to be numeric digits or uppercase letters representing digits in the base specified by the Modifier. Bases may be anywhere in the range $[2..36]$:

?- fread( r, 5, 8, R ) <- `1234567890`.  
R = 5349

?- fread( r, 5, 10, R ) <- `1234567890`.  
R = 12345

?- fread( r, 8, 16, R ) <- `1234ABCD`.  
R = 305441741

When used in free format mode, this format simply reads the next non-white-space token from the input stream, and attempts to interpret it as a number in the appropriate radix:

?- fread( r, 0, 2, R ) <- ` 0100 `.  
R = 4

?- fread( r, 0, 16, R ) <- ` 12AB 34CD `.  
R = 4779
The radix format is very useful when importing data from other computer files, such as database indices and hexadecimal dumps; irrespective of the base used, a standard Prolog integer is the result of a successful read with the "r" format.
free/9

return the current amounts of free space

\( \text{free( B, L, R, H, T, P, S, I, O )} \)

- B \quad \text{<variable>}
- L \quad \text{<variable>}
- R \quad \text{<variable>}
- H \quad \text{<variable>}
- T \quad \text{<variable>}
- P \quad \text{<variable>}
- S \quad \text{<variable>}
- I \quad \text{<variable>}
- O \quad \text{<variable>}

**Comments**

This returns the amount of free space, in bytes, in each of these memory areas: backtrack stack (B), local stack (L), reset stack (R), term heap (H), text heap (T), program heap (P), system stack (S), string input buffer (I) and string output buffer (O).

**Examples**

The following call returns the amount of program space currently free:

?\(- \text{free( _, _, _, _, _, P, _, _, _ ).} \)<enter>

P = 7151744

**Notes**

The free/9 predicate does not call the garbage collector, so the amounts of space reported in the term (H) and text (T) heaps may be less than that potentially available. If it is desired to obtain the full amount of either of these areas, a call to gc/1 should be inserted immediately prior to the call to free/9:

?\(- \text{gc( 1 ), free( _, _, _, H, T, _, _, _, _ ).} \)<enter>

H = 261490 ,
T = 1929504

Another predicate, total/9, returns the total amounts of memory available in the same nine areas: by comparing the results of calling free/9 and total/9, it is possible to compute additional statistics, such as the amount of space being used, or the percentage of space remaining free. A further related predicate, xinit/9, allows changes to be made to the memory allocation at runtime. Using this, it is possible to adjust memory usage in response to error conditions, or simply to optimise resource allocation within an application.
functor/3

convert between a term and its functor and arity

functor( Term, Functor, Arity )

?Term <term> or <variable>
?Functor <term> or <variable>
?Arity <integer> or <variable>

Comments  This succeeds if the given Term has the specified Functor and Arity. If Term is an uninstantiated variable, then Arity must be an integer greater than or equal to zero (0): if Arity is zero, Term will be bound to Functor; otherwise, it will be bound to a compound term with the given Functor and Arity, each argument being defined as a distinct unbound variable.

Examples  The following call builds a compound term using the given functor and arity:

?- functor( T, foo, 3 ).
T = foo(_1,_2,_3)

The next call returns the functor and arity of a given term:

?- functor( 123(456), F, A ).
F = 123 ,
A = 1

Notes  As shown by the second example above, WIN-PROLOG supports compound terms with functors other than atoms: indeed, the functor of a compound term may be a term of any type whatsoever. In traditional Prolog systems, lists were often represented as nested terms of the functor ".": the architecture of WIN-PROLOG is considerably more advanced than that of many Prolog implementations, and amongst other features has an efficient, true "list" data type. As a direct result of this, functor/3 does not convert arity-2 "dot" tuples into lists:

?- functor( X, ., 2 ).
X = '.'(_123,_456)

In traditional Prolog systems, the result would have been:

X = [ _123 | _456 ]
fwrite/4

write a formatted term

fwrite( Format, Width, Modifier, Term )

+Format   <atom> in the domain \{a,b,f,i,n,r,s\}.
+Width    <integer> in the range \[-8192..8192\]
+Modifier <integer> in the range \[-8192..8192\]
+Term     <term>

Comments This writes a Term to the current output stream using the given Format, field Width and Modifier. The Format is a single-character atom, defining the type of Term to be written:

<table>
<thead>
<tr>
<th>Format</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>atom</td>
</tr>
<tr>
<td>c</td>
<td>char list</td>
</tr>
<tr>
<td>f</td>
<td>fixed point number</td>
</tr>
<tr>
<td>i</td>
<td>integer</td>
</tr>
<tr>
<td>n</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>r</td>
<td>arbitrary radix</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
</tbody>
</table>

The field Width may be any positive or negative number, specifying the number of characters to be written to the output stream, or the special value zero (0), which allows "free format" output (see below). The Modifier is an integer which further defines the treatment of several formats (again, see below).

Examples The following call writes an integer right-justified in a field of five characters, surrounded by asterisks (*) to demonstrate the limits of the field:

?- write( * ), fwrite( i, 5, 0, 123 ), write( * ), nl.  
* 123*  
yes
Here, an atom is output left justified in a five-character field:

```prolog
?- write( * ), fwrite( a, -5, 0, abc ), write( * ), nl.
*abc  *
yes
```

**Notes**

As the above examples show, `fwrite/4` is a normal output predicate that can be used with any output stream. For the sake of clarity, however, all remaining examples in this section will use string output, via the `~>/2` predicate. This approach allows us to examine the exact relationships between a term, the format specification, and the resulting output data.

There are, broadly, three main groups of formats, respectively handling text, decimal number and arbitrary radix number output. Furthermore, there are three basic modes of formatted output: right-justified fixed, left-justified fixed and free. In this section, we will examine each group and each mode in turn.

The text group of formats comprises three separate cases which output text as atoms ("a"), char lists ("c") or strings ("s"). In fixed format mode, the sign of the field `Width` determines whether the item is left (negative) or right (positive) justified:

```prolog
?- fwrite( a, 5, 0, abc ) -> S.
S = `  abc`

?- fwrite( a, -5, 0, abc ) -> S.
S = `abc  `

?- fwrite( b, 5, 0, "abc" ) -> S.
S = `  abc`

?- fwrite( s, -5, 0, `abc` ) -> S.
S = `abc  `
```

In all three cases, the sign of the `Modifier` (but not its value) determines whether to truncate (negative) or generate a field overflow error (positive) when the given text item is longer than the field `Width`:

```prolog
?- fwrite( a, 5, -1, abcdef ) -> S.
! ----------------------------------------
! Error 41 : Format Field Overflow
! Goal     : fwrite(a,5,0,abcdef)
```
?- fwrite( a, 5, -1, abcdef ) -> S.
S = `abcde`

When used in free format mode (specified by a Width of zero), the text item is output directly, with no padding. In this case, the Modifier is ignored:

?- fwrite( a, 0, 0, 'hello world' ) -> S.
S = `hello world`

?- fwrite( b, 0, 0, "hello world" ) -> S.
S = `hello world`

?- fwrite( s, 0, 0, `hello world` ) -> S.
S = `hello world`

The number group of formats provides output of numbers as signed ("i") and unsigned ("n") decimal integers, or as floating point numbers ("f"). In fixed format mode, the sign of the field Width determines whether the number is left (negative) or right (positive) justified; the value of the Modifier is ignored in the "i" and "n" cases, but specifies the number of decimal places expected in the "f" case:

?- fwrite( i, 8, 0, 123 ) ~> S.
S = `     123`

?- fwrite( n, -8, 0, 123 ) ~> S.
S = `123     `

?- fwrite( f, 8, 3, 123 ) ~> S.
S = ` 123.000`

In all three cases, the sign of the Modifier determines whether to truncate (negative) or generate a field overflow error (positive) when the given number exceeds field Width:

?- fwrite( i, 5, 0, -1 ) ~> S.
S = `   -1`

?- fwrite( n, 5, 0, -1 ) ~> S.
! -----------------------------
! Error 41 : Format Field Overflow
! Goal : fwrite(n,5,0,-1)
?- fwrite( n, 5, -1, -1 ) -> S.
S = `42949`

Bearing in mind that WIN-PROLOG treats all integers as signed during internal arithmetic, and furthermore automatically converts floating point numbers to integers where possible, this trio of formats should be thought of more as a data export filters, rather than type specifiers. This can be shown by the following example, which outputs an integer in floating point format:

?- fwrite( f, 8, 3, 123 ) -> S.
S = ` 123.000`

The only difference between the "i" and "n" formats is that the former writes integers as signed an in the range [-214783648..214783647] while it outputs numbers unsigned, in the range [0..4294967295]. When used in free format mode, all three decimal number output formats simply write the number as specified, but with no padding:

?- fwrite( i, 0, 0, -1 ) -> S.
S = `-1`

?- fwrite( n, 0, 0, -1 ) -> S.
S = `4294967295`

The arbitrary radix group comprises a single format ("r") which is used to output numbers in bases other than 10. All such numbers are displayed with leading zeros, rather than spaces when displayed in fixed format mode, so in this format alone the sign of Width is ignored; meanwhile, the Modifier defines the base, which may be anywhere in the range [2..36]:

?- fwrite( r, 2, 10, 3 ) -> S.
S = `03`

?- fwrite( r, 5, 36, 29234652 ) -> S.
S = `HELLO`

When used in free format mode, this format simply writes the number as specified, but with no padding leading zeros:

?- fwrite( r, 0, 10, 3 ) -> S.
S = `3`
The radix format is very useful when exporting data to other computer files, such as database indices and hexadecimal dumps; it is also useful when outputting decimal numbers with leading zeros, such as in date or time displays.
garbage_collect/0

invoke the garbage collector explicitly

    garbage_collect

Comments

This forces a garbage collection if the "gc" Prolog flag is set to "on" (see gc/0 and nogc/0).

Examples

The following call invokes the garbage collector if the "gc" Prolog flag is set to "on":

    ?- garbage_collect.
    yes

Notes

This predicate is one of several included solely for Quintus Prolog compatibility. In WIN-PROLOG, the garbage collector is always enabled, and runs entirely automatically whenever required. There is never any need to set garbage collection "modes", nor can the garbage collector ever be disabled.

The only time it is ever desirable to run the garbage collector explicitly is before calling free/9, in order to work out how much space is available in the term or text heaps, or possibly before running a benchmark routine or profiling a program in order to provide a repeatable test. In either event, it is best to invoke the garbage collector with the gc/1 predicate, whose argument defines the type of garbage collection to perform.
**garbage_collect/1**

*invoke the garbage collector explicitly using given type*

```
   garbage_collect( Type )
```

+Type  <atom> in the domain {heap,text}

**Comments**

This forces a garbage collection of the given type, as shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap</td>
<td>collect garbage in the term heap only</td>
</tr>
<tr>
<td>text</td>
<td>collect garbage in the term and text heaps</td>
</tr>
</tbody>
</table>

**Examples**

The following call invokes a garbage collection of the term and text heaps:

```
?- garbage_collect( text ).
yes
```

**Notes**

This predicate is one of several included solely for Quintus Prolog compatibility. In **WIN-PROLOG**, the garbage collector is always enabled, and runs entirely automatically whenever required. There is never any need to set garbage collection "modes", nor can the garbage collector ever be disabled.

The only time it is ever desirable to run the garbage collector explicitly is before calling `free/9`, in order to work out how much space is available in the term or text heaps, or possibly before running a benchmark routine or profiling a program in order to provide a repeatable test. In either event, it is best to invoke the garbage collector with the `gc/1` predicate, whose argument defines the type of garbage collection to perform.
Enable certain explicit garbage collection invocations

\texttt{gc}

**Comments**
This sets the value of the "gc" Prolog flag to "on", so that the garbage_collect/0 predicate will be enabled (also see \texttt{nogc/0}, which resets the "gc" Prolog flag to "off").

**Examples**
The following sets the "gc" Prolog flag to "on":

\begin{verbatim}
?- gc.
yes
\end{verbatim}

**Notes**
This predicate is one of several included solely for Quintus Prolog compatibility. In \textbf{WIN-PROLOG}, the garbage collector is always enabled, and runs entirely automatically whenever required. There is never any need to set garbage collection "modes", nor can the garbage collector ever be disabled.

The only time it is ever desirable to run the garbage collector explicitly is before calling \texttt{free/9}, in order to work out how much space is available in the term or text heaps, or possibly before running a benchmark routine or profiling a program in order to provide a repeatable test. In either event, it is best to invoke the garbage collector with the \texttt{gc/1} predicate, whose argument defines the type of garbage collection to perform.
**gc/1**

*perform an explicit garbage collection*

```
gc( Type )
```

+Type <integer> in the range [0..3]

**Comments**

This performs a garbage collection of the given type, as shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>collect garbage in the term heap only</td>
</tr>
<tr>
<td>1</td>
<td>collect garbage in the term and text heaps</td>
</tr>
<tr>
<td>2</td>
<td>collect garbage in the term heap and reset indices</td>
</tr>
<tr>
<td>3</td>
<td>collect garbage in the term and text heaps and reset indices</td>
</tr>
</tbody>
</table>

**Examples**

The following call invokes a garbage collection of the term and text heaps:

```
?- gc( 1 ).
yes
```

**Notes**

Of all the garbage collection predicates, gc/1 is the one preferred for performing explicit garbage collections, as it is directly mapped to the WIN-PROLOG architecture, unlike garbage_collect/0, garbage_collect/1 and gc/0: these latter three predicates are included solely for Quintus Prolog compatibility. In WIN-PROLOG, the garbage collector is always enabled, and runs entirely automatically whenever required. There is never any need to set garbage collection "modes", nor can the garbage collector ever be disabled.

The only time it is ever desirable to run the garbage collector explicitly is before calling `free/9`, in order to work out how much space is available in the term or text heaps, or possibly before running a benchmark routine or profiling a program in order to provide a repeatable test. In either event, it is best to invoke the garbage collector with the `gc/1` predicate, whose argument defines the type of garbage collection to perform.
get/1

read a printable character from the current input stream

```prolog
get( Char )
```

?Char <variable> or <char>

**Comments**
This reads the next "printable" (non-white-space) character from the current input stream, and unifies Char with its integer character code. A printable character is defined to be one whose character code is greater than that of `<space>` (20h).

**Examples**
The following call reads the first printable character from the next line of input at the console, which begins with a `<tab>` character (this is skipped). Note that a call to the `flush/0` predicate has been added: this is to dispose of any excess characters ("ello" in this example):

```prolog
?- get( C ), flush. <enter>
|: hello <enter>
C = 104
```

**Notes**
The `get/1` predicate is closely related to `get0/1`: the only difference is that the former scans input for printable characters (those with character codes of 21h or greater), while the latter returns all characters. Both these predicates return the special value of "-1" if they encounter the end of file on the current input stream.
get0/1

read a character from the current input stream

get0( Char )

?Char <variable> or <char>

Comments  This reads the next character from the current input stream, and unifies Char with its integer character code.

Examples  The following call reads the first character from the next line of input at the console, which begins with a <tab> character (this is returned). Note that a call to the flush/0 predicate has been added: this is to dispose of any excess characters ("hello" in this example):

?- get0( C ), flush.
|: hello
C = 9

Notes  The get0/1 predicate is closely related to get/1: the only difference is that the former returns all characters, while the latter scans input for printable characters (those with character codes of 21h or greater). Both these predicates return the special value of ",-1" if they encounter the end of file on the current input stream.
**getb/1**

get a character directly from the keyboard or mouse

\[ \text{getb}( \text{Char} ) \]

-Char <variable>

**Comments**

This inputs a single Char directly from the keyboard or mouse, returning the integer character code for any key pressed, or a negative code representing a mouse button, as shown in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>left button pressed</td>
</tr>
<tr>
<td>-2</td>
<td>right button pressed</td>
</tr>
<tr>
<td>-4</td>
<td>middle button pressed</td>
</tr>
</tbody>
</table>

**Examples**

The following call waits for a key to be pressed, in this case the space bar; note that it is not necessary to press <enter> to complete the data input:

?- \text{getb}( \text{C} ).

<space>\text{C} = 32

**Notes**

This predicate is one of a special family of three which is principally aimed at helping the debugging of applications. Each of getb/1, grab/1, and putb/1 performs its input or output directly from or to the lowest-level of user I/O.

The getb/1 will return immediately if a byte is stored in the keyboard or mouse type-ahead buffer; otherwise, it will suspend program execution until a key or mouse button is pressed. The related grab/1 predicate also returns immediately any character stored in type-ahead, but simply fails if nothing is there: this allows programs to peek into type-ahead without risking being suspended if nothing is present. Finally, putb/1 outputs the given byte directly to the console window, irrespective of whichever output stream is currently set.
getx/2

read a word from the current input stream

\[ \text{getx( Size, Word )} \]

+Size <integer> in the range [-1..4] (X86) or [-1..8] (X64)
-Word <variable>

Comments
This reads the next Word of the given Size from the current input stream. Where Size is in the range [1..4] (X86) or [1..8] (X64), this number of bytes are read and assembled into an integer, assuming "little endian" encoding. If Size is zero (0), one byte of input is returned but not consumed; if Size is minus one (-1), eight bytes are read and interpreted as a floating point number.

Examples
The following call reads a four-byte (32-bit) word from a string, interpreting the result as an integer:

\[ \text{?- getx( 4, W ) <- `~@-A-@-A`} . \]

W = 256

Notes
The getx/2 predicate is useful for reading binary data from files imported from other applications. It assumes that files contain data in the "little endian" format, where the least significant byte of a long word is stored first, followed successively by each byte of greater significance. This model is employed by the Intel 80386 and all its successors, and is therefore inherent to Windows file and data structures. The related putx/2 predicate can be used to write integers in little endian binary format.

Used with a Size of one (1), getx/2 is effectively identical to get0/1: used with a size of zero (0), it is useful for peeking ahead at the input stream without consuming any data from it. The other two sizes most commonly of interest are two (2) and four (4), which relate to standard 16-bit and 32-bit words respectively: the size of three (3) is also supported, although 24-bit little endian integers are not normally used in data files in the Windows environment. The special case of minus one (-1) allows an IEEE-format 64-bit floating point number to be read and returned as an integer or a float as appropriate.

Unlike get/1 and get0/1, the present predicate fails if it encounters the end of file on the current input stream at the start of reading: if the word size has been specified as "2" or more, and end of file occurs after the first byte has been input, then an end of file error is generated.
gfx/1

perform graphics on the current device context

gfx( Grafix )
+Grafix <term>

Comments
This performs the "graphics procedure" specified in Grafix on the current device context (see gfx_begin/1, gfx_paint/1 and gfx_begin/3). A graphics procedure may consist of anything from a single graphics primitive to a deeply nested combination of object selections and primitive operations: the GraFiX subsystem is described in more detail in Appendix E.

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

```prolog
foo :-
    wcreate( foo, grafix, '', 10, 10, 100, 100, 16'80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).
```

The window handler, foo/4, has just one clause, which reacts to "msg_paint" messages by getting a clipped device context, drawing a grey circle, and then releasing the device context:

```prolog
foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
    gfx_end( foo ).
```

Once compiled, the following command will display a small square window which contains a grey circle:

```
?- foo.
yes <enter>
```

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Of these, one predicate in particular, namely gfx/1, supports a special sub-language that enables complex graphical operations to be easily represented as a series of "object" selections and "primitive" operations. Details of this sub-language can be found in Appendix E.
gfx_back_close/1

close a background object

gfx_back_close( Name )

+Name <atom>

Comments
This closes the background object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples
The following call will close the background object called "foo", assuming that such an object has previously been created:

?- gfx_back_close( foo ).
yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_back_create/4

create a background object

gfx_back_create( Name, Red, Green, Blue )

+Name <atom>
+Red <integer> in the range [0..255]
+Green <integer> in the range [0..255]
+Blue <integer> in the range [0..255]

Comments
This creates a background object with the given Name, using the colour specified by the given combination of Red, Green and Blue levels. If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples
The following call will create a background object called "foo", specifying its colour as bright yellow:

?- gfx_back_create( foo, 255, 255, 0 ).
yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx\"n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_back_dict/2

return a dictionary of background objects

gfx_back_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments
This returns a list of all the currently defined background objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples
The following call will return a list of currently defined background objects, assuming that one called "foo" has previously been created:

?- gfx_back_dict( 0, D ). 
D = [foo]

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_back_handle/2

convert between a background object and its handle

gfx_back_handle( Object, Handle )

?Object  <object_handle> or <variable>
?Handle   <integer> or <variable>

Comments  This converts between a background object and its raw (integer) handle. The Object parameter may be a named object, a handle of the form, "back(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

Examples  The following call will return the raw handle for a background object called "foo", assuming that such an object has previously been created:

?- gfx_back_handle( foo, H ).  
H = 65535

Notes  The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
**gfx_begin/1**

obtain an unclipped device context to begin graphics

```prolog
gfx_begin( Window )
```

+Window <window_handle> or <empty_list>

**Comments**

This saves the current graphics "device context", before obtaining a new, unclipped one for the named Window in preparation for drawing graphics. If the Window is specified as an empty list, the device context belonging to the current printer is obtained. The new device context is initialised as follows:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>stock(null_back)</td>
</tr>
<tr>
<td>brush</td>
<td>stock(ltgray_brush)</td>
</tr>
<tr>
<td>font</td>
<td>stock(prolog_fixed_font)</td>
</tr>
<tr>
<td>foreground</td>
<td>stock(white_fore)</td>
</tr>
<tr>
<td>pen</td>
<td>stock(black_pen)</td>
</tr>
<tr>
<td>mapping mode</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>origin</td>
<td>(0,0)</td>
</tr>
<tr>
<td>rop</td>
<td>stock(copypen_rop)</td>
</tr>
</tbody>
</table>

**Examples**

The following program creates a "grafix" window, obtains an unclipped device context for it, draws a grey circle, and then releases the device context:

```
foo :-
    wcreate( foo, grafix, ` `, 10, 10, 100, 100, 16'90800000 ),
    gfx_begin( foo ),
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
    gfx_end( foo ).
```

Once compiled, the following command will display a small square window which contains a grey circle:

```
?- foo.
```
The following short program sets up a print job, called "Square", through a common dialog (prnbox/4), begins a page (prnpag/1), uses the gfx*/n predicates to draw a grey square on the printer, and then terminates the print job (prnend/1):

```prolog
print_square( Printer, Driver, Port, Page ) :-
    prnbox( 'Square', Printer, Driver, Port ),
    prnpag( Page ),
    gfx_begin( [] ),
    gfx( rectangle(400,400,800,800) ),
    gfx_end( [] ),
    prnend( 0 ).
```

Once compiled, the following command will print a single page containing a grey square near its top left corner:

```prolog
?- print_square( N, D, P, C ).
N = 'HP DeskJet Plus' ,
D = 'HPDSKJET' ,
P = 'LPT1:' ,
C = 1
```

**Notes**

The first example only draws the graphics once, when the window is created: no provision is made for redrawing the image if it is overwritten by another window. See gfx_paint/1 for details of how to redraw "dirty" portions of windows.

Device contexts are scarce resources, and should always be "returned" when finished with. Every call to gfx_begin/1 or gfx_paint/1 must be matched by a later call to gfx_end/1, ideally just with one or more calls to gfx/1 in between:

```prolog
ideal_grafix( W ) :-
    gfx_begin( W ),
    ... perform one or more gfx/1 calls here ... 
    gfx_end( W ).

ideal_repaint( W ) :-
    gfx_paint( W ),
    ... perform one or more gfx/1 calls here ... 
    gfx_end( W ).
```
The **WIN-PROLOG** "$\text{GraFiX subsystem}$" comprises a complete set of predicates, generically referred to as "$\text{gfx}\_\text{nn}$" because virtually all their names begin with the letters "$\text{gfx}$". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in *Appendix E*. 
gfx_begin/3

obtain a device context to begin hit testing

    gfx_begin( Window, X, Y )

+Window          <window_handle>
+X              <integer>
+Y              <integer>

Comments
This saves the current graphics "device context", before obtaining a new, unclipped one for the named Window in preparation for hit testing on the pair of device coordinates, \((X,Y)\). The new device context is initialised as follows:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>font</td>
<td>stock(prolog_fixed_font)</td>
</tr>
<tr>
<td>mapping mode</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>origin</td>
<td>(0,0)</td>
</tr>
</tbody>
</table>

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

    foo :-
    wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).

The first clause of the window handler reacts to "msg_paint" messages by getting a clipped device context, calling grafix/0 to draw a grey circle, and then releasing the device context:

    foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    grafix,
    gfx_end( foo ).

The second clause reacts to "msg_mousemove" messages by getting a hit test device context to test the mouse coordinates \((X,Y)\), this time calling grafix/0 to test the circle against the mouse coordinates, releasing the device context. Finally, it checks whether the returned
"hit count" is one (1), binding the handler's fourth argument to the atom "close" if so: this in turn causes the window to be hidden.

```prolog
foo( foo, msg_mousemove, (X,Y), close ) :-
    gfx_begin( foo, X, Y ),
    grafix,
    gfx_test( Hits ),
    gfx_end( foo, X, Y ),
    Hits = 1.
```

The following simple graphics routine is used by both the "msg_paint" and "msg_mousemove" cases of the window handler, foo/4:

```prolog
grafix :-
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ).
```

Once compiled, the following command will display a small square window which contains a grey circle:

```
?- foo.
```

The window, complete with its circle, will remain on the screen until the user moves the mouse pointer over the circle. At this point, the window will suddenly vanish.

**Notes**

The `gfx/1` calls made during hit tests are identical to those used to draw graphics: the type of device context determines whether graphics programs generate output or count "hits".

Device contexts are scarce resources, and should always be "returned" when finished with. Every call to `gfx_begin/3` must be matched by a later call to `gfx_end/3`, ideally just with one or more calls to `gfx/1` and `gfx_test/1` in between:

```prolog
ideal_hit_test( W, X, Y, H ) :-
    gfx_begin( W, X, Y ),
    ... perform one or more gfx/1 calls here ...
    gfx_test( H ),
    gfx_end( W, X, Y ).
```

The **WIN-PROLOG "GraFiX subsystem"** comprises a complete set of predicates, generically referred to as "gfx" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device
contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_bitmap_close/1

close a bitmap object

gfx_bitmap_close( Name )
+Name <atom>

Comments This closes the bitmap object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples The following call will close the bitmap object called "foo", assuming that such an object has previously been created:

?- gfx_bitmap_close( foo ).

yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_bitmap_dict/2

return a dictionary of bitmap objects

gfx_bitmap_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments
This returns a list of all the currently defined bitmap objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples
The following call will return a list of currently defined bitmap objects, assuming that one called "foo" has previously been created:

?- gfx_bitmap_dict( 0, D ).
D = [foo]

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_bitmap_handle/2

convert between a bitmap object and its handle

gfx_bitmap_handle( Object, Handle )

?Object <object_handle> or <variable>
?Handle <integer> or <variable>

Comments This converts between a bitmap object and its raw (integer) handle. The Object parameter may be a named object, or a handle of the form, "bitmap(Raw)", where "Raw" is the raw handle. There are no "stock" bitmaps.

Examples The following call will return the raw handle for a bitmap object called "foo", assuming that such an object has previously been created:

?- gfx_bitmap_handle( foo, H ).
H = 65535

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx\*n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_bitmap_load/2

load a bitmap object

gfx_bitmap_load( Name, File )
+Name <atom>
+File <atom>

Comments This loads a bitmap object with the given Name from the specified File. If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples The following call will load a bitmap object called "foo" from the file "foo.bmp" in the current directory:

?- gfx_bitmap_load( foo, 'foo.bmp' ). yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.

This is one of a series of file-oriented predicates which are implemented directly in terms of the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
gfx_brush_close/1

close a brush object

    gfx_brush_close( Name )

+Name <atom>

Comments This closes the brush object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples The following call will close the brush object called "foo", assuming that such an object has previously been created:

    ?- gfx_brush_close( foo ).

yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_brush_create/5

create a brush object

gfx_brush_create( Name, Red, Green, Blue, Style )

+Name <atom>
+Red <integer> in the range [0..255]
+Green <integer> in the range [0..255]
+Blue <integer> in the range [0..255]
+Style <atom>

Comments This creates a brush object with the given Name, using the colour specified by the given combination of Red, Green and Blue levels and named Style (styles are listed in Appendix E). If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples The following call will create a brush object called "foo", specifying its colour as bright yellow and style as "solid":

?- gfx_brush_create( foo, 255, 255, 0, solid ).

yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_brush_dict/2

return a dictionary of brush objects

gfx_brush_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments
This returns a list of all the currently defined brush objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples
The following call will return a list of currently defined brush objects, assuming that one called "foo" has previously been created:

?- gfx_brush_dict( 0, D ).
D = [foo]

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_brush_handle/2

convert between a brush object and its handle

gfx_brush_handle( Object, Handle )

?Object <object_handle> or <variable>
?Handle <integer> or <variable>

Comments
This converts between a brush object and its raw (integer) handle. The Object parameter may be a named object, a handle of the form, "brush(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

Examples
The following call will return the raw handle for a brush object called "foo", assuming that such an object has previously been created:

?- gfx_brush_handle( foo, H ).<enter>

H = 65535

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_cleanup/0

release all saved device contexts

gfx_cleanup

**Comments**

This terminates all graphics operations abruptly, releasing all saved device contexts including the current one, and should be used in any error handler which is likely to interrupt a program that is performing graphics. It always succeeds, whether or not there were any orphaned device contexts to release.

**Examples**

The following error handler safely releases any device contexts that might have been orphaned by an error during a graphics program:

```prolog
'?ERROR?'( Number, Goal ) :-
gfx_cleanup,
error_hook( Number, Goal ).
```

**Notes**

Device contexts are scarce resources, and should always be "returned" when finished with. However, it is possible that even programs written in the correct form, such as that illustrated below, generate an error or fail between obtaining a device context and then releasing it:

```prolog
ideal_grafix( W ) :-
gfx_begin( W ),
... an unintentional error occurs during this bit ...
gfx_end( W ).
```

In such cases, the orphaned device contexts should be restored by the present predicate. Please note that in the development environment, gfx_cleanup/0 is called automatically whenever a program aborts (calls abort/0).

The **WIN-PROLOG "GraFiX subsystem"** comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_clipping/4

set or get a clipping rectangle

gfx_clipping( Left, Top, Right, Bottom )

?Left <integer> or <variable>
?Top <integer> or <variable>
?Right <integer> or <variable>
?Bottom <integer> or <variable>

Comments This sets or gets the Left, Top, Right and Bottom device coordinates of the current device context's clipping rectangle.

Examples The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

```
foo :-
    wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).
```

The window handler has just one clause, which reacts to "msg_paint" messages by getting a clipped device context, applying a further clipping rectangle to it, drawing as much as it can of a grey circle, and then releasing the device context:

```
foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    gfx_clipping( 40, 10, 60, 110 ),
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
    gfx_end( foo ).
```

Once compiled, the following command will display a small window which contains a rectangular portion of a grey circle:

```
?- foo.
yes
```

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device
contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_cursor_handle/2

convert between a cursor object and its handle

gfx_cursor_handle( Object, Handle )

?Object <object_handle> or <variable>
?Handle <integer> or <variable>

Comments  This converts between a cursor object and its raw (integer) handle. The Object parameter may be a handle of the form, "cursor(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

Examples      The following call will return the raw handle for a stock cursor object called "stock(wait_cursor)"

    ?- gfx_cursor_handle( stock(wait_cursor), H ).  <enter>

    H = 65535

Notes         The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_end/1

release a device context to end graphics

gfx_end( Window )

+Window <window_handle> or <empty_list>

Comments
This releases the current graphics "device context" for the named Window in order to end a graphics sequence, before restoring any previously saved device context. If the Window is specified as an empty list, the device context belonging to the current printer is released.

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

```
foo :-
  wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
  window_handler( foo, foo ),
  wshow( foo, 1 ).
```

The window handler has just one clause, which reacts to "msg_paint" messages by getting a clipped device context, drawing a grey circle, and then releasing the device context:

```
foo( foo, msg_paint, _, _ ) :-
  gfx_paint( foo ),
  gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
  gfx_end( foo ).
```

Once compiled, the following command will display a small square window which contains a grey circle:

```
?- foo.
yes
```

Notes
Device contexts are scarce resources, and should always be "returned" when finished with. Every call to gfx_begin/1 or gfx_paint/1 must be matched by a later call to gfx_end/1, ideally just with one or more calls to gfx/1 in between:

```
ideal_grafix( W ) :-
  gfx_begin( W ),
```

... perform one or more gfx/1 calls here ...
gfx_end( W ).

ideal_repaint( W ) :-
gfx_paint( W ),
... perform one or more gfx/1 calls here ...
gfx_end( W ).

The **WIN-PROLOG** "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in *Appendix E*. 
**gfx_end/3**

*release a device context to end hit testing*

```prolog
gfx_end( Window, X, Y )
```

+Window <window_handle>
+X <integer>
+Y <integer>

**Comments**
This releases the current testing "device context" for the named Window in order to end a hit test sequence on the pair of device coordinates, \((X,Y)\), before restoring any previously saved device context.

**Examples**
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

```prolog
foo :-
    wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).
```

The first clause of the window handler reacts to "msg_paint" messages by getting a clipped device context, calling grafix/0 to draw a grey circle, and then releasing the device context:

```prolog
foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    grafix,
    gfx_end( foo ).
```

The second clause reacts to "msg_mousemove" messages by getting a hit test device context to test the mouse coordinates \((X,Y)\), this time calling grafix/0 to test the circle against the mouse coordinates, releasing the device context. Finally, it checks whether the returned "hit count" is one (1), binding the handler's fourth argument to the atom "close" if so: this in turn causes the window to be hidden.

```prolog
foo( foo, msg_mousemove, (X,Y), close ) :-
    gfx_begin( foo, X, Y ),
    grafix,
    gfx_test( Hits ),
```
gfx_end( foo, X, Y ),
Hits = 1.

The following simple graphics routine is used by both the "msg_paint" and "msg_mousemove" cases of the window handler, foo/4:

grafix :-
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ).

Once compiled, the following command will display a small square window which contains a grey circle:

?- foo.
yes

The window, complete with its circle, will remain on the screen until the user moves the mouse pointer over the circle. At this point, the window will suddenly vanish.

Notes

The gfx/1 calls made during hit tests are identical to those used to draw graphics: the type of device context determines whether graphics programs generate output or count "hits".

Device contexts are scarce resources, and should always be "returned" when finished with. Every call to gfx_begin/3 must be matched by a later call to gfx_end/3, ideally just with one or more calls to gfx/1 and gfx_test/1 in between:

ideal_hit_test( W, X, Y, H ) :-
    gfx_begin( W, X, Y ),
    ... perform one or more gfx/1 calls here ... 
    gfx_test( H ),
    gfx_end( W, X, Y ).

The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_font_close/1

close a font object

gfx_font_close( Name )

+Name <atom>

Comments
This closes the font object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples
The following call will close the font object called "foo", assuming that such an object has previously been created:

?- gfx_font_close( foo ).
yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx"/n because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_font_create/4

create a font object

gfx_font_create( Name, Face, Size, Style )

+Name        <atom>
+Face        <atom>
+Size        <integer>
+Style       <atom>

Comments     This creates a font object with the given Name, using the typeface family specified by Face, in the given point Size and named Style (styles are listed in Appendix E). If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples      The following call will create a font object called "foo", specifying its face as "Arial", size as 16 points and style as "normal":

?- gfx_font_create( foo, arial, 16, normal ).

yes

Notes         The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx"/n because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_font_dict/2

return a dictionary of font objects

gfx_font_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}

-Dict <variable>

Comments This returns a list of all the currently defined font objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples The following call will return a list of currently defined font objects, assuming that one called "foo" has previously been created:

?- gfx_font_dict( 0, D ). <enter>
D = [foo]

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_font_handle/2

`convert between a font object and its handle`

```prolog
gfx_font_handle( Object, Handle )
```

?- `Object` <object_handle> or <variable>
?- `Handle` <integer> or <variable>

**Comments**
This converts between a font object and its raw (integer) handle. The Object parameter may be a named object, a handle of the form, "font(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

**Examples**
The following call will return the raw handle for a font object called "foo", assuming that such an object has previously been created:

```prolog
?- gfx_font_handle( foo, H ).
H = 65535
```

**Notes**
The **WIN-PROLOG "GraFiX subsystem"** comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in **Appendix E**.
gfx_fore_close/1

close a foreground object

    gfx_fore_close( Name )

+Name <atom>

Comments This closes the foreground object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples The following call will close the foreground object called "foo", assuming that such an object has previously been created:

    ?- gfx_fore_close( foo ).
    yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_fore_create/4

create a foreground object

gfx_fore_create( Name, Red, Green, Blue )

+Name <atom>
+Red <integer> in the range [0..255]
+Green <integer> in the range [0..255]
+Blue <integer> in the range [0..255]

Comments
This creates a foreground object with the given Name, using the colour specified by the given combination of Red, Green and Blue levels. If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples
The following call will create a foreground object called "foo", specifying its colour as bright yellow:

?- gfx_fore_create( foo, 255, 255, 0 ).

yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx"/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_fore_dict/2

return a dictionary of foreground objects

   gfx_fore_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments This returns a list of all the currently defined foreground objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples The following call will return a list of currently defined foreground objects, assuming that one called "foo" has previously been created:

        ?- gfx_fore_dict( 0, D ).
        D = [foo]

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_fore_handle/2

convert between a foreground object and its handle

gfx_fore_handle( Object, Handle )

?Object <object_handle> or <variable>
?Handle <integer> or <variable>

Comments  This converts between a foreground object and its raw (integer) handle. The Object parameter may be a named object, a handle of the form, "fore(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(\texttt{Name})", where "\texttt{Name}" is the name of an appropriate stock object.

Examples  The following call will return the raw handle for a foreground object called "foo", assuming that such an object has previously been created:

?- gfx_fore_handle( foo, H ).<enter>
H = 65535

Notes  The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_icon_close/1

close an icon object

    gfx_icon_close( Name )

+Name  <atom>

Comments  This closes the icon object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples  The following call will close the icon object called "foo", assuming that such an object has previously been created:

    %- gfx_icon_close( foo ).  
      yes

Notes  The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmap" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_icon_dict/2

return a dictionary of icon objects

gfx_icon_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments
This returns a list of all the currently defined icon objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples
The following call will return a list of currently defined icon objects, assuming that one called "foo" has previously been created:

?- gfx_icon_dict( 0, D ).
D = [foo]

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_icon_handle/2

convert between an icon object and its handle

    gfx_icon_handle( Object, Handle )

?Object          <object_handle> or <variable>
?Handle          <integer> or <variable>

Comments       This converts between an icon object and its raw (integer) handle. The Object parameter may be a named object, a handle of the form, "icon(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

Examples        The following call will return the raw handle for an icon object called "foo", assuming that such an object has previously been created:

    ?- gfx_icon_handle( foo, H ).  <enter>
    H = 65535

Notes           The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_icon_load/3

load an icon object

       gfx_icon_load( Name, File, Size )

+Name       <atom>
+File       <atom>
+Size       <integer>

Comments  This loads an icon object with the given Name matching the given Size in the specified icon File. If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool. If Size is specified as zero (0), the first icon in the resource is loaded at its default size.

Examples  The following call will load a 48*48 pixel icon object called "foo" from the first icon resource in the file "foo.ico" in the current directory:

       ?- gfx_icon_load( foo, 'foo.ico', 48 ).

         yes

Notes  This predicate was changed in WIN-PROLOG version 4.800: previously, the Size parameter was a numerical index into a file, which enabled a specific 32*32 pixel icon to be extracted from an icon library or executable file. With the advent of later versions of Windows, it became desirable to support smaller and larger icons, as well as icons with different bit depths, and the archaic index parameter was dropped in favour of one which allows the preferred display size to be specified. Many icon files contain multiple images, and gfx_icon_load/3 will automatically load the best fit for the linear pixel count specified in Size, and will further interpolate the image as necessary to obtain the exact size specified.

The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx\"/\"n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.

This is one of a series of file-oriented predicates which are implemented directly in terms of the Windows API. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
gfx_mapping/4

set or get a coordinate mapping

gfx_mapping( Xw, Yw, Xv, Yv )

?Xw <integer> or <variable>
?Yw <integer> or <variable>
?Xv <integer> or <variable>
?Yv <integer> or <variable>

Comments
This sets or gets the (Xw,Yw) window extents and (Xv,Yv) viewport extents of the current device context.

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

foo :-
    wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).

The window handler has just one clause, which reacts to "msg_paint" messages by getting a clipped device context, applying a coordinate mapping, drawing a squashed grey ellipse, and then releasing the device context:

foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    gfx_mapping( 3, 3, 2, 1 ),
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
    gfx_end( foo ).

Once compiled, the following command will display a small window which contains a squashed image of a grey ellipse:

?- foo.
yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device
contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_metafile_close/1

close a metafile object

gfx_metafile_close( Name )

+Name <atom>

Comments
This closes the metafile object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples
The following call will close the metafile object called "foo", assuming that such an object has previously been created:

?- gfx_metafile_close( foo ).

yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_metafile_dict/2

return a dictionary of metafile objects

gfx_metafile_dict( Flag, Dict )

+Flag          <integer> in the domain {-1,0,1}
-Dict          <variable>

Comments    This returns a list of all the currently defined metafile objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples    The following call will return a list of currently defined metafile objects, assuming that one called "foo" has previously been created:

?- gfx_metafile_dict( 0, D ).
D = [foo]

Notes       The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx/*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_metafile_handle/2

convert between a metafile object and its handle

gfx_metafile_handle( Object, Handle )

?Object <object_handle> or <variable>
?Handle <integer> or <variable>

Comments This converts between a metafile object and its raw (integer) handle. The Object parameter may be a named object, or a handle of the form, "metafile(Raw)", where "Raw" is the raw handle. There are no "stock" metafiles.

Examples The following call will return the raw handle for a metafile object called "foo", assuming that such an object has previously been created:

?- gfx_metafile_handle( foo, H ).

H = 65535

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx\*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_metafile_load/2

load a metafile object

    gfx_metafile_load( Name, File )

+Name <atom>
+File <atom>

Comments This loads a metafile object with the given Name from the specified File. If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples The following call will load a metafile object called "foo" from the file "foo.wmf" in the current directory:

    ?- gfx_metafile_load( foo, 'foo.wmf' ).
    yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.

This is one of a series of file-oriented predicates which are implemented directly in terms of the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
gfx_origin/2

set or get a coordinate origin

gfx_origin( Xo, Yo )

?Xo <integer> or <variable>
?Yo <integer> or <variable>

Comments
This sets or gets the (Xo,Yo) viewport origin of the current device context.

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

foo :-
    wcreate( foo, grafix, ``, 10, 10, 100, 100, 16`80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).

The window handler has just one clause, which reacts to "msg_paint" messages by getting a clipped device context, setting a viewport origin, drawing part of a grey circle, and then releasing the device context:

foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    gfx_origin( 50, 50 ),
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
    gfx_end( foo ).

Once compiled, the following command will display a window containing the top left quarter of a grey circle in its bottom right corner:

?- foo.
yes <enter>

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_paint/1

obtain a clipped device context to begin graphics

gfx_paint( Window )

+Window <window_handle>

Comments
This saves the current graphics "device context", before obtaining a new one for the named Window, clipped to the window's "dirty region", in preparation for drawing graphics. The new device context is initialised as follows:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>stock(null_back)</td>
</tr>
<tr>
<td>brush</td>
<td>stock(ltgray_brush)</td>
</tr>
<tr>
<td>font</td>
<td>stock(prolog_fixed_font)</td>
</tr>
<tr>
<td>foreground</td>
<td>stock(white_fore)</td>
</tr>
<tr>
<td>pen</td>
<td>stock(black_pen)</td>
</tr>
<tr>
<td>mapping mode</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>origin</td>
<td>(0,0)</td>
</tr>
<tr>
<td>rop</td>
<td>stock(copypen_rop)</td>
</tr>
</tbody>
</table>

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

```prolog
foo :-
  wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
  window_handler( foo, foo ),
  wshow( foo, 1 ).
```

The window handler has just one clause, which reacts to "msg_paint" messages by getting a clipped device context, drawing a grey circle, and then releasing the device context:

```prolog
foo( foo, msg_paint, _, _ ) :-
  gfx_paint( foo ),
  gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ),
```
gfx_end( foo ).

Once compiled, the following command will display a small square window which contains a grey circle:

?- foo.
yes

Notes

The above example shows the correct way to refresh graphics in a window. Every time part or all of a "grafix" window is uncovered or otherwise revealed, the WIN-PROLOG system sends its window handler a "msg_paint" message. The window handler should react by obtaining a device context through gfx_paint/1: this is automatically clipped to the "dirty region" of the window, so only those parts of a window which need redrawing are updated.

Device contexts are scarce resources, and should always be "returned" when finished with. Every call to gfx_begin/1 or gfx_paint/1 must be matched by a later call to gfx_end/1, ideally just with one or more calls to gfx/1 in between:

ideal_grafix( W ) :-
    gfx_begin( W ),
    ... perform one or more gfx/1 calls here ...
    gfx_end( W ).

ideal_repaint( W ) :-
    gfx_paint( W ),
    ... perform one or more gfx/1 calls here ...
    gfx_end( W ).

The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_pen_close/1

close a pen object

gfx_pen_close( Name )

+Name <atom>

Comments
This closes the pen object with the given Name, returning its resources to the system pool; if the object does not exist, this predicate fails.

Examples
The following call will close the pen object called "foo", assuming that such an object has previously been created:

?- gfx_pen_close( foo ).
yes

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_pen_create/5

create a pen object

gfx_pen_create( Name, Red, Green, Blue, Style )

+Name <atom>
+Red <integer> in the range [0..255]
+Green <integer> in the range [0..255]
+Blue <integer> in the range [0..255]
+Style <atom>

Comments This creates a pen object with the given Name, using the colour specified by the given combination of Red, Green and Blue levels and named Style (styles are listed in Appendix E). If a previous object of this type and name exists, it is automatically closed before the new version is created, returning its resources to the system pool.

Examples The following call will create a pen object called "foo", specifying its colour as bright yellow and style as "solid":

?- gfx_pen_create( foo, 255, 255, 0, solid ).

yes

Notes The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx" predicates because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_pen_dict/2

return a dictionary of pen objects

gfx_pen_dict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments
This returns a list of all the currently defined pen objects, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples
The following call will return a list of currently defined pen objects, assuming that one called "foo" has previously been created:

?- gfx_pen_dict( 0, D ). <enter>
D = [foo]

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_pen_handle/2

convert between a pen object and its handle

```
gfx_pen_handle( Object, Handle )
```

?Object <object_handle> or <variable>
?Handle <integer> or <variable>

Comments
This converts between a pen object and its raw (integer) handle. The Object parameter may be a named object, a handle of the form, "pen(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

Examples
The following call will return the raw handle for a pen object called "foo", assuming that such an object has previously been created:

```
?- gfx_pen_handle( foo, H ).
H = 65535
```

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_resolution/4

*check or get a device extent and resolution*

```prolog
gfx_resolution( Xw, Yw, Xr, Yr )
```

?Xw <integer> or <variable>
?Yw <integer> or <variable>
?Xr <integer> or <variable>
?Yr <integer> or <variable>

**Comments**
This checks or gets the \((Xw,Yw)\) extent and \((Xr,Yr)\) resolution of the device identified by the current device context.

**Examples**
The following program obtains an unclipped device context for WIN-PROLOG's "main window" (handle zero (0)), and uses it to obtain the extent (pixel dimensions) and resolution (pixels per inch) of client area of this window:

```prolog
get_size( Xw, Yw, Xr, Yr ) :-
gfx_begin( 0 ),
gfx_resolution( Xw, Yw, Xr, Yr ),
gfx_end( 0 ).
```

Once compiled, this program can be used to obtain the dimensions and resolution of the main window:

```prolog
?- get_size( Xw, Yw, Xr, Yr ).
Xw = 800 ,
Yw = 600 ,
Xr = Yr = 96
```

**Notes**
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "device contexts": these constitute the virtual environment into which all graphics are drawn in Windows. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_rop_handle/2

convert between a rop object and its handle

    gfx_rop_handle( Object, Handle )

?Object     <object_handle> or <variable>
?Handle     <integer> or <variable>

Comments  This converts between a rop (raster operation) object and its raw (integer) handle. The Object parameter may be a handle of the form, "cursor(Raw)", where "Raw" is the raw handle, or a stock object of the form "stock(Name)", where "Name" is the name of an appropriate stock object.

Examples  The following call will return the raw handle for a stock rop object called "stock(xorpen_rop)"

    ?- gfx_rop_handle( stock(xorpen_rop), H ).

    H = 7

Notes  The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx*/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_select/1

`select objects into the current device context`

``
gfx_select( Objects )

?Objects  <term>
``

Comments
This selects one or more objects or transformations into the current graphics device context.

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

```
foo :-
    wcreate( foo, grafix, ``, 10, 10, 100, 100, 16'80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).
```

The window handler has just one clause, which reacts to "msg_paint" messages by getting a device context, selecting a brush, drawing a circle, and releasing the device context:

```
foo( foo, msg_paint, _, _ ) :-
    gfx_paint( foo ),
    gfx_select( brush = stock(gray_brush) ),
    gfx( ellipse(10,10,90,90) ),
    gfx_end( foo ).
```

Once compiled, the following command will display a small window which contains a grey circle:

```
?- foo.
```

Notes
The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx'/n" because virtually all their names begin with the letters "gfx". Many of these, including the present predicate, are concerned with the management of "objects": these are named entities that describe colours, textures, images and so on. Certain objects, such as "pens" and "brushes", are selected within graphics procedures to change the appearance of graphics drawn by subsequent "primitive" operations; others, such as "bitmaps" and "icons", are drawn directly as primitives. Full details of objects, primitives and graphics procedures can be found in Appendix E.
gfx_test/1

perform a hit test on the current device context

gfx_test( Hits )

?Hits <integer> or <variable>

Comments  This returns the number of hits so far on the current device context. It can only be called when the device context has been explicitly set up for hit tests (see gfx_begin/3).

Examples   The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

 foo :-
    wcreate( foo, grafix, ` `, 10, 10, 100, 100, 16`80800000 ),
    window_handler( foo, foo ),
    wshow( foo, 1 ).

    foo( foo, msg_paint, _, _ ) :-
        gfx_paint( foo ),
        grafix,
        gfx_end( foo ).

    foo( foo, msg_mousemove, (X,Y), close ) :-
        gfx_begin( foo, X, Y ),
        grafix,
        gfx_test( Hits ),
       gfx_end( foo, X, Y ),
        Hits = 1.
The following simple graphics routine is used by both the "msg_paint" and "msg_mousemove" cases of the window handler, foo/4:

```prolog
grafix :-
    gfx( (brush = stock(gray_brush) -> ellipse(10,10,90,90)) ).
```

Once compiled, the following command will display a small square window which contains a grey circle:

```
?- foo.
yes
```

The window, complete with its circle, will remain on the screen until the user moves the mouse pointer over the circle. At this point, the window will suddenly vanish.

**Notes**

The WIN-PROLOG "GraFiX subsystem" comprises a complete set of predicates, generically referred to as "gfx/*/n" because virtually all their names begin with the letters "gfx". Of these, one predicate in particular, namely gfx/1, supports a special sub-language that enables complex graphical operations to be easily represented as a series of "object" selections and "primitive" operations. Details of this sub-language can be found in Appendix E.
gfx_transform/4

convert between logical and device coordinates

gfx_transform( Xlog, Ylog, Xdev, Ydev )

?Xlog    <integer> or <variable>
?Ylog    <integer> or <variable>
?Xdev    <integer> or <variable>
?Ydev    <integer> or <variable>

Comments
This converts a pair of "logical coordinates" to "device coordinates", or vice-versa, with respect to the current graphics device context. If Xlog and Ylog are integers specifying logical coordinates and Xdev and Ydev are variables, the latter are bound to the corresponding "X" and "Y" device coordinates; if Xdev and Ydev are integers specifying device coordinates and Xlog and Ylog are variables, the latter are bound to the corresponding "X" and "Y" logical coordinates.

Examples
The following program creates a "grafix" window, and attaches a window handler to it before showing the window:

    foo :-
        wcreate( foo, grafix, ``, 10, 10, 100, 100, 16`80800000 ),
        window_handler( foo, foo ),
        wshow( foo, 1 ).

The window handler has just one clause, which reacts to "msg_mousemove" messages by getting an unclipped device context, applying a coordinate mapping, drawing a squashed grey ellipse between the mouse cursor and the transformed points (90,90), and then releasing the device context:

    foo( foo, msg_mousemove, (X,Y), _ ) :-
        gfx_begin( foo ),
        gfx_mapping( 3, 3, 2, 1 ),
        gfx_transform( Xl, Yl, X, Y ),
        gfx( (brush = stock(gray_brush) -> ellipse(Xl,Yl,90,90)) ),
        gfx_end( foo ).

Once compiled, the following command will display a small window which will draw squashed images of a grey circle as the mouse moved across the window:
Notes

When a graphics device context is obtained through a call to `gfx_begin/1` or `gfx_paint/1`, it is initialised with an origin of (0,0) and a mapping of (0,0,0,0), resulting in a 1:1 relationship between logical and device coordinates. Two predicates, `gfx_origin/2` and `gfx_mapping/4`, can be used to modify the origin and mapping respectively, allowing graphics to be repositioned and scaled efficiently.

It is normally possible to write graphics code without regard to origins and mappings: programs simply set up and work to a convenient coordinate range, while allowing the resulting images to be zoomed and scrolled with ease.

When relating such transformed graphics to "external" items, such as the mouse cursor or perhaps the size of another window, it is necessary to think about logical and device coordinates. For example, the mouse cursor position is always reported in "device coordinates", relative to the top left corner of a window, because the mouse has no concept of origins and mappings. If (as in the somewhat contrived example above) it is desired to track the mouse within a remapped window, calls must be made to `gfx_transform/4` to convert the mouse (device) coordinates into logical coordinates before using them in a call to `gfx/1`.

This predicate is not strictly part of the WIN-PROLOG "GraFiX subsystem": rather it is a direct window handling predicate whose functionality is limited to "grafix" windows. Details of grafix windows, and indeed graphics in general, can be found in Appendix E.
gfx_window_cursor/2

set the cursor for the given grafix window

gfx_window_cursor( Window, Cursor )

+Window      <window_handle>
+Cursor      <object_handle>

Comments     This associates the specified Cursor handle with the given "grafix" Window, so that this cursor is displayed whenever the mouse pointer moves over the window.

Examples      The following call will cause the "wait" ("hourglass") cursor to be displayed whenever the mouse pointer moves over the grafix window "foo":

    ?- gfx_window_cursor( foo, stock(wait_cursor) ).
    yes

Notes         This predicate is not strictly part of the WIN-PROLOG "GrafiX subsystem": rather it is a direct window handling predicate whose functionality is limited to "grafix" windows. Details of grafix windows, and indeed graphics in general, can be found in Appendix E.
gfx_window_redraw/5

force a selective repaint of given grafix window

\[
gfx\_window\_redraw( \text{Window}, \text{Left}, \text{Top}, \text{Right}, \text{Bottom} )
\]

Comments
This invalidates a rectangular portion of the given "grafix" Window, specified by the given Left, Top, Right and Bottom device coordinates, causing the window to be sent a "msg_paint" message that will result in a correctly written window handler redrawing this part of the window.

Examples
The following call will invalidate part of the client area of a grafix window "foo", causing it to receive a "msg_paint" message:

\[
?- \text{gfx}\_\text{window}\_\text{redraw}( \text{foo}, 100, 100, 200, 300 ).
\]

Notes
Previous versions of WIN-PROLOG included a simpler arity-one version of this predicate, which always invalidated the entire client area of a given "grafix" window: this was rather a blunt instrument, and the present predicate offers considerably more flexibility. However, if it is wished to emulate the original predicate, the following code will suffice:

\[
gfx\_window\_redraw( \text{Window} ) :-
gfx\_window\_redraw( \text{Window}, 0, 0, \text{Width}, \text{Depth} ).
\]

This predicate is not strictly part of the WIN-PROLOG "GraFiX subsystem": rather it is a direct window handling predicate whose functionality is limited to "grafix" windows. Details of grafix windows, and indeed graphics in general, can be found in Appendix E.
gfx_window_scroll/3

scroll the given grafix window

gfx_window_scroll( Window, X, Y )

+Window          <window_handle>
+X               <integer>
+Y               <integer>

Comments: This scrolls the contents of the given "grafix" Window by the given horizontal (X) and vertical (Y) amounts, specified in device coordinates (pixels). Next, it invalidates the portion of the "client area" that has been exposed, causing the window to be sent a "msg_paint" message that will result in a correctly written window handler redrawing these parts.

Examples: The following call will scroll the client area of the grafix window "foo" right by 20 pixels and up by 30 pixels:

?- gfx_window_scroll( foo, 20, -30 ).

yes

Notes: This predicate is not strictly part of the WIN-PROLOG "GraFiX subsystem": rather it is a direct window handling predicate whose functionality is limited to "grafix" windows. Details of grafix windows, and indeed graphics in general, can be found in Appendix E.
grab/1

check for a character directly from the keyboard or mouse

\texttt{\textbf{grab}(\textbf{Char})}

- \textbf{Char} \quad \textbf{<variable>}

Comments

This attempts to input a single \texttt{Char} directly from the keyboard or mouse, returning the integer character code for any key pressed, or a negative code representing a mouse button, as shown in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>left button pressed</td>
</tr>
<tr>
<td>-2</td>
<td>right button pressed</td>
</tr>
<tr>
<td>-4</td>
<td>middle button pressed</td>
</tr>
</tbody>
</table>

Examples

The following call continually outputs the "escargot" character (character code 40h) until a key is pressed, in this case the space bar; note that it is not necessary to press <\texttt{enter}> to complete the data input:

\texttt{?- repeat, putb( 64 ), grab( B ), nl, !.}

\texttt{@@@@@@@@@<space>}

\texttt{B = 32}

Notes

This predicate is one of a special family of three which is principally aimed at helping the debugging of applications. Each of \texttt{getb/1}, \texttt{grab/1} and \texttt{putb/1} performs its input or output directly from or to the lowest-level of user I/O.

The \texttt{getb/1} will return immediately if a character is stored in the keyboard or mouse type-ahead buffer; otherwise, it will suspend program execution until a key or mouse button is pressed. The related \texttt{grab/1} predicate also returns immediately any character stored in type-ahead, but simply fails if nothing is there: this allows programs to peek into type-ahead without risking being suspended if nothing is present. Finally, \texttt{putb/1} outputs the given byte directly to the console window, irrespective of whichever output stream is currently set.
ground/1

test whether a term is a ground term

ground( Term )

?Term <term>

Comments
This succeeds if Term is ground, which means that it is currently bound to a term containing no unbound variables.

Examples
The following calls test various cases:

?- ground( 123 ). <enter>
yes

?- ground( [1,2,3] ). <enter>
yes

?- ground( foo ). <enter>
yes

?- ground( bar(X) ). <enter>
no

?- ground( X ). <enter>
no

Notes
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

The term "ground" means non-variable and not containing any unbound variables. Such terms cannot be further instantiated through normal unification processes, although they may still be processed by standard recursive programs.
halt/0

exit to the operating system

halt

Comments This predicate performs an assortment of checks, and if these are successful, terminates the current Prolog session, exiting to the operating system and passing an exit code of zero (0).

Examples If you wanted to terminate the current Prolog session, you could type the following command (you will be offered the chance to save any unsaved program files):

```prolog
?- halt.
```

Notes Another predicate, exit/1, performs much the same job as either halt/0 or halt/1, the only difference being that it is a little more direct. Both the halt/n predicates perform an assortment of checks before finally closing a Prolog session: within the development environment, for example, they both check for unsaved program files, and offer the chance to save them. As another example, when testing applications with the "Run/Application" menu item (Developer Edition only), both halt/n predicates are intercepted, so that your application "returns" to the development environment upon completion.

In effect, the two halt/n predicates are written in terms of each other and of exit/0 much as follows:

```prolog
halt :-
    halt( 0 ).

halt( Code ) :-
    ( ... check if testing application ...
    -> true
    ; ... see if any files need saving ...
    -> system_menu( 1, file, save_all ),
        exit( Code )
    ; exit( Code )
    ).
```

Because exit/1 always exits directly, without offering to save files, it should be used with care: halt/0 and halt/1 are recommended for more general usage.
halt/1

exit to the operating system with a return code

\[ \text{halt( Code )} \]

+Code: integer

Comments

This predicate performs an assortment of checks, and if these are successful, terminates the current Prolog session, exiting to the operating system and passing it the given exit Code.

Examples

If you wanted to terminate the current Prolog session, you could type the following command (you will be offered the chance to save any unsaved program files):

\[ ?- \text{halt( 0 ).} \]

Notes

Another predicate, exit/1, performs much the same job as either halt/0 or halt/1, the only difference being that it is a little more direct. Both the halt/n predicates perform an assortment of checks before finally closing a Prolog session: within the development environment, for example, they both check for unsaved program files, and offer the chance to save them. As another example, when testing applications with the "Run/Application" menu item (Developer Edition only), both halt/n predicates are intercepted, so that your application "returns" to the development environment upon completion.

In effect, the two halt/n predicates are written in terms of each other and of exit/0 much as follows:

\[
\begin{align*}
\text{halt} & : - \\
& \text{halt( 0 ).}
\end{align*}
\]

\[
\begin{align*}
\text{halt( Code )} & : - \\
& \{ \ldots \text{check if testing application} \ldots \} \\
& \rightarrow \text{true} \\
& \; \ldots \text{see if any files need saving} \ldots \\
& \rightarrow \text{system_menu( 1, file, save_all ),} \\
& \; \text{exit( Code )} \\
& \; \text{exit( Code )} \\
& ).
\end{align*}
\]
Because `exit/1` always exits directly, without offering to save files, it should be used with care: `halt/0` and `halt/1` are recommended for more general usage.
hash/1

compile an existing predicate with the hashing compiler

\[
\text{hash( Pred )}
\]

+Pred <pred_spec>

Comments

This invokes the hashing compiler to compile an existing predicate, that has previously been compiled by the incremental compiler. When files are "consulted" (loaded with consult/1), their predicates are incrementally compiled: hashing such predicates improves their run-time performance.

Examples

The following command loads one of the example programs, "salesman.pl", which includes a data predicate, dist/3:

\[
\text{?- consult( examples(salesman) ).}
\]

# 0.000 seconds to consult salesman.pl [c:\tempwin\examples]
yes

The dist/3 predicate can be hashed as follows:

\[
\text{?- hash( dist/3 ).}
\]

% 0.000 seconds to hash predicate dist / 3
% Quota = 100, Count = 300, Cases = 24, Total = 6, Worst = 3
yes

Notes

The hash/1 predicate can only hash programs which are currently incrementally compiled, and furthermore are defined neither as "dynamic" (see dynamic/1) nor "multifile" (see multifile/1). Both these latter classes of predicate need to remain incrementally compiled, so that clauses can be added to or removed from their definitions when required. If an attempt is made to optimise anything other than a static, incrementally compiled predicate, an error will be signalled.

Code compiled with the hashing compiler can still be debugged and listed, and can return matches many times faster then code compiled with either the incremental or optimising compilers, although with some restrictions. Depending upon the setting of the "hash_files" Prolog flag, hashing may be applied automatically after loading a source file. Other aspects of hashing are controlled by the "hash_limit" and "hash_quota" Prolog flags, as described in Appendix N.
hash/3

**compile a predicate with the hashing compiler with settings**

\[
\text{hash( Pred, Quota, Limit )}
\]

+Pred <pred_spec>
+Quota <integer>
+Limit <limit>

**Comments**
This invokes the hashing compiler to compile an existing predicate, that has previously been compiled by the incremental compiler, using the given Quota and Limit settings (see Appendix N for a detailed explanation).

**Examples**
The following command loads one of the example programs, "salesman.pl", which includes a data predicate, \texttt{dist/3}:

\[?reak consult( examples(salesman) ).\]

\# 0.000 seconds to consult salesman.pl [c:\tempwin\examples]

\texttt{yes}

The \texttt{dist/3} predicate can be hashed as follows:

\[?reak hash( dist/3, 100, 250 ).\]

\% 0.000 seconds to hash predicate dist / 3
\% Quota = 100, Count = 300, Cases = 24, Total = 6, Worst = 3

\texttt{yes}

**Notes**
The \texttt{hash/3} predicate can only hash programs which are currently incrementally compiled, and furthermore are defined neither as "dynamic" (see dynamic/1) nor "multifile" (see multifile/1). Both these latter classes of predicate need to remain incrementally compiled, so that clauses can be added to or removed from their definitions when required. If an attempt is made to optimise anything other than a static, incrementally compiled predicate, an error will be signalled.

Code compiled with the hashing compiler can still be debugged and listed, and can return matches many times faster then code compiled with either the incremental or optimising compilers, although with some restrictions. Depending upon the setting of the "hash_files" Prolog flag, hashing may be applied automatically after loading a source file. See \texttt{hash/1} for further information.
**help/1**

*display a help topic or test for help installation*

\[ \text{help( Term )} \]

?Term <term>

**Comments**

This predicate displays help about a specific topic, or tests for the installation of the **WIN-PROLOG** help files. If the given `Term` is a variable, this predicate will return the full pathname of the main help index file (HELP_IDX.HTM) if it exists; otherwise, it will fail. If `Term` is not a variable, and a matching help file exists, this is displayed using a default web browser; if not, the browser will display the main help index file (HELP_IDX.HTM).

**Examples**

The following command displays help about the `fread/4` predicate, assuming the help files are installed in the **WIN-PROLOG** "HELP" directory:

\[ \text{?- help( fread/4 ).} \]

yes <enter>

**Notes**

With the advent of Windows Vista, support for the old Windows Help system, which was accessed through the "WinHelp" API, has been dropped by Microsoft: starting with version 4.7, **WIN-PROLOG** now uses HTML files and a web browser to store and display help.

Two help predicates are built into **WIN-PROLOG**: the first, `help/1`, attempts to locate a help file matching a given term, and if found, displays the file using the computer's default web browser; if the indicated file is not present, the main index file, HELP_IDX.HTM, is displayed instead. The second predicate, `help/2`, provides a mapping between a given term and the name of its help file, allowing programs to locate and handle these files explicitly.

Each help topic is self-contained within a single HTML file, stored in **WIN-PROLOG**'s "HELP" subdirectory; any HTML file within this directory can become part of the help subsystem, provided it is correctly named with respect to its subject. In order to add help for any given term, the user can call `help/2` to generate the appropriate file name, and then write some HTML (or even plain text) and save it with the given name.
help/2

convert a help topic into a file name

help( Term, File )

+Term <term>
-File <variable>

Comments  This predicate converts the given Term into a help file name, which is computed by hashing the text of the term, and appending it to the pathname of the WIN-PROLOG "HELP" directory; as a special case, if Term is the empty list ([]), the name of main help index file (HELP_IDX.HTM) is returned.

Examples  The following command returns the name of the file that should contain help about the fread/4 predicate:

?- help( fread/4, File ).
File = 'c:\pro386w\help\t0akqjxt.htm'

Notes  With the advent of Windows Vista, support for the old Windows Help system, which was accessed through the "WinHelp" API, has been dropped by Microsoft: starting with version 4.7, WIN-PROLOG now uses HTML files and a web browser to store and display help.

Two help predicates are built into WIN-PROLOG: the first, help/1, attempts to locate a help file matching a given term, and if found, displays the file using the computer's default web browser; if the indicated file is not present, the main index file, HELP_IDX.HTM, is displayed instead. The second predicate, help/2, provides a mapping between a given term and the name of its help file, allowing programs to locate and handle these files explicitly.

Each help topic is self-contained within a single HTML file, stored in WIN-PROLOG's "HELP" subdirectory; any HTML file within this directory can become part of the help subsystem, provided it is correctly named with respect to its subject. In order to add help for any given term, the user can call help/2 to generate the appropriate file name, and then write some HTML (or even plain text) and save it with the given name.
hide/2

hide or unhide an atom

\texttt{hide( Atom, Mode )}

\texttt{?Atom} \hspace{1cm} \texttt{<atom> or <variable>}
\texttt{+Mode} \hspace{1cm} \texttt{<integer> in the domain \([0,1]\)}

Comments

This predicate is used to "hide" or "unhide" an atom, and is used as the basis of the \texttt{WIN-PROLOG} module system. Given an \texttt{Atom}, if \texttt{Mode} is specified as "1", that atom is marked so that it can no longer be unified in input: in other words, it is "hidden". Provided no other unhidden atom of the same name exists, a given \texttt{Atom} can also be unhidden by specifying a \texttt{Mode} of "0". If \texttt{Atom} is an unbound variable, a randomly-generated, guaranteed unique atom is returned, either hidden or unhidden, depending upon the value of \texttt{Mode}.

Examples

The details of hidden and unhidden atoms are complex, but the following example shows some of the bevaviour; enter the following commands:

\texttt{?- assert( foo(fred) ).} \hspace{1cm} \texttt{<enter>}
\texttt{yes}

\texttt{?- foo( fred ).} \hspace{1cm} \texttt{<enter>}
\texttt{yes}

All we have done is assert a clause, "foo(fred)", and then call it to confirm that it is present. Now enter the command:

\texttt{?- hide( fred, 1 ).} \hspace{1cm} \texttt{<enter>}
\texttt{yes}

This will "hide" the atom "fred", so that it will no longer be recognised on input; enter the following commands:

\texttt{?- foo( fred ).} \hspace{1cm} \texttt{<enter>}
\texttt{no}

\texttt{?- foo( X ).} \hspace{1cm} \texttt{<enter>}
\texttt{X = fred}
Although "X" still returns an atom spelled "fred", this atom is no longer unified with a newly-created atom of the same spelling.

A more obviously useful purpose for this predicate is as a symbol generator: often if is necessary to create a random name, perhaps for use as a temporary file, or perhaps a predicate name, or just as a unique identifier. The hardest part of doing this is guaranteeing that the new name is unique, and fortunately, hide/2 can perform this function quickly and efficiently; enter the command:

```prolog
?- hide( Atom, 0 ).
Atom = sachoysu
```

Each time hide/2 is called in this way, an atom is create which is guaranteed not to exist elsewhere in the current instance of WIN-PROLOG.

**Notes**

A built-in pseudo-random number generator (PRANG) is used to generate each successive letter in the 8-character (X86) or 16-character (X64) names returned by hide/2; each integer is scaled in the range 0..25, and then added to the ASCII value for lower-case "a", and then an existing atom of the resulting values is searched for. If it is found, the process starts again: because of the very random nature of the letters, most often, the first attempt generates a unique name. The random, unique lowercase, 8/16-character names generated by hide/2 are used throughout the WIN-PROLOG development environment for temporary file names, buffer names, and so on: sometimes, you will see these names being reported in error messages.

Please note that is is possible (though highly unlikely) for a given atom to be generated more than once during a single WIN-PROLOG session, but only if all references to the first instance have been lost and a full text garbage collection has taken place (see gc/1) since the atom's original creation. To guarantee entirely that such a name cannot occur twice, it is necessary simply to maintain at least one reference to it, for example by asserting a simple fact that lists the atom:

```prolog
gensym( Atom ) :-
    var( Atom ),
    hide( Atom, 0 ),
    assert( in_use(Atom) ).
```

There is no need ever to call the in_use/1 predicate: the simple existence of a clause containing the newly generated atom will be sufficient to guarantee that it cannot occur again in the current WIN-PROLOG session.

It should be noted that hidden atoms have a number of potentially confusing properties. For example, they cannot be unified with any atom newly created during input, even if they have the same spelling; similarly, it is quite possible for two or more identically spelt atoms to occur on the hidden dictionary, while the main dictionary guarantees that any atom will only ever appear once (see dict/2). The hide/2 predicate can be used to transfer atoms between the visible and hidden portions of the dictionary.
hide_dialog/1

hide a modeless dialog

\[ \text{hide_dialog( Window )} \]

+Window <window_handle>

Comments This hides the given dialog \textit{Window}, immediately continuing execution of the calling program. The behaviour of the dialog itself is governed by its attached "window handler".

Examples The following program creates a very simple dialog with two buttons, labelled "Hello" and "World" respectively:

\[
\text{create_foo} :-
\text{Dstyle} = [\text{ws_popup, ws_caption, dlg_ownedbyprolog}],
\text{Bstyle} = [\text{ws_child, ws_visible, bs_pushbutton}],
\text{wdcreate( foo, `foo`, 100, 100, 170, 75, Dstyle )},
\text{wccreate( (foo,1), button, `Hello`, 10, 10, 70, 30, Bstyle )},
\text{wccreate( (foo,2), button, `World`, 90, 10, 70, 30, Bstyle )}.
\]

When compiled, and the following command is entered, the dialog "foo" is created, but not yet displayed:

\[
?- \text{create_foo}. \quad \text{<enter>}
\]

yes

To show the dialog modelessly, make the following call:

\[
?- \text{show_dialog( foo )}. \quad \text{<enter>}
\]

yes

The dialog will appear, with its two buttons: at this point, clicking the buttons will have no effect. To wait for and retrieve a button press, enter the following command:

\[
?- \text{wait_dialog( foo, X )}. \quad \text{<enter>}
\]

If the user presses the "Hello" button, its lowercase text will be returned as follows:
Unlike modal dialogs, which handled by `call_dialog/2`, this dialog remains on the screen even after a button click. It is therefore possible to call the dialog again to retrieve further button clicks:

```prolog
?- wait_dialog( foo, X ).
X = world
```

When it is finally desired to hide the dialog, the following call can be made:

```prolog
?- hide_dialog( foo ).
yes
```

**Notes**

The example shown above uses the default window handler, `window_handler/4`, which simply binds the lowercase text of any clicked button to the `Result`. It is the action of binding this variable which actually "completes" the dialog.

Window handlers are simply arity-4 programs which intercept messages destined for a window (including dialogs and their controls), allowing arbitrary actions to take place. Consider the following example:

```prolog
foo_handler( (foo,1), msg_button, _, Result ) :-
    time( _, _, _, H, M, S, _ ),
    Result = finished_at(H,M,S).

foo_handler( Window, Message, Data, Result ) :-
    window_handler( Window, Message, Data, Result ).
```

The first clause detects that a button called "(foo,1)" has been clicked, and completes the dialog by returning the structure "finished_at(H,M,S)" rather than the atom "hello". All other messages are passed to the default window handler. Once compiled, the new window handler is attached to the dialog window with a call to `window_handler/2`, as shown here:

```prolog
?- window_handler( foo, foo_handler ).
yes
```

If the dialog is invoked once again, and the "(foo,1)" button (labelled "Hello") is pressed, the time will be returned instead of the atom "hello":

```prolog
X = hello
```
?- show_dialog( foo ).
yes

?- wait_dialog( foo, X ).
X = finished_at(10,8,42)

?- hide_dialog( foo ).
yes

Please note that "modal" and "modeless" dialogs have exactly the same window handlers: in fact, in WIN-PROLOG, these two types of dialog are one and the same thing as each other. All that determines whether a dialog behaves in a modal or modeless fashion at run-time is whether it is invoked by call_dialog/2 (modal) or show_dialog/1 (modeless). A third class of dialog, sometimes referred to as a "wizard", behaves semi-modally: like a modal dialog, it waits for input from the user when called with wait_dialog/2, but like a modeless dialog, it remains visible and in focus between successive calls, until finally hidden by invoking hide_dialog/1.
history/1

get or set the console history contents or size

\texttt{\textbf{history( History )}}

?History \text<dvariable>, \text<integer> or \text<list>

Comments

This gets or sets the console command \textit{History} contents or sets its size. When \textit{History} is a variable, it is bound to a list starting with an integer giving the size of the history buffer, followed by zero or more strings containing the stored commands; if \textit{History} is bound to a list of this format, it both sets the size of the history buffer from the initial integer, and loads its contents from the strings in the list. If \textit{History} is an integer, it simply sets the size of the history buffer, otherwise leaving its contents unchanged.

Examples

The following call returns the console history status:

\texttt{?- history( History ).}

\texttt{History = [64,\textquoteleft\texttt{history( History ).}\textquoteleft]}  

The next call turns sets the history buffer to contain up to 255 entries:

\texttt{?- history( 255 ).}

yes

Notes

The "Command History" is a powerful feature of \textbf{WIN-PROLOG}, providing immediate recall of up to 255 previously entered commands; moreover, the history will persist between sessions if the "save_settings" flag is set to "yes" (see \texttt{prolog_flag/3}), making the entering of repeated commands very easy. When saved in this way, the command history is stored in a lightly encrypted file called "pro386w.hst", which can only be loaded by the installed version of \textbf{WIN-PROLOG} that saved it.

To navigate through the entries in the command history, the user simply presses \texttt{<ctrl-up>} and \texttt{<ctrl-down>} to copy successive previous commands into the "input zone", after which they be edited and reentered.
index/2

declare multiple argument indices

\[ \text{index( Predicate, Indices )} \]

+Predicate <pred_spec>
+Indices <list_of <integer>>

Comments
This declares a list of Indices specifying which arguments should be indexed when the given Predicate is compiled using the optimising compiler.

Examples
Consider the following program:

\[
\begin{align*}
&\text{foo( 1, 2 ).}\hfill \\
&\text{foo( 3, 4 ).}\hfill \\
&\text{foo( 5, 6 ).}\hfill \\
&\text{:- index( bar/2, [1,2] ).}\hfill \\
&\text{bar( 1, 2 ).}\hfill \\
&\text{bar( 3, 4 ).}\hfill \\
&\text{bar( 5, 6 ).}\hfill \\
\end{align*}
\]

The definitions of foo/2 and bar/2 are identical: however, an in-line call to index/2 has specified that the latter predicate should be indexed on both its arguments. If compiled with the optimising compiler, these two predicates behave slightly differently when called with their first argument unbound and second argument bound: foo/2 leaves a choicepoint, signified by the need to press <space> before finding there are no more solutions, while bar/2 returns immediately to the prompt:

\[
\begin{align*}
&\text{?- foo( A, 4 ).} \hfill <\text{enter}> \\
&A = 3 ; \hfill <\text{space}> \\
&\text{no} \hfill <\text{space}> \\
&\text{?- bar( A, 4 ).} \hfill <\text{enter}> \\
&A = 3 \hfill <\text{enter}> 
\end{align*}
\]
This can be shown even more explicitly by utilising `call/2` to report the "exit ports" of the respective calls; "exit" means successful but with a pending choicepoint, while "done" means successful with no remaining alternatives:

```
?- call( foo(A,4), P ).  
A = 3 ,  
P = true ;

A = _ ,  
P = fail

?- call( bar(A,4), P ).  
A = 3 ,  
P = !
```

Notes

All programs, whether compiled by the incremental compiler or the optimising compiler, are normally indexed on their first argument. This means that clauses are potentially selected only if the type, and sometimes the value, of their first argument matches that of the call. With the `index/2` predicate, this indexing can be extended to other arguments, but only when predicates are compiled with the optimising compiler.

Indexing is a fairly complex subject, and a full understanding of its features and behaviour can enable the user to write efficient, deterministic code with minimal use of the "cut" (/0). See the Windows Programming Guide for further information about indexing.
initialization/1

declare a goal to be run on loading a file

initialization( Goal )
+Goal <goal>

Comments
This provides a mechanism to embed a Goal in a source or object file, so that it is executed after the file is loaded.

Examples
Consider a file, "foo.pl", containing both an initialisation goal (call to initialization/1) and an inline command:

```prolog
:- initialization write( hello ), nl.
:- write( world ), nl.
foo.
```

If consulted, it will produce output as follows:

```prolog
?- consult( foo ).
world
# 0.000 seconds to consult foo.pl [c:\pro386w\]
hello
yes
```

Notice that the inline command, "write( world ), nl", has been executed during the consulting of the file, while the initialisation goal, "write( hello ), nl", was executed after consultation was complete. Now see what happens if you optimise this file:

```prolog
?- optimize_files( foo ).
world
yes
```

Notice that the inline command, "write( world ), nl", was executed during optimisation, but that the initialisation goal, "write( hello ), nl", was not. Now load the optimised file:

```prolog
?- load_files( foo ).
# Abolishing foo.pl [c:\pro386w\]
```
This time, the inline command, "write( world ), nl", was omitted, but that the initialisation goal, "write( hello ), nl", was executed after the file loaded successfully.

Notes
As can be seen in the above examples, initialisation goals and in-line commands are similar in certain respects, but differ in detail. An initialisation goal, specified as an in-line command which calls initialization/1, stores a request to execute the given goal after the file has been loaded. These requests are persistent, and survive the file being compiled. Normal in-line commands are simply executed, on the fly, at the time a file is consulted, and are not "saved" in any sense.

Normally, in-line commands are used to perform compile-time, source code related operations, such as the declaration of operators or indices (see op/3 and index/2), while initialisation goals are used to perform run-time, application related operations, such as the loading of supplementary files (see load_files/1), or initialisation of dialog windows.

Finally, please note that, by default, "initialization" is declared as a powerful prefix operator (1150, fx), which means that it is normally written without parentheses surrounding its arguments (see the example above).
inpos/1

get or set the input stream pointer

\[
inpos( \text{Position} )
\]

?Position <integer>, <float> or <variable>

Comments This gets or sets the Position of the input (read) pointer in the current input stream. When called with an unbound variable, the existing Position is returned as an integer; if called with an integer, the input pointer is set to the given Position.

Examples The following call skips past the fourth character of input, and reads the next three characters into a string, using fread/4 and the input redirection predicate, <-/2:

\[
?- (\text{inpos}(4), \text{fread}(s, 3, 0, S)) \leftarrow \text{`cat and dog'}.\]

\[
S = \text{`and'}
\]

The next call reads the first term from a string, and then returns the new input position:

\[
?- (\text{read}(T), \text{inpos}(P)) \leftarrow \text{`foo. bar'}.\]

\[
T = \text{foo},
\]

\[
P = 5
\]

Notes The inpos/1 and outpos/1 predicates can be used respectively to reposition the input (read) and output (write) pointer for any disk file, memory file or input or output string, but neither can be used on serial devices like the user input/output stream.

Please note that Windows does not support independent read and write pointers in files, so that in any file opened simultaneously for input and output, inpos/1 and outpos/1 will both manipulate the same, read/write pointer.

Starting with WIN-PROLOG 6.000, file sizes are no longer limited to \(2^{32}-1\) bytes in size: "Fat" files of up to \(2^{53}-1\) bytes can be created, and randomly positioned using inpos/1 and outpos/1 for reading and writing respectively. This number equates to 9,007,199,254,740,991 bytes, or just over 9 petabytes (9000 terabytes) in size, and is the current limit since this is the biggest whole number than can be stored and computed with integer precision in a 64-bit floating point number.
get or set the input stream

\texttt{input(Stream)}

\texttt{?Stream} \quad \text{<stream> or <variable>}

Comments

This gets or sets the current input stream. When called with an unbound variable, the existing stream is returned; if called with a stream identifier, this is set as the current input stream. Streams may be specified as integers (for various types of user input), atoms (naming currently open disk or memory files; see \texttt{fcreate/5}), or conjunctions of a string and an integer, specifying an input string and its input pointer. These are summarised in the following table:

<table>
<thead>
<tr>
<th>Stream</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>buffered console (user) input</td>
</tr>
<tr>
<td>1</td>
<td>raw console input (not double buffered)</td>
</tr>
<tr>
<td>2</td>
<td>direct keyboard input (no echo)</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>the name of a currently open disk or memory file</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>a pair of (&lt;string&gt;,&lt;integer&gt;) where the string is &quot;read&quot; like an input device, and the integer specifies the current input (read) pointer</td>
</tr>
</tbody>
</table>

Examples

The following call reads a term from the direct keyboard input device, which does not echo to the console (the characters typed by the user are shown bracketed in "\textit{italics}"):

\texttt{?- input( 2 ), read( T ), input( 0 ).}<enter>
\texttt{<a><b><c><period><space>T = abc}

The next call reads from a string, starting at its sixth character:

\texttt{?- input( `one. two. ` ,5 )}, read( T ), input( 0 ).}<enter>
\texttt{T = two}

Assuming that a file called "foo" has been opened for input (see \texttt{fcreate/5}), the following call will read the next term, say "hello(world)", from this file (note the use of \texttt{input/1} both to preserve the current input stream and set the new one):
```prolog
?- input( C ), input( foo ), read( T ), input( C ).
C = 0 ,
T = hello(world)
```

Notes

The `input/1` predicate provides low-level control over the input stream. It is similar in many ways to the traditional `see/1` and `seeing/1` predicates, although with several more powerful features. Unlike `see/1`, the `input/1` predicate only works with existing files and streams: it will not automatically open a file that is named for the first time. By leaving the opening of files to an explicit predicate, `fcreate/5`, and the positioning of its input (read) pointer to another predicate, `inpos/1`, `input/1` is considerably more transparent than `see/1`.

Every type of input stream supported by WIN-PROLOG, from user devices through disk and memory files to string input, can be correctly managed by `input/1`. Any program that wishes to temporarily switch input streams should simply "bracket" the switch by picking up the current stream, and resetting it when finished. Consider the following example:

```prolog
read_from_stream( Stream, Term ) :-
    input( Current ),
    input( Stream ),
    read( Term ),
    input( Current ).
```

The existing input stream is picked up in the variable "Current", before a switch is made to the desired new input stream, "Stream". A term is then read into "Term", before input is restored to the original stream, "Current".
integer/1

test whether a term is an integer

integer( Term )

?Term <term>

Comments This succeeds if Term is an integer.

Examples The following calls test various cases:

?- integer( 123 ).
yes <enter>

?- integer( 123.45 ).
no <enter>

?- integer( 123.00 ).
yes <enter>

?- integer( 9876543210 ).
no <enter>

?- integer( X ).
no <enter>

Notes The arithmetic evaluator in WIN-PROLOG treats all numbers simply as such: numbers. It automatically converts integers to floats when required, and always attempts to represent the final result as an integer, for example:

?- X is 67.5 * 1.2.
X = 81 <enter>

Notice that the result, "81", is an integer, and not a float. Conversely, it is possible to generate floats when computing with integers:

?- X is 22 / 7.

X = 3.14285714285714

Except when calling built-in predicates, most of which require true integers as their numeric inputs, there is no need to distinguish between integers and floats in WIN-PROLOG.
compute an arithmetic expression

Result is Expression

(Result = <number> or <variable> + Expression = <expr>)

Comments

This evaluates the given arithmetic expression and unifies the solution with Result. The expression may be anything from a simple number to a deeply nested term containing one or more of the subterms, representing functions and operators. These can be split into two groups, the first of which can be applied to any numbers (integer or floating point) and the second of which are limited to use on integers. The following table shows the general functions and operators:

<table>
<thead>
<tr>
<th>Term</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X + Y</td>
<td>adds X to Y</td>
</tr>
<tr>
<td>X - Y</td>
<td>subtracts Y from X</td>
</tr>
<tr>
<td>- X</td>
<td>returns the negative of X</td>
</tr>
<tr>
<td>X * Y</td>
<td>multiplies X by Y</td>
</tr>
<tr>
<td>X / Y</td>
<td>divides X by Y</td>
</tr>
<tr>
<td>X // Y</td>
<td>performs integer division of X by Y, truncating the result towards zero</td>
</tr>
<tr>
<td>X mod Y</td>
<td>computes X modulo Y, where the result has the same sign as Y</td>
</tr>
<tr>
<td>X ^ Y</td>
<td>raises X to the power of Y</td>
</tr>
<tr>
<td>?(X)</td>
<td>computes a linear congruential pseudo random floating point number between zero and X</td>
</tr>
<tr>
<td>@(X)</td>
<td>computes a Marsaglia Zaman pseudo random floating point number between zero and X</td>
</tr>
<tr>
<td>abs(X)</td>
<td>computes the absolute value of X</td>
</tr>
<tr>
<td>acos(X)</td>
<td>computes the arccosine of X (degrees)</td>
</tr>
<tr>
<td>aln(X)</td>
<td>computes the natural antilogarithm of X</td>
</tr>
<tr>
<td>alog(X)</td>
<td>computes the common antilogarithm of X</td>
</tr>
<tr>
<td>asin(X)</td>
<td>computes the arcsine of X (degrees)</td>
</tr>
<tr>
<td>atan(X)</td>
<td>computes the arctangent of X (degrees)</td>
</tr>
<tr>
<td>cos(X)</td>
<td>computes the cosine of X (degrees)</td>
</tr>
<tr>
<td>Term</td>
<td>Function</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fp(X)</td>
<td>computes the fractional part of X (this has the same sign as X)</td>
</tr>
<tr>
<td>int(X)</td>
<td>computes the nearest integer less than or equal to X (truncates towards -infinity)</td>
</tr>
<tr>
<td>ip(X)</td>
<td>computes the integer part of X (truncated towards zero; this has the same sign as X)</td>
</tr>
<tr>
<td>ln(X)</td>
<td>computes the natural logarithm of X</td>
</tr>
<tr>
<td>log(X)</td>
<td>computes the common logarithm of X</td>
</tr>
<tr>
<td>max(X,Y)</td>
<td>computes the maximum of X and Y (the nearest to +infinity)</td>
</tr>
<tr>
<td>min(X,Y)</td>
<td>computes the minimum of X and Y (the nearest to -infinity)</td>
</tr>
<tr>
<td>pi(X)</td>
<td>computes the value of X multiplied by the mathematical constant, pi</td>
</tr>
<tr>
<td>rand(X)</td>
<td>computes a linear congruential pseudo random floating point number between zero and X</td>
</tr>
<tr>
<td>sign(X)</td>
<td>computes the sign of X (-1, 0, or 1 for negative, zero or positive respectively)</td>
</tr>
<tr>
<td>sin(X)</td>
<td>computes the sine of X (degrees)</td>
</tr>
<tr>
<td>sq(X)</td>
<td>computes the square of X</td>
</tr>
<tr>
<td>sqrt(X)</td>
<td>computes the square root of X</td>
</tr>
<tr>
<td>tan(X)</td>
<td>computes the tangent of X (degrees)</td>
</tr>
</tbody>
</table>

The following table shows the bit manipulation operators, all of which are specific to 32-bit (X86) or 64-bit (X64) signed integers, in the ranges -(2 ^ 31) . (2 ^ 31)-1 and -(2 ^ 63) . (2 ^ 63)-1 respectively:

<table>
<thead>
<tr>
<th>Term</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>I \ J</td>
<td>computes the bitwise &quot;and&quot; I and J</td>
</tr>
<tr>
<td>I V J</td>
<td>computes the bitwise &quot;inclusive or&quot; I and J</td>
</tr>
<tr>
<td>I &lt;&lt; J</td>
<td>computes the arithmetic shift left of I by J bits (vacated bits are cleared to zero)</td>
</tr>
<tr>
<td>I &gt;&gt; J</td>
<td>computes the arithmetic shift right of I by J bits (msb is propagated into the vacated bits)</td>
</tr>
<tr>
<td>(I)</td>
<td>computes the bitwise &quot;not&quot; of I</td>
</tr>
<tr>
<td>a(I,J)</td>
<td>computes the bitwise &quot;and&quot; I and J (this is the same as &quot;I \ J&quot;)</td>
</tr>
<tr>
<td>l(I,J)</td>
<td>computes the left rotation of the I by J bits</td>
</tr>
<tr>
<td>o(I,J)</td>
<td>computes the bitwise &quot;inclusive or&quot; I and J (this is the same as &quot;I V J&quot;)</td>
</tr>
<tr>
<td>r(I,J)</td>
<td>computes the right rotation of the I by J bits</td>
</tr>
<tr>
<td>x(I,J)</td>
<td>computes the bitwise &quot;exclusive or&quot; I and J</td>
</tr>
</tbody>
</table>
Examples

The following calls show a variety of expressions being evaluated or tested:

?- X is 2 + 2.
X = 4
<enter>

?- 5 is 10 / 2.
yes
<enter>

?- X is 22 / 7.
X = 3.142857142857143
<enter>

?- X is asin(sin(45) * cos(60) / tan(75)).
X = 5.436029972077099
<enter>

Notes

The `is/2` predicate is the primary "maths engine" in Prolog programs, being used for everything from the simple incrementing of a counter through to complex arithmetical computations. In addition to the basic set of standard arithmetic operators, WIN-PROLOG includes a full set of "scientific calculator" functions, covering logarithms, linear trigonometry, truncation and rounding functions. For simulation work it also includes a carefully-researched pseudo random number generator (see `seed/1` for further information), and finally a set of bit-oriented integer manipulation operators.

There are various things to note about arithmetic handling in WIN-PROLOG: firstly, there is automatic conversion between the integer and floating point data types during computations, and wherever possible, results are converted back to integers upon completion. Consider the following call:

?- X is 2 * 4.5.
X = 9
<enter>

The `is/2` predicate needs to multiply the integer "2" with the floating point number "4.5", and so it converts the former into the floating point value "2.0" prior to the multiplication. Before returning the result "9.0", a check is made to see whether this value can be represented precisely by an integer; in this case it can, so the computed result is converted back into the integer "9", and it is this that is returned.

A significant new feature in WIN-PROLOG 5.000 is the replacement of an old, bought-in floating point support package, with a brand new one written from the ground up by Brian D Steel, originally for his utility programs. This new package is faster and more accurate than its predecessor, allowing an extra digit or so of usable precision, which is reflected in output, which now defaults to 16 significant digits, rather than the 15 of earlier versions of WIN-PROLOG.
Importantly, more function round-trip accurately than before. For example, in WIN-PROLOG prior to 5.000, many functions would introduce small errors when reversed. Consider the call:

```prolog
?- X is asin(sin(10)).
X = 10
```

In an ideal world, the arcsine of the sine of a number, say "10", should be that same number: apparently this is what the above command has successfully computed. Unfortunately, floating point arithmetic is calculated only to a limited precision, and rounding errors can creep in. However, in (say) WIN-PROLOG 4.920, we can show the returned value is not exactly "10":

```prolog
?- X is asin(sin(10)), Y is X - 10.
X = 10 ,
Y = 1.77635683940025E-015
```

In WIN-PROLOG 5.000 and later, this function (and many other cases like it) round-trips correctly, returning an exact result:

```prolog
?- X is asin(sin(10)), Y is X - 10.
X = 10 ,
Y = 0
```

One small difference between WIN-PROLOG 5.000 and earlier versions, is that exponents in large or small numbers are now printed with a lowercase "e", and without leading zeros. So, for example, in WIN-PROLOG 4.920:

```prolog
?- X is 2 ^ 81 .
X = 2.41785163922923E024
```

But in WIN-PROLOG 5.000, the output contains an additional digit of mantissa, a lowercase "e", and no leading zero on the exponent, as well as being more accurate:

```prolog
?- X is 2 ^ 81 .
X = 2.417851639229258e24
```

It is important to note that is/2 itself does not apply any rules of "precedence" when computing expressions, and nor does it need to. When an expression is read by Prolog, it is the declaration of operators (see op/3) which defines the term's structure. The is/2 predicate simply recurses over whatever term it is given. If operators like "**" and "++" are redefined, the apparent computation order of expressions can change.
keys/1

return the status of the shift, ctrl, alt and lock keys

    keys( Keys )
    -Keys <variable>

Comments  This returns a 32-bit integer in Keys, whose least significant 8 bits identify which of the <shift>, <ctrl> and <alt> keys are being pressed and which of the <scroll lock>, <num lock>, <caps lock> and <ins> toggles are currently set to "on". The bit mask for each of these keys are listed in the following table:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Bit Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;shift&gt;</td>
<td>3</td>
<td>2'00000011</td>
</tr>
<tr>
<td>&lt;ctrl&gt;</td>
<td>4</td>
<td>2'00000100</td>
</tr>
<tr>
<td>&lt;alt&gt;</td>
<td>8</td>
<td>2'00001000</td>
</tr>
<tr>
<td>&lt;scroll lock&gt;</td>
<td>16</td>
<td>2'00010000</td>
</tr>
<tr>
<td>&lt;num lock&gt;</td>
<td>32</td>
<td>2'00100000</td>
</tr>
<tr>
<td>&lt;caps lock&gt;</td>
<td>64</td>
<td>2'01000000</td>
</tr>
<tr>
<td>&lt;ins&gt;</td>
<td>128</td>
<td>2'10000000</td>
</tr>
</tbody>
</table>

Examples  Where none of the <shift>, <ctrl> and <alt> keys is pressed, and none of the locks is "on", keys/1 returns zero:

    ?- keys( K ).
    K = 0 <enter>

If <scroll lock> is set to "on", but none of the other keys has changed its state, the same call will return this information:

    ?- keys( K ).
    K = 16 <enter>

Notes  The keys/1 predicate is useful when writing debuggers or other applications where it is desirable to control behaviour according to the states of various lock keys. An example can be found in the library file, "trace.pl", whose debugging output is "unleashed" whenever <scroll lock> is set to on.
keysort/2

sort a list of key-value pairs into ascending order

keysort( List1, List2 )

+List1 <list_of (<term> - <term>)>
+List1 <variable> or <list>

Comments  This sorts the list of (key-value) terms in List1 into ascending order according to the standard ordering of terms, and unifies the result with List2.

Examples  The following command sorts the given list of (key-value) terms into ascending order:

?- keysort( [3-the,1-quick,4-brown,2-fox], S ).
S = [1 - quick,2 - fox,3 - the,4 - brown]

Notes  Unlike sort/2, the keysort/2 predicate does not remove duplicate entries from the sorted list. It also has the theoretical restriction that it should be used only with lists every one of whose elements is of the form shown below, although this is not strictly enforced in WIN-PROLOG:

key - value

Another sorting predicate, sort/3, provides a considerably more flexible approach to key-based sorting, allowing the user to specify precisely which subterm to use as the sort key on any particular occasion, and without duplicate removal:

?- sort( [3-the,1-quick,4-brown,2-fox], S, [2] ).
S = [1 - quick,2 - fox,3 - the,4 - brown]

?- sort( [3-the,1-quick,4-brown,2-fox], S, [3] ).
S = [4 - brown,2 - fox,1 - quick,3 - the]

See the entry for sort/3 for further information about sorting, duplicate checking, and other features.
**leash/2**

*set leash ports for the debugger*

```prolog
leash( Point, Ports )
```

+Point <atom>
+Ports <atom> or <list_of <atom>>

**Comments**
This sets the "leash ports" that define how the system debuggers interact with programs. The first argument is one of the atoms, "head" or "body", and describes the Point whose leash ports are being set; the second argument may be a single atom or a list of atoms defining the Ports, and comprising one or more of "call", "exit", "redo" and "fail".

**Examples**
The following call leashes just the "exit" and "fail" ports for the "head" point, reducing the number of times the debugger pauses for user input:

```prolog
?- leash( head, [exit,fail] ).
```

# Debugging leash ports for <head> switched to
# exit, fail
yes

**Notes**
By default, all four ports are leashed in both the head and the 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. The fewer the number of ports that are set, the less often the debugger stops for user intervention: using `leash/2` to remove all ports for both the head and body allows the debugger to perform a full trace without any user interaction whatsoever. See `leashed/2` for information about how to retrieve the current leash ports.
leashed/2

get or check the current leash ports for the debugger

leashed( Point, Port )

?Point <atom> or <variable>
?Port <atom> or <variable>

Comments  This gets or checks the "leash ports" that define how the system debuggers interact with programs. The first argument may be a variable or one of the atoms, "head" or "body", and describes the Point whose leash ports are being checked; the second argument may be a variable or one of the atoms, "call", "exit", "redo" and "fail", defining the Port.

Examples  Suppose that the following call has been made to set the "head" point to leash just the "exit" and "fail" ports:

?- leash( head, [exit,fail] ).
# Debugging leash ports for <head> switched to
# exit, fail
yes

The following call will fail, because the specified port is not set:

?- leashed( head, call ).
no

This call will report each of the ports that is set for the head in turn (note that, unlike leash/2, whose second argument may be a list of ports, leashed/2 only returns one port at a time):

?- leashed( head, P ).
P = exit ;
P = fail

Notes  By default, all four ports are leashed in both the head and the 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 leash/2 for further information.
len/2

test or get the length of a term or generate a list

\[
\text{len( Term, Length )}
\]

?Term  <term> or <variable>
?Length <integer> or <variable>

Comments

This is a multiple-use predicate, which can be used both to get or check the "length" of atoms, strings, numbers, lists and tuples, and to generate lists of a given length. When \text{Term} is an unbound variable and \text{Length} is an integer, a list of distinct variables is generated and bound to \text{Term}. When \text{Term} is anything other than an unbound variable, its "length" is computed and unified with \text{Length}. The following table shows how different terms' lengths are computed:

<table>
<thead>
<tr>
<th>Term</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>length is the number of characters that write/1 would use to display the number</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>length is the number of characters in the atom</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>length is the number of characters in the string</td>
</tr>
<tr>
<td>&lt;list&gt;</td>
<td>length is the number of elements in the list</td>
</tr>
<tr>
<td>&lt;tuple&gt;</td>
<td>length is the number of elements in the tuple, including its principle functor</td>
</tr>
</tbody>
</table>

Examples

The following call returns the number of characters in a string:

\[
?- \text{len( `hello world`, L ).} \quad \text{<enter>}
\]
L = 11

This call returns the number of characters that are required to display the number "1.234":

\[
?- \text{len( 001.23400, L ).} \quad \text{<enter>}
\]
L = 5

When testing tuples, the principle functor is included:

\[
?- \text{len( foo(bar), 2 ).} \quad \text{<enter>}
\]
yes
If the first argument is variable and the second is an integer, a list of variables is generated:

?- len( L, 3 ).
L = [_1,_2,_3]

Notes

The `len/2` predicate is written in assembler code, and provides a fast and efficient method for measuring the lengths of a variety of terms, albeit with a number of different interpretations of the meaning of the word, "length". Particularly useful is its ability to measure the print length of a number, or indeed of an unquoted atom or string: this information enables sophisticated layout formatting or graphics, for example, allowing text or numbers to be centred in a space or window:

```prolog
write_centre( [], _ ).
write_centre( [Head|Tail], Width ) :-
    len( Head, Length ),
    Tab is (Width-Length) // 2,
    tab( Tab ),
    write( Head ),
    nl,
    write_centre( Tail, Width ).
```

Once compiled, this program can be used to write out a list of atomic terms centered in a column of the given width:

?- write_centre( [a,`tasty`,0.314159e001], 10 ).
a
tasty
3.14159
yes
length/2

test or get the length of a list or generate a list

\[ \text{length}( \text{List}, \text{Length} ) \]

?List <list> or <variable>
?Length <integer> or <variable>

Comments

This can be used to get or check the length of a list, or to generate lists of a given length. When \textit{Length} is an integer, a list of distinct variables is generated and unified with \textit{List}. When \textit{Length} is an unbound variable, the length of \textit{List} is computed and unified with \textit{Length}. This predicate is non-deterministic, and can generate successive lengths and lists on backtracking.

Examples

When called with two variables, a succession of lists and lengths is generated:

\[ \text{- length}( L, N ) . \]
\[ L = [ ] , \quad N = 0 ; \]
\[ L = [ _1 ] , \quad N = 1 ; \]
\[ L = [ _1, _2 ] , \quad N = 2 \]

Notes

The \textit{length}/2 predicate can be used to test and generate lists, and because it is written in Prolog, it can also handle non-deterministic cases as shown above. When used simply to measure the length of a fully specified list, it is somewhat slower than \textit{len}/2: the latter predicate can also measure the lengths of other data types, including numbers, atoms, strings and tuples.
library_directory/1

*dynamic predicate defining logical library directories*

```prolog
library_directory( Path )
```

Path <file_spec>

**Comments**

This dynamic (and therefore user-modifiable) predicate contains a series of clauses, each of which relates the fixed name, "library", to a logical file Path.

**Examples**

You can see the default definition of `library_directory/1`, and how it is linked into the more general definition of `file_search_path/2` simply by listing it with `listing/1`:

```prolog
?- listing( [library_directory, file_search_path] ).

% library_directory/1

library_directory( prolog('library\') ).

% file_search_path/2

file_search_path( system, prolog('system\') ).

file_search_path( examples, prolog('examples\') ).

file_search_path( files, prolog('files\') ).

file_search_path( library, A ) :-
    library_directory( A ).

file_search_path( temp, A ) :-
    ( env( B ),
      member( ('TEMP', C), B )
    -> stratm( C, A )
```
As can be seen in the listings above, the only significance of `library_directory/1` is that it is called by one of the default clauses in `file_search_path/2`. The former predicate is therefore nothing more than a curious extension of the latter, and of the two, it is recommended to use the latter when defining new logical file names: see the entry for `file_search_path/2` for further information.
list/1

test whether a term is a fully-specified list

\[
\text{list( Term )}
\]

?Term <term>

Comments
This succeeds if \text{Term} is a fully-specified list, which means that it is currently bound to a list all of whose elements are enumerated.

Examples
The following calls test various cases:

\begin{verbatim}
?- list( 123 ).
no

?- list( [1,2,3] ).
yes

?- list( [1,2|X] ).
no

?- list( [] ).
yes

?- list( foo ).
no

?- list( X ).
no
\end{verbatim}

Notes
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

There are two type tests for lists: \text{list/1} and \text{lst/1}. The former succeeds for any term which is a fully-specified list; that is, a list every one of whose elements is enumerated (though not necessarily instantiated), and which therefore does not terminate with "bar notation" (ie, "[...|X]"). The latter succeeds for any list, irrespective of its ending.
listing/0

list all dynamic predicates to the current output stream

listing

Comments
This portrays a listing of all dynamic predicates to the current output stream.

Examples
The following calls create a single clause for each of a pair of dynamic predicates, foo/2 and bar/1, and then display these with a call to listing/0:

?- assert( foo(hello,there) ), assert( bar(world) ).
yes

?- listing.
% foo/2
foo( hello, there ).

% bar/1
bar( world ).
yes

Notes
The listing/0 predicate displays all known dynamic predicates, but no others: the related listing/1 predicate provides additional flexibility, by allowing any incrementally compiled predicate or collection of predicates to be listed.

When source files are loaded by WIN-PROLOG (see consult/1), or program windows are incrementally compiled, the original names of variables are stored alongside the program code: the listing/0 and listing/1 predicates use these original variable names when displaying any such programs.
**listing/1**

*list the specified predicates to the current output stream*

```
listing( List )
+List <atom>, <pred_spec>, <list> or <conjunction>
```

**Comments**
This portrays a listing of one or more *incrementally compiled* predicates, specified in `List`, to the current output stream. The given `List` may be any of the following types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;atom&gt;</td>
<td>list all incrementally compiled predicates of the given name, irrespective of their arity</td>
</tr>
<tr>
<td>&lt;pred_spec&gt;</td>
<td>this takes the form, &quot;atom/integer&quot; (name/arity), and the single specified predicate is listed</td>
</tr>
<tr>
<td>&lt;list&gt;</td>
<td>one or more of the first two options may be combined in a list</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>one or more of the first two options may be combined in a conjunction</td>
</tr>
</tbody>
</table>

**Examples**
Consider the following program, which comprises the four predicates, `foo/1`, `foo/2`, `bar/1` and `bar/2`:

```
foo( one ).
foo( two, three ).
bar( four ).
bar( five, six ).
```

Once compiled (with the *incremental compiler*), both "foo" predicates (`foo/1` and `foo/2`) can be listed as follows:

```
?- listing( foo ).
% foo/1
foo( one ).
% foo/2
```
foo( two, three ).

yes

If desired, the "foo/2" predicate can be also listed by itself:

?- listing( foo/2 ).

% foo/2

foo( two, three ).

yes

The next call lists foo/1 together with both "bar" predicates:

?- listing( [foo/1,bar] ).

% foo/1

foo( one ).

% bar/1

bar( four ).

% bar/2

bar( five, six ).

yes

Notes

The listing/1 predicate is mainly used during program development to examine dynamic predicates; in conjunction with tell/1 and told/0 it also provides a simple way to save programs as source files. When source files are loaded by WIN-PROLOG (see consult/1), or program
windows are incrementally compiled, the original names of variables are stored alongside the program code: the listing/0 and listing/1 predicates use these original variable names when displaying any such programs.

There is a curious anomaly in listing/1: Edinburgh Prolog systems lay great emphasis on the use of "name/arity" predicate specifications. A predicate called "foo/1" is considered to be entirely unrelated to and independent from another called "foo/2"; however, listing/1 allows predicates of different arities to be referred to by a single name!
load_files/1

load the most recent versions of files

load_files( FileSpec )

+FileSpec  <file_specs>

Comments
This predicate loads the programs contained in the disk file or files specified in FileSpec, even if they have already been loaded and have not been modified since. Any source files are loaded with the incremental compiler; any object files are loaded directly. If the file has previously been loaded, and since modified, the existing copy is abolished before reloading.

Examples
The following call loads the example program, "BENCHMRK.PL", into memory:

?- load_files( examples(benchmrk) ).
# 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
yes

Running the same query a second time abolishes the existing copy of the program, and then reloads it:

?- load_files( examples(benchmrk) ).
# Abolishing benchmrk.pl [c:\pro386w\examples\]
# 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
yes

Notes
This predicate is very similar to ensure_loaded/1, the only difference being that the latter only loads files that have not previously been loaded, or which have been modified since they were loaded. Where both source (.PL) and object (.PC) files exist for the same program, both predicates load the most recent, based on the files' timestamps.

Code loaded with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed. See the entries for compile/1 and consult/1 for further information about loading source files, and optimize/1 and optimize_files/1 for information about how to optimise code.

When predicates are loaded from a file, they retain an association with that file: if the file is reloaded, these predicates are automatically abolished. This provides a convenient method for "cleaning up" between successive runs of a system, especially where dynamic predicates have been defined in the source file. Any commands present in the source file are executed on the fly, as they are encountered.
by the loader.

If FileSpec does not specify a file extension, one of the "source" and "object" extensions (normally '.PL' or '.PC' respectively) is assumed, the choice depending upon which of these names the most recent, existing file.

Please note that the current method of maintaining the relationship between a source file and its object file is based purely on the source file's name, minus its extension. If a source file is compiled, and its object file subsequently renamed, then ensure_loaded/1 and load_files/1 will be unable to determine that they are two versions of the same file. Similarly, if source or object files are given non-standard extensions, neither of these predicates will know how to load the "latest" version.
load_files/2

load files according to the specified options

\[ \text{load_files( FileSpec, Options )} \]

+FileSpec \(<\text{file_specs}>\)
+Options \(<\text{list_of <tuple>}>\)

Comments

This predicate loads the programs contained in the disk file or files specified in \text{FileSpec}, according to the given list of \text{Options}. If the file has previously been loaded, and since modified, the existing copy is abolished before reloading. Each option takes the form of a tuple, "name(value)" as described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_dynamic</td>
<td>false</td>
<td>only predicates in file that are so declared are loaded as dynamic</td>
</tr>
<tr>
<td>all_dynamic</td>
<td>true</td>
<td>all predicates in file are loaded as dynamic</td>
</tr>
<tr>
<td>if</td>
<td>changed</td>
<td>only load file if not already loaded, or if changed since it was loaded</td>
</tr>
<tr>
<td>if</td>
<td>true</td>
<td>always load file</td>
</tr>
<tr>
<td>load_type</td>
<td>compile</td>
<td>load source file with optimising compiler</td>
</tr>
<tr>
<td>load_type</td>
<td>latest</td>
<td>load most recent of source or object file of the same name</td>
</tr>
<tr>
<td>load_type</td>
<td>object</td>
<td>load object file directly</td>
</tr>
<tr>
<td>load_type</td>
<td>source</td>
<td>load source file with incremental compiler</td>
</tr>
</tbody>
</table>

Two further options are included for source-file compatibility with Quintus Prolog; please note, however that both are ignored in the current version of \text{WIN-PROLOG}. The compatibility options are shown in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>imports</td>
<td>all</td>
<td>in Quintus, allows a module to import all predicates</td>
</tr>
<tr>
<td>imports</td>
<td>(&lt;\text{pred_specs}&gt;)</td>
<td>in Quintus, allows a module to import given predicates</td>
</tr>
<tr>
<td>must_be_module</td>
<td>false</td>
<td>in Quintus, does not insist that given file is a module</td>
</tr>
<tr>
<td>must_be_module</td>
<td>true</td>
<td>in Quintus, insists that given file is a module</td>
</tr>
</tbody>
</table>

Examples

The following call loads the example program, "BENCHMRK.PL", into memory:
Running the same query a second time abolishes the existing copy of the program, and then reloads it:

?- load_files( examples(benchmrk), [if(true)] ).
# 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
yes

Running a similar query, but with the "if" option changed from "true" to "changed", simply succeeds, because the file has not changed since being loaded:

?- load_files( examples(benchmrk), [if(changed)] ).
yes

Notes

This predicate effectively combines the entire functionality of compile/1, consult/1, ensure_loaded/1, load_files/1 and reconsult/1. In addition, it provides an option to load source files with all predicates implicitly declared dynamic, which can be useful when loading programs from earlier Prolog systems which did not implement a distinction between static and dynamic predicates. Where the "load_type" option is set to "latest", and both source (.PL) and object (.PC) files exist for the same program, load_files/2 loads the most recent, based on the files' timestamps: in this case, source files are loaded using the incremental compiler.

Code loaded with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed. See the entries for compile/1 and consult/1 for further information about loading source files, and optimize/1 and optimize_files/1 for information about how to optimise code.

When predicates are loaded from a file, they retain an association with that file: if the file is reloaded, these predicates are automatically abolished. This provides a convenient method for "cleaning up" between successive runs of a system, especially where dynamic predicates have been defined in the source file. Any commands present in the source file are executed on the fly, as they are encountered by the loader.

If FileSpec does not specify a file extension, one of the "source" and "object" extensions (normally '.PL' or '.PC' respectively) is assumed, the choice depending upon which of these names the most recent, existing file.

Please note that the current method of maintaining the relationship between a source file and its object file is based purely on the source
file's name, minus its extension. If a source file is compiled, and its object file subsequently renamed, then load_files/2 will be unable to determine that they are two versions of the same file. Similarly, if source or object files are given non-standard extensions, neither of these predicates will know how to load the "latest" version.
**lst/1**

**test whether a term is a list**

\[ \text{lst( Term )} \]

?Term \text{<term>}

**Comments**

This succeeds if Term is a list or empty list.

**Examples**

The following calls test various cases:

?- \text{lst( 123 )}.
no

?- \text{lst( [1,2,3] )}.
yes

?- \text{lst( [1,2|X] )}.
X = _

?- \text{lst( [] )}.
yes

?- \text{lst( foo )}.
no

?- \text{lst( X )}.
no

**Notes**

This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

There are two type tests for lists: \text{list/1} and \text{lst/1}. The former succeeds for any term which is a fully-specified list; that is, a list every one of whose elements is enumerated (though not necessarily instantiated), and which therefore does not terminate with "bar notation" (ie, "[...|X]"). The latter succeeds for any list, irrespective of its ending.
lwrupr/2

convert from lower to upper case or vice versa

\[ \text{lwrupr}( \text{Lower}, \text{Upper} ) \]

?Lower <atom>, <string> or <variable>
?Upper <atom>, <string> or <variable>

Comments
This takes an atom or string as one of its arguments, and a variable as the other: if Lower is given and Upper is an unbound variable, then Upper is bound to the upper case equivalent of Lower; conversely, if Lower is an unbound variable and Upper is given, then Lower is bound to the lower case equivalent of Upper.

Examples
The following call converts a given string from mixed case to upper case:

?- lwrupr( `Mixed String`, U ).
U = `MIXED STRING`

The next call converts a given atom from mixed case to lower case:

?- lwrupr( L, 'Mixed Atom' ).
L = 'mixed atom'

Notes
Because lwrupr/2 converts \textit{arbitrarily mixed} case atoms or strings to their lower or upper case equivalents, it is not possible to call it in "checking" mode with both arguments given. Without one or the other being a variable, there would be no way to signal to the predicate whether the first argument should be converted to upper case and compared with the second, or whether the second should be converted to lower case and compared with the first.

The type returned in the output argument depends upon the type given as the input: if an atom is given, then an atom is returned; likewise, if a string is given, a string is returned.

Please note that lwrupr/2 only translates characters from the 7-bit ASCII character set, in other words [A..Z] and [a..z]: upper-half (8-bit) characters are mapped differently in different font encodings: for this reason, such characters are copied without modification in this predicate.
**mclose/1**

*close a MIDI device*

```prolog
mclose( Name )
```

+Name <atom>

**Comments**

This closes a MIDI device of the given `Name`, gracefully terminating any pending events that may be associated with it.

**Examples**

The following call uses `mcreate/4` to create an output MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI synth:

```prolog
?- mcreate( foo, -1, 0, 0 ).
yes
```

Assuming the connection has succeeded, we can send a simple MIDI "Note On" message using `msend/3`:

```prolog
?- msend( foo, 0, (144,60,64) ).
yes
```

This message sends a "Note On" message to channel zero of the MIDI device (144), playing Middle C (60) with medium velocity (64). Depending upon how the MIDI synth is set up, the note may play and then decay (like a piano), or play and sustain (like an organ). In either case, to stop the sound, we should send another "Note On" message to the same channel, specifying the same note (60) but with a velocity of zero:

```prolog
?- msend( foo, 0, (144,60,0) ).
yes
```

Many notes can play simultaneously, on up to 16 different sounds; when it is desired to stop MIDI activity, we can close the MIDI device with `mclose/1`, releasing its resources:

```prolog
?- mclose( foo ).
yes
```

**Notes**

Starting in version 4.700, **WIN-PROLOG** has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access.
to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes mclose/1, mcreate/4, mdata/4, mdict/2, midhdl/2, mrecv/3, msend/3, mstat/3 and mtime/2 together with midi_handler/1 and midi_handler/2; see Appendix Q for a detailed overview.
**mcreate/4**

Create a MIDI device for input or output

```prolog
mcreate( Name, Event, Sysex, Index )
```

+Name <atom>
+Event <integer>
+Sysex <integer>
+Index <integer>

**Comments**

This creates a MIDI device of the given Name, using the given Event and Sysex buffer sizes, on the device specified by the given Index. The value of Event specifies the input or output mode of the MIDI device, as shown in the following table:

<table>
<thead>
<tr>
<th>Event</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>output, buffering is not applicable</td>
</tr>
<tr>
<td>0</td>
<td>input, buffering a default of 8192 events</td>
</tr>
<tr>
<td>&lt;int&gt;</td>
<td>input, buffering given number of events</td>
</tr>
</tbody>
</table>

The meaning of the Sysex is dependent upon whether the MIDI device is used for input or output. For input, it specifies the number of independent one-kilobyte long messages that can be buffered; for output, it specifies the maximum size, in kilobytes, of any one long message:

<table>
<thead>
<tr>
<th>Sysex</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>handle up to 64 messages/kilobytes</td>
</tr>
<tr>
<td>&lt;int&gt;</td>
<td>handle up to given number of messages/kilobytes</td>
</tr>
</tbody>
</table>

**Examples**

The following call uses mcreate/4 to create an output MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI synth:

```prolog
?- mcreate( foo, -1, 0, 0 ).
yes
```

Assuming the connection has succeeded, we can send a simple MIDI "Note On" message using msend/3:
?- msend( foo, 0, (144,60,64) ).
yes

This message sends a "Note On" message to channel zero of the MIDI device (144), playing Middle C (60) with medium velocity (64). Depending upon how the MIDI synth is set up, the note may play and then decay (like a piano), or play and sustain (like an organ). In either case, to stop the sound, we should send another "Note On" message to the same channel, specifying the same note (60) but with a velocity of zero:

?- msend( foo, 0, (144,60,0) ).
yes

Many notes can play simultaneously, on up to 16 different sounds; when it is desired to stop MIDI activity, we can close the MIDI device with mclose/1, releasing its resources:

?- mclose( foo ).
yes

Notes

Once a MIDI device is created successfully, a clock is initialised to count the elapsed milliseconds since its creation (see mtime/2): together with automatically generated midi events, this enables the precise timing of asynchronous recording and playback.

This predicate returns immediately; midi events (see above) are sent to notify of changes in midi device status, and the midi device can be tested explicitly (see mstat/3).

Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes mclose/1, mcreate/4, mdata/4, mdict/2, midhd/2, mrecv/3, msend/3, mstat/3 and mtime/2 together with midi_handler/1 and midi_handler/2; see Appendix Q for a detailed overview.
mdata/4

return data about a MIDI device

mdata( Name, Event, Sysex, Index )

+Name <atom>
-Event <variable>
-Sysex <variable>
-Index <variable>

Comments
This returns the maximum Event and Sysex counts and device Index for the MIDI device of the given Name. For an output device, the Event count is always -1, and the Sysex count represents the size, in kilobytes, available for any one output message; for an input an input device, the counts specify the number of buffered events and sysex messages that can be stored without reading (see mrecv/3) before data is lost. To see the maximum amount of data is actually buffered at a given time, use mstat/3.

Examples
The following call uses mcreate/4 to create an input MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI input:

?- mcreate( foo, 0, 0, 0 ).
    yes

The above call uses the default settings for Event and Sysex buffer sizes, and we can check the actual settings like this:

?- mdata( foo, Event, Sysex, Index ).
    Event = 8192 ,
    Sysex = 64 ,
    Index = 0

Now we can check for the presence of any pending MIDI events on this input channel:

?- mstat( foo, Event, Sysex ).
    Event = Sysex = 0

This result confirms that no data yet exists, and any call to mrecv/3 will fail. Assuming the connection has succeeded, a MIDI source (such as a keyboard) is attached, and that the user has now pressed and released a key, a second call to mstat/3 will indicate the presence of
two events:

?- mstat( foo, Event, Sysex ).
Event = 2 ,
Sysex = 0

In the knowledge that two events are now waiting, we can receive the data:

?- mrecv( foo, Time, Data ).
Time = 1234 ,
Data = (144,60,64)

This returns a "Note On" message from channel zero of the MIDI device (144), indicating that the user played Middle C (60) with medium velocity (64) 1234 milliseconds after the MIDI device was created. Assuming the user has released the key, a second call will return further data:

?- mrecv( foo, Time, Data ).
Time = 5678 ,
Data = (144,60,0)

This "Note On" message shows a velocity of zero (0), which indicates that the user released the key 5678 milliseconds after the MIDI device was created. Assuming no further MIDI events have been generated by the user, a third call will fail:

?- mrecv( foo, Time, Data ).
no

Once we are finished, we can close the MIDI device with mclose/1, releasing its resources:

?- mclose( foo ).
yes

Notes

Two types of MIDI message exist: short and long. Short messages comprise 3 bytes, and are represented in WIN-PROLOG as a conjunction of 3 small integers, and define things like "Note On" events, controller changes and patch changes. Long messages are represented in WIN-PROLOG as a Strings, and contain SysEx (System Exclusive) data, either in whole or in part.

Every MIDI input event, whether it represents a short or a long message, is stored in a buffer, whose size is determined in the call to
mcreate/4: it is the number buffered events that is returned in the Event parameter in mstat/3. Short messages are wholly contained within this buffer, while long messages also consume space in the SysEx buffer, whose size is also determined in the call to mcreate/4. The latter buffer is divided into a series of one-kilobyte (1024 byte) segments, and it is the number of segments currently containing buffered sysex data that is returned in the Sysex parameter of mstat/3. The total amount of buffer space can be determined by calling mdata/4, which, together with mstat/3, simplifies the management of MIDI memory requirements.

Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes mclose/1, mcreate/4, mdata/4, mdict/2, midhdl/2, mrecv/3, msend/3, mstat/3 and mtime/2 together with midi_handler/1 and midi_handler/2; see Appendix Q for a detailed overview.
mdf/3

perform a message digest five on data from current input

\[
\text{mdf( Wanted, Digested, Result )}
\]

- Wanted  <integer>
- Digested <variable>
- Result   <variable>

Comments

This digests up to the number of characters \textit{Wanted}, from the current 8-bit input stream, computes the 128-bit message digest five (MD5) on the data, and returns the number of characters successfully \textit{Digested} as well as the \textit{Result} MD5 message digest as a list of four 32-bit integers.

Examples

The following command hashes all characters in a string, returning its length and MD5:

\[
? - \text{mdf( -1, Count, MDF ) } \leftarrow \text{`Hello World`.

\begin{align*}
\text{Count} &= 11 , \\
\text{MDF} &= [-1316156751,1098244196,-1683376379,-448844057] & \% \text{X86} \\
\text{MDF} &= [2978810545,1098244196,2611590917,3846123239] & \% \text{X64}
\end{align*}
\]

Notes

An input value of "-1" interpreted is interpreted as "process the entire file", using a 53-bit (X86) or 64-bit (X64) counter. Because this predicate digests directly from input, without using Prolog's internal data structures, it can handle large amounts of data very efficiently.

There are three data checking predicates in WIN-PROLOG, providing differing levels of security. The first is \textit{crc/3}, which performs the 32-bit "Cyclic Redundancy Check", or "CRC32", a reliable checksum a given data stream. The other two predicates, \textit{mdf/3} and \textit{sha/3}, perform the 128-bit "Message Digest Five", or "MD5", and 256-bit "Secure Hash Algorithm", or "SHA-256" message digests respectively.

While CRC32 is fine for quick checks, it is not "secure": there is a $2^{-32}$ chance of two data streams having the same CRC32, and it is easy to create files with a given CRC32 value. Meanwhile, MD5 is relatively secure, with only a $2^{-128}$ chance of a clash: it is also not thought possible to design a file with a specific MD5, although recent attacks are making it look less than ideal. For maximum security, use SHA-256, which has only a $2^{-256}$ chance of a collision, and is considered computationally impossible to crack.

Note that \textit{mdf/3} computes the MD5 message digest as a set of 4, 32-bit integers: normally, such digests are represented as a 32-character hexadecimal string, simply by concatenating the little-endian hexadecimal representations of these integers. Consider the following program:
mdf( File ) :-
  fcreate( input, File, 0, 0, 0 ),
  input( Current ),
  input( input ),
  mdf( -1, _, List ),
  input( Current ),
  fclose( input ),
  forall( member( Int, List ),
    ( B0 is Int \ 255,
      B1 is Int >> 8 \ 255,
      B2 is Int >> 16 \ 255,
      B3 is Int >> 24 \ 255,
      fwrite( r, 2, 16, B0 ),
      fwrite( r, 2, 16, B1 ),
      fwrite( r, 2, 16, B2 ),
      fwrite( r, 2, 16, B3 )
    )
  ),
  nl.

This takes as its input the name of a single file, which is opened in "ISO" (8-bit, ISO/IEC 8859-1) mode, and then digested using mdf/3; each of the integers in the resulting digest is then written to the current output stream in little-endian hexadecimal format by extracting the component bytes with is/2, and then using fwrite/4. Suppose we had a text file, "HELLO.TXT", containing just the characters, "hello world~M~J"; once the above program is compiled, the following call will display the conventional MD5 signature of the file:

?- mdf( 'hello.txt' ).
A0F2A3C1DCD5B1C4CAC71BF0C03F2FF1BD
yes

Note that both CRC-32 and SHA-256 values are traditionally output in big-endian format, making direct use of fwrite/4 possible when displaying the results of crc/3 and sha/3 respectively, but MD5 values are conventionally output in little-endian format, requiring some "bit twiddling" to display the output of mdf/3 correctly.
mdict/2

return a list of MIDI devices

\[
\text{mdict}( \text{Flag}, \text{Dict} )
\]

+Flag <integer> in the domain \{-1, 0, 1\}
-Dict <variable>

Comments

This returns a list of all the currently open MIDI devices, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples

The following call uses mcreate/4 to create an output MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI synth:

\[
?- \text{mcreate}( \text{foo}, -1, 0, 0 ).
\]

yes

Now we'll check for a list of all currently open MIDI devices (assuming that "foo" is the only file created so far):

\[
?- \text{mdict}( 0, D ).
\]

D = [foo]

The following call can be used to close all MIDI devices:

\[
?- \text{mdict}( 0, D ), \text{forall}( \text{member}(S,D), \text{mclose}(S) ).
\]

D = [foo],
S = _

Notes

Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes mclosel/1, mcreate/4, mdata/4, mdicel/2, midhdl/2, mrecv/3, msend/3, mstat/3 and mtime/2 together with midi_handler/1 and midi_handler/2; see Appendix Q for a detailed overview.
**mem/3**

*directly access a subterm within a given term*

```prolog
mem( Term, Path, Subterm )
```

+Term <term>
+Path <list_of <integer>>
-Subterm <variable>

**Comments**

This is used to access an arbitrary Subterm, or member, within a more complex term. The given Term may be of any type, and the Path comprises a list of integer indices which define the Subterm to be accessed.

**Examples**

The following call directly returns the third element of a list:

```prolog
?- mem( [one,two,three,four,five], [3], M ).
```

```
M = three
```

Paths can be as long as needed, providing depthwise as well as breadthwise access. In this case, the first number in the path selects the third element of a tuple (this is the tuple’s second argument: the principle functor is its first element), and then selects the second element of this list:

```prolog
?- mem( one(two,[three,four],five), [3,2], M ).
```

```
M = four
```

Using a negative integer in a path returns everything immediately following the indexed subterm:

```prolog
?- mem( [one,two,three,four,five], [-3], M ).
```

```
M = [four,five]
```

**Notes**

The mem/3 predicate is extremely fast, and provides direct access to any part of any compound term, including the head or tail of any point in a list, tuple, conjunction or disjunction. It works directly with the built-in data types of **WIN-PROLOG**, and therefore recognises the principle functor of a tuple as its first element, its first argument as the second element, and so on. With conjunctions and true disjunctions, the first element of the term is its first argument.
**member/2**

*get or check a member of a list*

```
member( Term, List )
```

- **Term** <term>
- **List** <list> or <variable>

**Comments**

This predicate succeeds when its first argument is bound to a *Term* which is a member of the *List* bound to the second argument. The latter may be a fully or partially instantiated list, or simply a variable; *member/2* can backtrack to generate alternative solutions where appropriate.

**Examples**

The following command extracts each of the elements of a given list in turn:

```
?- member( T, [black,and,white]).
```

```
T = black ;
T = and ;
T = white ;
```

no

If *List* is an unbound variable or an incompletely specified list, successive lists are generated on backtracking:

```
?- member( bar, [foo,bar|L] ).
```

```
L = _ ;
L = [bar|_1] ;
L = [_2,bar|_3] ;
```

**Notes**

The *member/2* predicate is a classic Prolog program, and is widely used for extracting elements from lists, or simply checking for their presence: however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of *member/2*. In order to avoid errors, such references must be renamed or removed before loading files into **WIN-PROLOG**.
member/3

get or check a member of a list and its position

member( Term, List, Position )

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

Comments

This predicate succeeds when its first argument is bound to a Term which is a member of the List bound to the second argument, and its third argument is bound to this element's Position. The second argument may be a fully or partially instantiated list, or simply a variable; member/3 can backtrack to generate alternative solutions where appropriate.

Examples

The following command extracts each of the elements of a given list in turn:

?- member( T, [black, and, white], P ).

T = black ,
P = 1 ;

T = and ,
P = 2 ;

T = white ,
P = 3 ;

no

Notes

The member/3 predicate is a special variant of the classic Prolog program, member/2: when it is called with Position as an unbound variable, the present predicate is exactly like its arity-two sibling, supporting all the latter's backtracking and list-generating features. When, however, Position is given as an integer, member/3 becomes deterministic, returning just the given solution and leaving no choicepoints.

Built in to WIN-PROLOG is another predicate, mem/3, which provides very fast, direct access not only to list elements, but also to elements within tuples, conjunctions and disjunctions, even deep within subterms.
member/4

get or check a member of a list and its left and right sublists

\[
\text{member( Term, List, Left, Right )}
\]

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

Comments
This predicate succeeds when its first argument is bound to a Term which is a member of the List bound to the second argument, and the sublists to its Left and Right are bound to the third and fourth arguments respectively. The last three arguments may be a fully or partially instantiated lists, or simply a variable; member/4 can backtrack to generate alternative solutions where appropriate.

Examples
The following command splits a list of atoms at the word "and", backtracking to give further solutions as requested:

?- member( and, [black,and,white,or,red,and,blue], Left, Right ).
Left = [black] ,
Right = [white,or,red,and,blue] ;
Left = [black,and,white,or,red] ,
Right = [blue] ;
no

Notes
The member/4 predicate is a special variant of the classic Prolog program, member/2, but which returns the sublists to the left and right hand sides of the given list member. It is useful in all manner of parsing, shuffling and other related algorithms, and can be used to test for membership or simply to generate solutions, depending on which arguments are bound at the time of the call, and how fully instantiated are any lists.
message_box/3

create a simple message box and return a response

message_box( Buttons, Message, Response )

+Buttons  <atom>
+Message   <term>
?Response  <atom> or <variable>

Comments
This displays a "message box" dialog, containing the given Message and combination of Buttons, and then waits for one of these to be clicked, before unifying its name (one of the atoms, "ok", "cancel", "yes" or "no") with Response. The following table lists the supported button combinations:

<table>
<thead>
<tr>
<th>Name</th>
<th>Buttons</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>OK</td>
</tr>
<tr>
<td>okcancel</td>
<td>OK + Cancel</td>
</tr>
<tr>
<td>yesno</td>
<td>Yes + No</td>
</tr>
<tr>
<td>yesnocancel</td>
<td>Yes + No + Cancel</td>
</tr>
</tbody>
</table>

Examples
The following call creates a message box containing a single "OK" button, together with the given message; the result is only returned once the user clicks on the button or hits <enter>:

?- message_box( ok, value('Pi',3.14159), D ). <enter>
D = ok

Notes
The message_box/3 predicate provides simple access to the standard Windows message box dialog, albeit with less flexibility than its sibling, msgbox/4. The former predicate always displays a fixed title, "WIN-PROLOG" together with the "exclamation" icon; the latter allows the user to specify the title, icon and style, and supports additional button combinations.

Both predicates allow any arbitrary term to be given as the "message": although strings and atoms will be the most common data types used for messages, support for other terms (such as in the example above) simplifies applications such as error handling, or one-off debugging messages. Terms are displayed unquoted, as if output by the write/1 predicate.
user-defined hook to gain control on receipt of a message

'\?MESSAGE?'( Window, Message, Data, Goal )

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt; or &lt;variable&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Message</td>
<td>&lt;integer&gt; or &lt;variable&gt;</td>
</tr>
<tr>
<td>?Data</td>
<td>&lt;integer&gt; or &lt;variable&gt;</td>
</tr>
<tr>
<td>?Goal</td>
<td>&lt;goal&gt; or &lt;variable&gt;</td>
</tr>
</tbody>
</table>

Comments
This is not a built-in predicate, but an optional hook that can be written by the user to gain control whenever a Windows message occurs. The target window, raw message number, data parameter and interrupted goal are unified with Window, Message, Data and Goal respectively.

Examples
The following program will write out information about all messages to windows other than the console ("(1,1)"). Note that the console must be excluded, because simply writing to it generates further messages; the first clause syphons off console messages quietly:

'\?MESSAGE?'( (1,1), Message, Data, Goal ) :-

\n
message_hook( (1,1), Message, Data, Goal ).

'\?MESSAGE?'( Window, Message, Data, Goal ) :-

\n
output( Current ),
output( 0 ),
writeq( Window-Message-Data-Goal ),
nl,
message_hook( Window, Message, Data, Goal ),
output( Current ).

After compiling this program, any activity in windows other than the console will display messages such as the following:

(untitled,1) - 5 - 1 - eread(_1,_2)
(untitled,1) - 22 - -1071644604 - eread(_1,_2)
(untitled,1) - 4 - 6492 - eread(_1,_2)
Notes

The `?MESSAGE?/4` hook predicate is only significant during program development, while the development environment is active. In stand-alone applications, any arity-four predicate can be nominated for the task of intercepting messages. Because the requirements of a stand-alone application may differ from that of development-time testing, it is usually best to choose a predicate other than `?MESSAGE?/4` in applications.

In the absence of a user-supplied definition of `?MESSAGE?/4`, the WIN-PROLOG development environment will call the built-in predicate, `message_hook/4`. This processes the message, either executing a system function, menu option or the like, or passing the message on to a user-defined window handler (see `window_handler/2` and `window_handler/4`).

Please note that the WIN-PROLOG development environment is entirely message-driven, and relies on the correct handling of messages in order to function. The `?MESSAGE?/4`, if defined, gets messages ahead of the environment, and must call `message_hook/4` to process the message unless it is explicitly intended to disable a particular feature. Please also note that it is the environment that gives messages names rather than numbers, and which also parses data parameters into meaningful terms. Unlike `?MESSAGE?/4`, user-defined window handlers are called by the environment, with meaningful message names and data parameters, and it is recommended that these are used to process messages in all but very special, system-level programs.
**message_hook/4**

*default message hook in development environment*

```prolog
message_hook( Window, Message, Data, Goal )
```

+Window <window_handle>
+Message <integer>
+Data <integer>
+Goal <goal>

**Comments**

This predicate is called by the development environment to handle the given *Message* and *Data* parameter for the given *Window*, which occurred during execution of the given *Goal*. It is normally only called directly from a user-defined message hook (see `?MESSAGE?/4`).

**Examples**

This predicate should not normally be called directly, apart from in the very special case of a user message hook (see `?MESSAGE?/4`). Its sole function is to initiate a system feature such as a menu option, or to pass a named message and parsed data item to a user-defined window handler. The following program will write out information about all messages to windows other than the console ("(1,1)"), before calling `message_hook/4` to process them:

```prolog
'?MESSAGE?'( (1,1), Message, Data, Goal ) :-
  !,
  message_hook( (1,1), Message, Data, Goal ).

'?MESSAGE?'( Window, Message, Data, Goal ) :-
  output( Current ),
  output( 0 ),
  writeq( Window-Message-Data-Goal ),
  nl,
  message_hook( Window, Message, Data, Goal ),
  output( Current ).
```

**Notes**

The `message_hook/4` predicate is normally only used in system-level programming, where it is necessary to handle messages in their raw state, and before they are made visible to the environment. In most applications programming, use should be made of user-defined window handler (see `window_handler/2` and `window_handler/4`) instead.
midhdl/2

convert between logical and midi device handle

midhdl( Name, Raw )

?Name <atom>
?Raw <integer> or <variable>

Comments

This converts between a MIDI device handle and its Raw, integer counterpart. When Name is an unbound variable, Raw must be an integer: an attempt is made to check that this integer identifies a MIDI device, and if not the predicate fails. If successful, a further test is made to see whether the MIDI device was created by WIN-PROLOG: if so, its logical window handle is bound to MIDI device as an atom; otherwise, the predicate fails. If Name is an atom, a check is made to see if it defines a currently-existing MIDI device; if so, the raw handle of that window is unified with Raw. In both cases, midhdl/2 fails if the specified MIDI device does not exist.

Examples

The following call uses mcreate/4 to create an output MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI synth:

?- mcreate( foo, -1, 0, 0 ).
yes

Assuming the connection has succeeded, we can pick up the logical MIDI handle using midhdl/2:

?- midhdl( foo, Midi ).
Midi = -2099258076

Once we are finished, we can close the MIDI device with mclose/1, releasing its resources:

?- mclose( foo ).
yes

Notes

The midhdl/2 predicate performs two primary functions: firstly, it provides a quick test for the existence of any given MIDI device: "midhdl( MIDI device, _ )" will simply succeed or fail depending upon whether or not the MIDI device exists. Secondly, it converts window handles to and from their "raw" integer state. The MIDI device predicates within WIN-PROLOG work with "logical" MIDI device handles; however, external functions called with winapi/4 cannot interpret such handles.
Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes \texttt{mclose/1}, \texttt{mcreate/4}, \texttt{mdatadata/4}, \texttt{mdict/2}, \texttt{midhdl/2}, \texttt{mrecv/3}, \texttt{msend/3}, \texttt{mstat/3} and \texttt{mtime/2} together with \texttt{midi_handler/1} and \texttt{midi_handler/2}; see Appendix Q for a detailed overview.
midi_handler/2

get or set the handler for the given midi device

    midi_handler( Name, Handler )

+Name       <atom>
+Handler     <atom> or <variable>

Comments This gets or sets the name of an arity-3 predicate that will be used as the socket Handler for the MIDI device of the given Name. If Handler is an unbound variable, it returns the name of the existing handler; otherwise, it sets the given Handler name to be associated with the device Name.

Examples By default, all messages to MIDI devices are handled by a predicate, midi_handler/3; you can see this by calling midi_handler/2 with a MIDI device name, even before the MIDI device has been created:

    ?- midi_handler( foo, H ).
    H = midi_handler

Now consider the following MIDI device handler:

    foo( MIDI, Time, Count ) :-
        (  write( MIDI = Time / Count ),
            nl
        ) ~> user,
        midi_handler( MIDI, Time, Count ).

Once this code has been compiled, we can assign foo/3 to become the MIDI device handler for our forthcoming MIDI device, foo:

    ?- midi_handler( foo, foo ).
    yes

The following call uses mcreate/4 to create an input MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer’s default MIDI input:

    ?- mcreate( foo, 0, 0, 0 ).
yes

Assuming the connection has succeeded, and a MIDI source (such as a keyboard) is attached, the user presses a note; the following message will be displayed:

\[ \text{foo} = 1234 / 1 \]

This message shows that an event has been received for MIDI device "foo", arriving 1234 milliseconds after the device was created, and with a "lapse" count of one (1). Normally a MIDI handler would use this as the cue to receive the note, but here we will do it manually:

?- \text{mrecv( foo, Time, Data ).}
Time = 1234 ,
Data = (144,60,64)

This returns a "Note On" message from channel zero of the MIDI device (144), indicating that the user played Middle C (60) with medium velocity (64) 1234 milliseconds after the MIDI device was created. When the user releases the key, a second event will be signalled by handler, and we can retrieve this message as before:

?- \text{foo} = 5678 / 1

?- \text{mrecv( foo, Time, Data ).}
Time = 5678 ,
Data = (144,60,0)

This "Note On" message shows a velocity of zero (0), which indicates that the user released the key 5678 milliseconds after the MIDI device was created. Assuming no further MIDI events have been generated by the user, a third call will fail:

?- \text{mrecv( foo, Time, Data ).}
\text{no}

Once we are finished, we can close the MIDI device with \text{mclose/1}, releasing its resources:

?- \text{mclose( foo ).}
yes

A final piece of housekeeping is to remove the association of the MIDI device name, "foo", with the handler, "foo/0". We do this with
\textit{midi_handler/2}, passing in either the name, "midi_handler" or the empty list, "[]", which is a shorthand for the same:

\begin{verbatim}
?- midi_handler( foo, [] ).
yes
\end{verbatim}

\textbf{Notes}

A MIDI device can be created in one of two modes: input or output. Input devices can be handled simply by polling, using \texttt{mrecv/3} to attempt to retrieve data repetitively, with or without \texttt{mstat/3} being used to check the input status, but this tends to result in programs that hog the CPU. A far better approach to processing incoming MIDI is to write a handler, assign it to a MIDI channel with \texttt{midi_handler/2}, and call \texttt{mrecv/3} to receive messages only when signalled.

The "lapse count" (Count) parameter serves as a performance indicator. Should two or more MIDI events be signalled before the MIDI handler is successfully invoked, only one event will actually be signalled: the lapse count will indicate the number of events that are being signalled simultaneously. After a successful call to \texttt{mrecv/3}, if further data remains in the input buffers, a new event is signalled immediately, this time with a lapse count of zero (0).

In general, you can safely ignore the lapse count, other than to keep a general watch on its values. Most of the time, it should report as "0" or "1", sometimes climbing to "2" or "3" in period of very busy MIDI activity. If the count gets much larger than this, it indicates that your application cannot process incoming MIDI data quickly enough.

MIDI devices that have been set to output generate events in response to calls to \texttt{msend/3} and \texttt{mtime/2}, either immediately, if the calls succeed, or after an appropriate delay, if they fail. The lapse count is always returned as "-1" for output MIDI devices, and can be used to distinguish between input and output devices in a shared handler.

Starting in version 4.700, \textsl{WIN-PROLOG} has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (Sysex librarians, sequencers, arpeggiators, transposers, etc), and for \textsc{ai} music research (music recognition, analysis, etc).

The full set of MIDI predicates includes \texttt{mclose/1}, \texttt{mcreate/4}, \texttt{mdata/4}, \texttt{mdict/2}, \texttt{midhdl/2}, \texttt{mrecv/3}, \texttt{msend/3}, \texttt{mstat/3} and \texttt{mtime/2} together with \texttt{midi_handler/1} and \texttt{midi_handler/2}; see Appendix Q for a detailed overview.
**midi_handler/3**

*default MIDI device handler*

\[
\text{midi_handler}( \text{Name}, \text{Time}, \text{Count} )
\]

+Name <atom>
+Time <atom>
+Count <integer>

**Comments**

This provides standard, default processing of MIDI device events, specified by the MIDI device of the given Name, the Time at which the handler was invoked, and the lapse Count.

**Examples**

By default, all messages to MIDI devices are handled by a predicate, \text{midi_handler/3}; you can see this by calling \text{midi_handler/2} with a MIDI device name, even before the MIDI device has been created:

\[
?\text{- midi_handler}( \text{foo}, \text{H} ).
\]

\[
\text{H} = \text{midi_handler}
\]

Now consider the following MIDI device handler:

\[
\text{foo( MIDI, Time, Count ) :-}
\]
\[
( \text{ write( MIDI = Time } / \text{ Count ),}
\]
\[
\text{ nl}
\]
\[
\text{ ) } \text{-> user,}
\]
\[
\text{ midi_handler( MIDI, Time, Count ).}
\]

Once this code has been compiled, we can assign \text{foo/3} to become the MIDI device handler for our forthcoming MIDI device, \text{foo}:

\[
?\text{- midi_handler( \text{foo, foo} ).}
\]

\[
\text{yes}
\]

The following call uses \text{mcreate/4} to create an input MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer’s default MIDI input:

\[
?\text{- mcreate( foo, 0, 0, 0 ).}
\]
Assuming the connection has succeeded, and a MIDI source (such as a keyboard) is attached, the user presses a note; the following message will be displayed:

    foo = 1234 / 1

This message shows that an event has been received for MIDI device "foo", arriving 1234 milliseconds after the device was created, and with a "lapse" count of one (1). Normally a MIDI handler would use this as the cue to receive the note, but here we will do it manually:

    ?- mrecv( foo, Time, Data ).
    Time = 1234 ,
    Data = (144,60,64)

This returns a "Note On" message from channel zero of the MIDI device (144), indicating that the user played Middle C (60) with medium velocity (64) 1234 milliseconds after the MIDI device was created. When the user releases the key, a second event will be signalled by handler, and we can retrieve this message as before:

    ?- foo = 5678 / 1
    ?- mrecv( foo, Time, Data ).
    Time = 5678 ,
    Data = (144,60,0)

This "Note On" message shows a velocity of zero (0), which indicates that the user released the key 5678 milliseconds after the MIDI device was created. Assuming no further MIDI events have been generated by the user, a third call will fail:

    ?- mrecv( foo, Time, Data ).
    no

Once we are finished, we can close the MIDI device with mclose/1, releasing its resources:

    ?- mclose( foo ).
    yes

A final piece of housekeeping is to remove the association of the MIDI device name, "foo", with the handler, "foo/0". We do this with
mid_handler/2, passing in either the name, "mid_handler" or the empty list, "[]", which is a shorthand for the same:

?- midi_handler( foo, [] ).  
yes

Notes

A MIDI device can be created in one of two modes: input or output. Input devices can be handled simply by polling, using mrecv/3 to attempt to retrieve data repetitively, with or without mstat/3 being used to check the input status, but this tends to result in programs that hog the CPU. A far better approach to processing incoming MIDI is to write a handler, assign it to a MIDI channel with mid_handler/2, and call mrecv/3 to receive messages only when signalled.

The "lapse count" (Count) parameter serves as a performance indicator. Should two or more MIDI events be signalled before the MIDI handler is successfully invoked, only one event will actually be signalled: the lapse count will indicate the number of events that are being signalled simultaneously. After a successful call to mrecv/3, if further data remains in the input buffers, a new event is signalled immediately, this time with a lapse count of zero (0).

In general, you can safely ignore the lapse count, other than to keep a general watch on its values. Most of the time, it should report as "0" or "1", sometimes climbing to "2" or "3" in period of very busy MIDI activity. If the count gets much larger than this, it indicates that your application cannot process incoming MIDI data quickly enough.

MIDI devices that have been set to output generate events in response to calls to msend/3 and mtime/2, either immediately, if the calls succeed, or after an appropriate delay, if they fail. The lapse count is always returned as "-1" for output MIDI devices, and can be used to distinguish between input and output devices in a shared handler.

Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes mclose/1, mcreate/4, mdata/4, mdict/2, midhdl/2, mrecv/3, msend/3, mstat/3 and mtime/2 together with mid_handler/1 and mid_handler/2; see Appendix Q for a detailed overview.
mkdir/1

create a file directory

    mkdir( Directory )

    +Directory <atom>

Comments This creates the named, empty Directory on disk. If a directory or file with the given name already exists, an error is generated.

Examples The following command creates a directory call "temp" within the current working directory:

    ?- mkdir( temp ).

    yes

Notes This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.

Creating a directory with mkdir/1 does not change the current working directory. To do this, a subsequent call should be made to chdir/1. Empty directories can be deleted with rmdir/1.
mnuhdl/2

convert between a menu and its handle

\texttt{mnuhdl( Menu, Handle )}

\texttt{?Menu} \hspace{1cm} \texttt{<object\_handle> or <variable>}
\texttt{?Handle} \hspace{1cm} \texttt{<integer> or <variable>}

**Comments**

This converts between a menu and its raw (integer) handle. The \textit{Menu} parameter may be a named menu, a handle of the form, "menu(Raw)", where "Raw" is the raw handle.

**Examples**

The following call will return the raw handle for a menu called "foo", assuming that such a menu has previously been created:

\texttt{?- mnuhdl( foo, H ).}
\texttt{H = 65535}

**Notes**

This is one of a family of \textbf{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
mrecv/3

receive data from a MIDI device

\[ \text{mrecv}( \text{Name}, \text{Time}, \text{Data} ) \]

+Name atom
-Time variable
-Data variable

Comments

This attempt to receive data from a connection associated with the MIDI device of the given Name, returning the Time at which the data was received, and returning the latter either as a conjunction of three integers, or as a string bound to Data. If no data is available on the MIDI device, mrecv/3 fails; otherwise it will return one MIDI short or long message.

Examples

The following call uses mcreate/4 to create an input MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI input:

\[ \text{?- mcreate}( \text{foo}, 0, 0, 0 ). \]

\text{yes} \quad \text{<enter>}

Assuming the connection has succeeded, and a MIDI source (such as a keyboard) is attached, we can attempt to receive some data:

\[ \text{?- mrecv}( \text{foo}, \text{Time}, \text{Data} ). \]

\begin{verbatim}
Time = 1234 ,
Data = (144,60,64)
\end{verbatim} \quad \text{<enter>}

This returns a "Note On" message from channel zero of the MIDI device (144), indicating that the user played Middle C (60) with medium velocity (64) 1234 milliseconds after the MIDI device was created. Assuming the user has released the key, a second call will return further data:

\[ \text{?- mrecv}( \text{foo}, \text{Time}, \text{Data} ). \]

\begin{verbatim}
Time = 5678 ,
Data = (144,60,0)
\end{verbatim} \quad \text{<enter>}

This "Note On" message shows a velocity of zero (0), which indicates that the user released the key 5678 milliseconds after the MIDI device was created. Assuming no further MIDI events have been generated by the user, a third call will fail:
Once we are finished, we can close the MIDI device with mclose/1, releasing its resources:

?- mclose( foo ).
yes

Notes

Immediately a MIDI device is created (see mcreate/4), an associated timer is set running, counting in milliseconds. For input devices, each time an event occurs, it is stored together with the current value in the timer, enabling precise timings to be recorded, irrespective of when mrecv/3 is called to retrieve the event. For output devices, the timer is used to determine whether or not a given MIDI message can be sent with msend/3, and to schedule an event if not. The value in the timer can be retrieved or tested with mtime/2, which can also be used to schedule events.

Two types of MIDI message exist: short and long. Short messages comprise 3 bytes, and are represented in WIN-PROLOG as a conjunction of 3 small integers, and define things like "Note On" events, controller changes and patch changes. Long messages are represented in WIN-PROLOG as a Strings, and contain SysEx (System Exclusive) data, either in whole or in part.

Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySx librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes mclose/1, mcreate/4, mdata/4, mdict/2, midhdl/2, mrecv/3, msend/3, mstat/3 and mtime/2 together with midi_handler/1 and midi_handler/2; see Appendix Q for a detailed overview.
**ms/2**

*call a goal and return its execution duration*

\[ \text{ms( Goal, Duration )} \]

+Goal <goal>
?Duration <number> or <variable>

**Comments**
This calls the given Goal, timing its execution in milliseconds (mS), and unifying the result with Duration. It succeeds or fails depending upon whether the Goal itself succeeds or fails and, if Duration is given, whether the execution time matches it.

**Examples**
The following command invokes the member/3 predicate, returning the total elapsed execution time of three separate solutions. Note that the second and third times are dependent on how quickly the user presses <space> between solutions:

```
?- ms( member(X,[1,2,3]), D ).
X = 1 ,
D = 7.542868636752213e-3 ;

X = 2 ,
D = 512.3038435106194 ;

X = 3 ,
D = 1024.645837174618 ;

no
```

The following program can be used to implement a well-behaved sleep function (see wait/1, which provides a mechanism to "yield" to other Windows processes):

```
sleep( Milliseconds ) :-
    ms( (repeat,wait(0)), Duration ),
    Duration >= Milliseconds,
    !.
```

**Notes**
The ms/2 predicate typically returns time to a resolution of around 1µs (one microsecond) or better, depending upon the computer plat-
form in use; the actual resolution, in terms of ticks per second, can be determined by calling the system time predicate with a pair of variables (see \texttt{time/2}). Note that prior to \textbf{WIN-PROLOG 4.7}, the resolution of \texttt{ms/2} was 1ms (one millisecond), with all times returned as integers.
msend/3

send data to a MIDI device

```prolog
msend( Name, Time, Data )
```

- **Name** <atom>
- **Time** <variable>
- **Data** <string> or (<integer>,<integer>,<integer>)

**Comments**
This attempts to send the given Data to a MIDI device of the given Name, provided it has already attained the given Time. If the specified Time has not yet elapsed since the MIDI device was created, mrecv/3 fails, but a MIDI event is scheduled on the device to signal the earliest time when the predicate can be called successfully.

**Examples**
The following call uses mcreate/4 to create an output MIDI device with the default sysex buffer size on device index “0”, which usually refers to a computer’s default MIDI synth:

```prolog
?- mcreate( foo, -1, 0, 0 ).
yes
```

Assuming the connection has succeeded, we can send a simple MIDI "Note On" message using msend/3:

```prolog
?- msend( foo, 0, (144,60,64) ).
yes
```

By specifying a time of zero (0), we can guarantee this message will be sent immediately, specifying in this case a "Note On" message to channel zero of the MIDI device (144), playing Middle C (60) with medium velocity (64). The following call will attempt to cancel the note, by sending another "Note On" message to the same channel, specifying the same note (60) but with a velocity of zero, once the timer has reached 5678 milliseconds:

```prolog
?- msend( foo, 5678, (144,60,0) ).
no
```

In this example, the call has failed, because the timer has not yet reached 5678 milliseconds. However, an event will have been signalled, and once this has been detected (see midi_handler/3), the same call will work:
Once we are finished, we can close the MIDI device with `mclose/1`, releasing its resources:

?- mclose( foo ).
   yes

Notes

Immediately a MIDI device is created (see `mcreate/4`), an associated timer is set running, counting in milliseconds. For input devices, each time an event occurs, it is stored together with the current value in the timer, enabling precise timings to be recorded, irrespective of when `mrecv/3` is called to retrieve the event. For output devices, the timer is used to determine whether or not a given MIDI message can be sent with `msend/3`, and to schedule an event if not. The value in the timer can be retrieved or tested with `mtime/2`, which can also be used to schedule events.

Two types of MIDI message exist: short and long. Short messages comprise 3 bytes, and are represented in `WIN-PROLOG` as a conjunction of 3 small integers, and define things like "Note On" events, controller changes and patch changes. Long messages are represented in `WIN-PROLOG` as a Strings, and contain SysEx (System Exclusive) data, either in whole or in part.

Starting in version 4.700, `WIN-PROLOG` has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (Sy5ex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes `mclose/1`, `mcreate/4`, `mdata/4`, `mdict/2`, `midhdl/2`, `mrecv/3`, `msend/3`, `mstat/3` and `mtime/2` together with `midi_handler/1` and `midi_handler/2`; see Appendix Q for a detailed overview.
msgbox/4

create a message box and return a response

\[
\text{msgbox} ( \text{Title}, \text{Message}, \text{Style}, \text{Response} )
\]

+Title \text{<term>}
+Message \text{<term>}
+Style \text{<integer>}
-Response \text{<integer> or <variable>}

Comments
This displays a standard Windows message box in the given Style, with the given Title and Message, and then waits for a button to be clicked, before unifying its numerical value with Response. The Style comprises a sum of four components, shown here in hexadecimal notation, the first of which specifies which buttons to use:

<table>
<thead>
<tr>
<th>Value</th>
<th>Buttons</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000000</td>
<td>OK</td>
</tr>
<tr>
<td>16'00000001</td>
<td>OK + Cancel</td>
</tr>
<tr>
<td>16'00000002</td>
<td>Abort + Retry + Ignore</td>
</tr>
<tr>
<td>16'00000003</td>
<td>Yes + No + Cancel</td>
</tr>
<tr>
<td>16'00000004</td>
<td>Yes + No</td>
</tr>
<tr>
<td>16'00000005</td>
<td>Retry + Cancel</td>
</tr>
</tbody>
</table>

The second element of the style defines which icon, if any, to display alongside the message:

<table>
<thead>
<tr>
<th>Value</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000000</td>
<td>&lt;None&gt;</td>
</tr>
<tr>
<td>16'00000010</td>
<td>Stop Sign</td>
</tr>
<tr>
<td>16'00000020</td>
<td>Question Mark</td>
</tr>
<tr>
<td>16'00000030</td>
<td>Exclamation Mark</td>
</tr>
<tr>
<td>16'00000040</td>
<td>Information Mark</td>
</tr>
</tbody>
</table>

The third component of the style determines which of the buttons should gain initial focus as the default button:
Finally, the last style feature determines the overall appearance and modality of the message box:

<table>
<thead>
<tr>
<th>Value</th>
<th>Modality</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000000</td>
<td>Application</td>
</tr>
<tr>
<td>16'00001000</td>
<td>System</td>
</tr>
<tr>
<td>16'00002000</td>
<td>Task</td>
</tr>
<tr>
<td>16'00003000</td>
<td>None</td>
</tr>
</tbody>
</table>

The button clicked by the user is identified by one of the following numbers, which is unified with Button:

<table>
<thead>
<tr>
<th>Value</th>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OK</td>
</tr>
<tr>
<td>2</td>
<td>Cancel</td>
</tr>
<tr>
<td>3</td>
<td>Abort</td>
</tr>
<tr>
<td>4</td>
<td>Retry</td>
</tr>
<tr>
<td>5</td>
<td>Ignore</td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

**Examples**

The following call creates a message box whose style includes a single "OK" button and exclamation mark icon, together with the given title and message; the result is only returned once the user clicks on the button or hits <enter>:

```
?- msgbox(["Tasty"], value("Pi",3.14159), 48, D).
D = 1
```

**Notes**

The \texttt{msgbox/4} predicate provides direct access to the standard Windows message box dialog, with greater flexibility than its sibling, \texttt{message_box/3}. The latter predicate always displays a fixed title, "WIN-PROLOG", and always shows the "exclamation" icon; the former allows
the user to specify the title, icon and style, and supports additional button combinations.

Both predicates allow any arbitrary term to be given as the "message": although strings and atoms will be the most common data types used for messages, support for other terms (such as in the example above) simplifies applications such as error handling, or one-off debugging messages. Terms are displayed unquoted, as if output by the write/1 predicate.

It is important to take care when experimenting with message box styles: several values can result in a message box with no buttons whatsoever. Since the message box is a modal dialog, such an occurrence locks WIN-PROLOG, and provides no way of regaining control. Under these circumstances it is usually necessary to terminate the Prolog session from the task manager, which will result in the loss of any changes made since a program was last saved.
mstat/3

return or test status of a MiDi device

mstat( Name, Event, Sysex )

+Name       <atom>
?Event      <integer> or <variable>
?Sysex      <integer> or <variable>

Comments   This gets or tests the Event and Sysex counts for the MiDi device of the given Name. For an output device, the Event and Sysex counts are always -1 and 0 respectively; for an input device, the counts represent the number of buffered events and sysex messages that can be read without failure (see mrecv/3). To see the maximum amount of data that can be buffered, use mdata/4.

Examples   The following call uses mcreate/4 to create an input MiDi device with the default sysex buffer size on device index "0", which usually refers to a computer’s default MiDi input:

?- mcreate( foo, 0, 0, 0 ).  <enter>
yes

Now we can check for the presence of any pending MiDi events on this input channel:

?- mstat( foo, Event, Sysex ).  <enter>
Event = Sysex = 0

This result confirms that no data yet exists, and any call to mrecv/3 will fail. Assuming the connection has succeeded, a MiDi source (such as a keyboard) is attached, and that the user has now pressed and released a key, a second call to mstat/3 will indicate the presence of two events:

?- mstat( foo, Event, Sysex ).  <enter>
Event = 2 ,
    Sysex = 0

In the knowledge that two events are now waiting, we can receive the data:

?- mrecv( foo, Time, Data ).  <enter>
Time = 1234 ,
Data = (144,60,64)

This returns a "Note On" message from channel zero of the MIDI device (144), indicating that the user played Middle C (60) with medium velocity (64) 1234 milliseconds after the MIDI device was created. Assuming the user has released the key, a second call will return further data:

?- mrecv( foo, Time, Data ).
<enter>
Time = 5678 ,
Data = (144,60,0)

This "Note On" message shows a velocity of zero (0), which indicates that the user released the key 5678 milliseconds after the MIDI device was created. Assuming no further MIDI events have been generated by the user, a third call will fail:

?- mrecv( foo, Time, Data ).
<enter>
no

Once we are finished, we can close the MIDI device with mclose/1, releasing its resources:

?- mclose( foo ).
<enter>
yes

Notes
Two types of MIDI message exist: short and long. Short messages comprise 3 bytes, and are represented in WIN-PROLOG as a conjunction of 3 small integers, and define things like "Note On" events, controller changes and patch changes. Long messages are represented in WIN-PROLOG as a Strings, and contain SysEx (System Exclusive) data, either in whole or in part.

Every MIDI input event, whether it represents a short or a long message, is stored in a buffer, whose size is determined in the call to mcreate/4: it is the number buffered events that is returned in the Event parameter in mstat/3. Short messages are wholly contained within this buffer, while long messages also consume space in the SysEx buffer, whose size is also determined in the call to mcreate/4. The latter buffer is divided into a series of one-kilobyte (1024 byte) segments, and it is the number of segments currently containing buffered sysex data that is returned in the SysEx parameter of mstat/3. The total amount of buffer space can be determined by calling mdata/4, which, together with mstat/3, simplifies the management of MIDI memory requirements.

Starting in version 4.700, WIN-PROLOG has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages.
when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes \texttt{mclose/1}, \texttt{mcreate/4}, \texttt{mdata/4}, \texttt{mdict/2}, \texttt{mihdl/2}, \texttt{mrecv/3}, \texttt{msend/3}, \texttt{mstat/3} and \texttt{mtime/2} together with \texttt{midi_handler/1} and \texttt{midi_handler/2}; see Appendix Q for a detailed overview.
mtime/2

return or test elapsed time of a MIDI device

```prolog
mtime( Name, Time )
```

+Name <atom>
?Time <integer> or <variable>

Comments

This gets or tests the elapsed Time count for the MIDI device of the given Name. If Time is a variable, it returns the current time, in milliseconds, since the specified device was created (see mcreate/4); if the time is given as an integer, mtime/2 succeeds if the specified value is less than or equal to the elapsed time, and fails if greater. In the latter case, a timer is scheduled to signal an event whenever the lesser of the given timestamp and that of any pending event expires.

Examples

The following call uses mcreate/4 to create an input MIDI device with the default sysex buffer size on device index "0", which usually refers to a computer's default MIDI input:

```prolog
?- mcreate( foo, 0, 0, 0 ).
yes
```

Now we can check for the presence of any pending MIDI events on this input channel:

```prolog
?- mtime( foo, Time ).
Time = 1234
```

This shows that the given MIDI device has been running for 1234 milliseconds; the following call, made at some later point, will succeed or fail depending upon whether the elapsed time has reached 5678 milliseconds (in this case, yes):

```prolog
?- mtime( foo, 5678 ).
yes
```

Once we are finished, we can close the MIDI device with mclose/1, releasing its resources:

```prolog
?- mclose( foo ).
yes
```
Immediately a MIDI device is created (see \texttt{mcreate/4}), an associated timer is set running, counting in milliseconds. For input devices, each time an event occurs, it is stored together with the current value in the timer, enabling precise timings to be recorded, irrespective of when \texttt{mrecv/3} is called to retrieve the event. For output devices, the timer is used to determine whether or not a given MIDI message can be sent with \texttt{msend/3}, and to schedule an event if not. The value in the timer can be retrieved or tested with \texttt{mtime/2}, which can also be used to schedule events.

Starting in version 4.700, \textbf{WIN-PROLOG} has built-in support for MIDI (Musical Instrument Digital Interface), providing direct and easy access to both input and output MIDI devices, such as sound cards, external keyboards, synthesizers and samplers. The MIDI device predicates have been designed to work in an "Asynchronous" mode, together with dedicated high-resolution timers and special MIDI event messages when ready to process data. They are designed for use in both practical MIDI applications (SySex librarians, sequencers, arpeggiators, transposers, etc), and for AI music research (music recognition, analysis, etc).

The full set of MIDI predicates includes \texttt{mclose/1}, \texttt{mcreate/4}, \texttt{mdata/4}, \texttt{mdict/2}, \texttt{mhdil/2}, \texttt{mrecv/3}, \texttt{msend/3}, \texttt{mstat/3} and \texttt{mtime/2} together with \texttt{midi_handler/1} and \texttt{midi_handler/2}; see Appendix Q for a detailed overview.
multifile/1

*declare predicates as being defined in more than one file*

```
multifile( Preds )
+Preds <pred_specs>
```

**Comments**
This records that any predicates specified in *Preds* may be defined in clauses from more than one file.

**Examples**
The following call states that a predicate called *foo/1* may consist of clauses from more than one file:

```
?- multifile( foo/1 ).
yes
```

A number of predicates can be declared in a single call, using either a list or a conjunction:

```
yes

?- multifile( (bar/2,sux/3,you/4) ).
yes
```

Because "multifile" is defined as a prefix operator with a very high precedence, it is also possible to declare predicates using the following syntax:

```
?- multifile foo/1 .
yes

?- multifile bar/2, sux/3, you/4.
yes
```

**Notes**
Normally an existing predicate is replaced when a file is loaded which contains a new definition of that predicate: if this predicate has been declared "multifile", however, the new clauses from the file are added to the existing definition. Note that when optimising files, the *incremental compiler* is used to compile any predicates declared multifile: the *optimising compiler* creates single, stand-alone procedures from each predicate, and its use implicitly prevents the incremental extending of code when new files are loaded.
**name/2**

*convert between an atomic term and a char list*

```prolog
name( Atomic, List )
```

- ?Atomic: <number>, <atom> or <string>
- ?List: <char_list> or <variable>

**Comments**

This converts between atomic data types and their unquoted equivalents, represented as char lists. If `Atomic` is bound to a number, atom or string, then a list of integers comprised of its character codes is unified with `List`; if `Atomic` is an unbound variable and `List` is bound to a list of integers, the latter is assembled into the corresponding `Atomic` term.

**Examples**

The following command returns a char list comprising the characters that represent the number "123.45":

```prolog
?- name( 123.45, L ).
```

```
L = [49,50,51,46,52,53]
```

When converting from a list to an atomic term, a number is returned if possible:

```prolog
?- name( '123.45', L ), name( T, L ).
```

```
L = [49,50,51,46,52,53],
T = 123.45
```

The following call confirms that the string "123.45" contains the characters, "123.45":

```prolog
?- name( `123.45`, "123.45" ).
```

```
yes
```

**Notes**

As can be seen for the examples above, `name/2` is not orthogonal. When converting from an atomic data type to a list and then converting back, a different data type might result. In particular, `name/2` only returns atoms or numbers, and never strings. Numbers are returned whenever the char list contains only the character code representation of a number.

The `name/2` predicate is a legacy from the earliest Edinburgh Prolog systems, and other, newer predicates, including `atom_chars/2`, `number_chars/2` and `string_chars/2`, should be used to provide safer, more predictable data conversions.
**nl/0**

*output a new line to the current output stream*

```
nl
```

**Comments**
This predicate simply outputs a carriage return and line feed to the current output stream, to complete a line of output.

**Examples**
The following call writes out two lines, each finishing with a new line command:

```
?- write( hello ), nl, write( world ), nl.
hello
world
yes
```

**Notes**
The *nl/0* predicate provides a portable way to output the end of line sequence: although in Windows, it is directly equivalent to outputting `<cr><lf>`, other operating systems use different character combinations to indicate a new line.
no_style_check/1

disable the specified compile-time style checking

    no_style_check( Type )

    +Type <atom>

Comments This disables the specified Type of compile-time style checking, as shown in the following table:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>single_var</td>
<td>named variable occurs only once in a given clause</td>
</tr>
<tr>
<td>discontinuous</td>
<td>clauses for a given relation are not contiguous</td>
</tr>
<tr>
<td>multiple</td>
<td>predicate is defined in more than one file</td>
</tr>
<tr>
<td>all</td>
<td>all of the above</td>
</tr>
</tbody>
</table>

Examples The following call turns off all style checking:

    ?- no_style_check( all ).
    yes

Notes In general, it is recommended to use style checking when compiling programs: the most common Prolog programming errors are the result of mis-spelt variable names, and when set, the single_var style check can usually pick this up at compile time.
Nodebug/0

set the debugging (debug) mode to "off"

nodebug

Comments
This predicate sets the debugging mode to "off", which will prevent the debugger being invoked even if a new command is entered at the console or a predicate is called for which a spypoint is currently set.

Examples
The following call sets a spypoint on a predicate "foo/1", but then sets the debugging mode to "off" so that the debugger will not take over, even when foo/1 is called. At some later stage, the debugging mode could be set to "debug" to enable this and any other pending spypoints:

?- spy( foo/1 ).  
# Spypoint placed on predicate foo / 1
yes

?- nodebug.     
yes

Notes
The debugging mode has three settings: "off", "debug", and "trace". In the first case, the debugger is never invoked, while in the second, it is invoked if a spied predicate is called. In the third case, the debugger is invoked immediately a query is entered at the console prompt.

The three debugging modes are mutually exclusive, but are somewhat confusingly set by four predicates, as shown in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>nodebug/0 or notrace/0</td>
</tr>
<tr>
<td>debug</td>
<td>debug/0</td>
</tr>
<tr>
<td>trace</td>
<td>trace/0</td>
</tr>
</tbody>
</table>

The nodebug/0 and notrace/0 predicates are effectively identical, and either can be used to cancel the effects of either debug/0 or trace/0. Please note that WIN-PROLOG's default debugging mode is "debug".
**nofileerrors/0**

_disable the reporting of file error messages_

**nofileerrors**

**Comments** This sets the fileerrors flag to "off", so that no error is generated by predicates such as see/1, tell/1 and open/2 if a specified file cannot be opened.

**Examples** The following pair of calls clear the fileerrors flag, and then attempts to "see" a nonexistent file, resulting in failure:

```prolog
?- nofileerrors. <enter>
yes

?- see( foo ). <enter>
no
```

**Notes** The fileerrors flag is set by default, so that, unlike the above example, errors normally occur when an attempt is made to open a nonexistent or prohibited file. The nofileerrors/0 predicate has an opposite, fileerrors/0, which is used to set the fileerrors flag. This, and many other flags, can also be managed via the prolog_flag/2 and prolog_flag/3 predicates.
nogc/0

*disable certain explicit garbage collection invocations*

```
nogc
```

**Comments**

This sets the value of the "gc" Prolog flag to "off", so that the garbage_collect/0 predicate will be disabled (also see gc/0, which resets the "gc" Prolog flag to "off").

**Examples**

The following sets the "gc" Prolog flag to "off":

```
?- nogc.
yes
```

**Notes**

This predicate is one of several included solely for Quintus Prolog compatibility. In WIN-PROLOG, the garbage collector is always enabled, and runs entirely automatically whenever required. There is never any need to set garbage collection "modes", nor can the garbage collector ever be disabled.

The only time it is ever desirable to run the garbage collector explicitly is before calling free/9, in order to work out how much space is available in the term or text heaps, or possibly before running a benchmark routine or profiling a program in order to provide a repeatable test. In either event, it is best to invoke the garbage collector with the gc/1 predicate, whose argument defines the type of garbage collection to perform.
**nohash/1**

_revert an existing hashed predicate to incremental status_

```prolog
nohash( Pred )
```

+Pred <pred_spec>

**Comments**

This removes the effect of the _hashing compiler_ to revert an existing predicate, that has previously been hashed, to the state it was in when original compiled by the _incremental compiler_. When files are "consulted" (loaded with `consult/1`), their predicates are incrementally compiled, and may also be automatically hashed to improve their run-time performance.

**Examples**

The following command loads one of the example programs, "salesman.pl", which includes a data predicate, `dist/3`:

```prolog
?- consult( examples(salesman) ).
```

# 0.000 seconds to consult salesman.pl [c:\tempwin\examples]

% 0.000 seconds to hash predicate dist / 3
% Quota = 100, Count = 300, Cases = 24, Total = 6, Worst = 3

yes

The `dist/3` predicate can be de-hashed as follows:

```prolog
?- nohash( dist/3 ).
```

% 0.000 seconds to remove hashing from predicate dist / 3

yes

**Notes**

The _nohash/1_ predicate can only revert programs which are currently hashed, usually in order to re-hash the same predicate using different "quota" and "limit" settings (see `hash/3`).

Code compiled with the hashed compiler can still be debugged and listed, and can return matches many times faster then code compiled with either the incremental or optimising compilers, although with some restrictions. Depending upon the setting of the "hash_files" Prolog flag, hashing may be applied automatically after loading a source file, as described in Appendix N.
nohash/3

revert a hashed predicate to incremental, returning settings

nohash( Pred, Quota, Count )

+Pred <pred_spec>
-Quota <variable>
-Count <variable>

Comments
This removes the effect of the hashing compiler to revert an existing predicate, that has previously been hashed, to the state it was in when original compiled by the incremental compiler, and returns the Quota and clause Count values, which can be used in subsequent calls to hash/3.

Examples
The following command loads one of the example programs, "salesman.pl", which includes a data predicate, dist/3:

?- consult( examples(salesman) ).
<enter>

# 0.000 seconds to consult salesman.pl [c:\tempwin\examples]
% 0.000 seconds to hash predicate dist / 3
% Quota = 100, Count = 300, Cases = 24, Total = 6, Worst = 3
yes

The dist/3 predicate can be de-hashed as follows:

?- nohash( dist/3, Quota, Count ).
<enter>

% 0.000 seconds to remove hashing from predicate dist / 3
yes

Notes
The nohash/1 predicate can only revert programs which are currently hashed, usually in order to re-hash the same predicate using different "quota" and "limit" settings (see hash/3).

Code compiled with the hashing compiler can still be debugged and listed, and can return matches many times faster then code compiled with either the incremental or optimising compilers, although with some restrictions. See hash/1 for further information.
nonvar/1

test whether the given term is not an unbound variable

    nonvar( Term )

?Term <term>

Comments  This predicate succeeds if the given Term is any Prolog term other than an unbound variable; otherwise it fails.

Examples  The following calls test various cases:

    ?- nonvar( 123 ).                    <enter>
    yes

    ?- nonvar( [1,2,3] ).               <enter>
    yes

    ?- nonvar( foo ).                   <enter>
    yes

    ?- nonvar( `bar` ).                 <enter>
    yes

    ?- nonvar( X ).                     <enter>
    no

Notes  This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments. In this case, the predicate has an exact opposite: nonvar/1 is effectively defined in terms of var/1, as follows:

    nonvar( Term ) :-
    \+ var( Term ).
nospy/1

remove spypoints from one or more predicates

\[ \text{nospy( Preds )} \]
\[ +\text{Preds} \quad \text{<pred_specs>} \]

Comments
This removes spypoints from the predicates specified in \textit{Preds}.

Examples
Assuming a predicate exists, called \textit{foo/1}, and that it has a spypoint already set, the following call removes the spypoint:

\[ ?- \text{nospy( foo/1 ).} \]
\[ \text{# Spypoint removed from predicate foo / 1} \]
\[ \text{yes} \]

A number of predicates can be "unspied" in a single call, using either a list or a conjunction:

\[ ?- \text{nospy( [bar/2,sux/3,you/4] ).} \]
\[ \text{# Spypoint removed from predicate bar / 2} \]
\[ \text{# Spypoint removed from predicate susx / 3} \]
\[ \text{# Spypoint removed from predicate you / 4} \]
\[ \text{yes} \]

\[ ?- \text{nospy( (bar/2,sux/3,you/4) ).} \]
\[ \text{yes} \]

Because "nospy" is defined as a prefix operator, it is also possible to declare predicates using the following syntax:

\[ ?- \text{nospy foo/1.} \]
\[ \text{yes} \]

Notes
When the \textit{debugging mode} is set to "debug" (see \textit{debug/0}), and a predicate is called for which a spypoint is currently set (a "spied" predicate), the currently selected system debugger is automatically invoked to trace the predicate's execution. The \textit{spy/1} predicate is used to add a spypoint to one or more predicates, and \textit{nospy/1} is used to remove them once these predicates have been successfully debugged; meanwhile, \textit{nospyall/0} can remove spypoints from all predicates at once.
nospyall/0

remove spypoints from all predicates

nospyall

Comments
This removes spypoints from all spied predicates.

Examples
Assuming a predicate exists, called foo/1, and that it is the only predicate for which a spypoint is already set, the following call removes the spypoint:

?- nospyall.
# Spypoint removed from predicate foo / 1
yes

Notes
When the debugging mode is set to "debug" (see debug/0), and a predicate is called for which a spypoint is currently set (a "spied" predicate), the currently selected system debugger is automatically invoked to trace the predicate's execution. The spy/1 predicate is used to add a spypoint to one or more predicates, and nospy/1 is used to remove them once these predicates have been successfully debugged; meanwhile, nospyall/0 can remove spypoints from all predicates at once.

Note that there is no "spyall/0" predicate: setting spypoints on every predicate would defeat the purpose of "debug" mode, which is to allow a program to run unhindered until the specific item being debugged is first invoked. If it is intended to trace a query from the very top, the debugging mode should be set to "trace" (see trace/0).
not/1

"logical negation": check that a goal is not true

\texttt{not Call}

+Call \texttt{<goal>}

Comments

This predicate succeeds if the given \texttt{Call} fails, fails if the given \texttt{Call} succeeds without binding any variables, or raises an error if the \texttt{Call} does attempt to bind one or more variables, a process known as "logical negation". It is used to check that the proposition made by a call can be shown not to be true.

Examples

The following call succeeds, because the given goal fails:

\begin{verbatim}
?- \texttt{not integer( foo ).}
\end{verbatim}

yes

Conversely, the following call will fail, because the given goal succeeds, without binding any variables:

\begin{verbatim}
?- \texttt{not integer( 123 ).}
\end{verbatim}

no

The \texttt{not/1} predicate never binds variables: if the given goal fails, then nothing is bound by definition; if the goal succeeds and binds some variables, then an error is raised:

\begin{verbatim}
?- \texttt{not (X = foo, fail).}
X = _
\end{verbatim}

\begin{verbatim}
?- \texttt{not (X = foo, true).}
! ----------------------------------------
! Error 21 : Control Error
! Goal : not (_1 = foo,true)
\end{verbatim}

Although variables are unbound by \texttt{not/1}, note that any side effects are left in place:

\begin{verbatim}
?- \texttt{not (write( hello ), nl).}
\end{verbatim}
Care should be taken to distinguish `not/1`, which implements "logical negation", from `\+/1`, which implements "negation as failure". In the latter case, if the goal being tested succeeds by binding one or more variables, `\+/1` simply fails, undoing the bindings; in the former case, `not/1` generates an error.

Most early Prolog systems implemented `not/1` as negation as failure, and one of the commonest problems that arise when porting code to `WIN-PROLOG` is that this predicate gives errors in cases where it used to fail. In general, when adapting old code for modern Prologs, all calls to `not/1` should be replaced with calls to `\+/1`.

notrace\(\/0\)

*set the debugging (trace) mode to "off"*

\[\text{notrace}\]

**Comments**

This predicate sets the *debugging mode* to "off", which will prevent the debugger being invoked even if a new command is entered at the console or a predicate is called for which a spypoint is currently set.

**Examples**

The following call sets the debugging mode to "off" so that the debugger will not take over when a subsequent command is entered at the console:

\[
\begin{align*}
\text{?- notrace.} & \quad \text{yes} \quad \text{<enter>}
\end{align*}
\]

**Notes**

The debugging mode has three settings: "off", "debug", and "trace". In the first case, the debugger is never invoked, while in the second, it is invoked if a spied predicate is called. In the third case, the debugger is invoked immediately a query is entered at the console prompt.

The three debugging modes are mutually exclusive, but are somewhat confusingly set by four predicates, as shown in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>nodebug/0 or notrace/0</td>
</tr>
<tr>
<td>debug</td>
<td>debug/0</td>
</tr>
<tr>
<td>trace</td>
<td>trace/0</td>
</tr>
</tbody>
</table>

The *nodebug/0* and *notrace/0* predicates are effectively identical, and either can be used to cancel the effects of either *debug/0* or *trace/0*. Please note that WIN-PROLOG's default debugging mode is "debug".
number/1

test whether a term is a number

\texttt{number( \textit{Term} )}

\texttt{?Term} \quad \texttt{<term>}

Comments  This succeeds if \textit{Term} is a number.

Examples The following calls test various cases:

\texttt{?- number( 123 ).}
yes

\texttt{?- number( 123.45 ).}
yes

\texttt{?- number( \texttt{foo} ).}
no

\texttt{?- number( \texttt{`123.45`}).}
no

\texttt{?- number( X ).}
no

Notes The arithmetic evaluator in WIN-PROLOG treats all numbers simply as such: numbers. It automatically converts integers to floats when required, and always attempts to represent the final result as an integer, for example:

\texttt{?- X is 67.5 * 1.2.}
\texttt{X = 81}

Notice that the result, "81", is an integer, and not a float. Conversely, it is possible to generate floats when computing with integers:

\texttt{?- X is 22 / 7.}
\[ X = 3.14285714285714 \]

Except when calling built-in predicates, most of which require true integers as their numeric inputs, there is no need to distinguish between integers and floats in **WIN-PROLOG**.
number_atom/2

convert between a number and an atom

number_atom( Number, Atom )

?Number <number> or <variable>
?Atom <atom> or <variable>

Comments  This converts between numbers and atoms corresponding to their printed representation. Where Number is an unbound variable, and Atom is bound to an atom containing the printed representation of a number, that number is "read" from the atom and returned as Number; Where Number is already a number, it is "written" to an atom which is unified with Atom.

Examples  The following call returns the printed representation of the number, "123":

?- number_atom( 123, A ).
A = '123'

The next call returns a number from the given representation:

?- number_atom( N, '1.23e2' ).
N = 123

This call fails, because the printed representation of the number "123" is not "1.23e2":

?- number_atom( 123, '1.23e2' ).
no

Notes  As can be seen in the third example above, care should be taken when checking numbers against their printed representations. Where both arguments are given, the number is written out to an "ideal" representation, and an attempt is then made to unify this with the given representation.

This predicate is one of a trio of three: between them, number_atom/2, number_chars/2 and number_string/2 convert between numbers and their atom, char list and string representations respectively.
number_chars/2

convert between a number and a char list

\[\text{number_chars( Number, List )}\]

\(?\text{Number} \quad <\text{number}> \text{ or } <\text{variable}>\)
\(?\text{List} \quad <\text{char_list}> \text{ or } <\text{variable}>\)

Comments

This converts between numbers and char lists corresponding to their printed representation. Where \textit{Number} is an unbound variable, and \textit{List} is bound to a char list containing the printed representation of a number, that number is "read" from the list and returned as \textit{Number}; Where \textit{Number} is already a number, it is "written" to a char list which is unified with \textit{List}.

Examples

The following call returns the printed representation of the number, "123":

\(<\text{enter}>\)
\(?-\text{number_chars( 123, L ).}\)
\(L = [49,50,51]\)

The next call returns a number from the given representation:

\(<\text{enter}>\)
\(?-\text{number_chars( N, "1.23e2" ).}\)
\(N = 123\)

This call fails, because the printed representation of the number "123" is not "1.23e2":

\(<\text{enter}>\)
\(?-\text{number_chars( 123, "1.23e2" ).}\)
\no\)

Notes

As can be seen in the third example above, care should be taken when checking numbers against their printed representations. Where both arguments are given, the number is written out to an "ideal" representation, and an attempt is then made to unify this with the given representation.

This predicate is one of a trio of three: between them, \textit{number_atom/2}, \textit{number_chars/2} and \textit{number_string/2} convert between numbers and their atom, char list and string representations respectively.
**number_string/2**

*convert between a number and a string*

```
number_string( Number, String )
```

?Number          <number> or <variable>
?String          <string> or <variable>

**Comments**

This converts between numbers and strings corresponding to their printed representation. Where *Number* is an unbound variable, and *String* is bound to a string containing the printed representation of a number, that number is "read" from the string and returned as *Number*; Where *Number* is already a number, it is "written" to a string which is unified with *String*.

**Examples**

The following call returns the printed representation of the number, "123":

```
?- number_string( 123, S ).
S = `123`
```

The next call returns a number from the given representation:

```
?- number_string( N, `1.23e2` ).
N = 123
```

This call fails, because the printed representation of the number "123" is not "1.23e2":

```
?- number_string( 123, `1.23e2` ).
no
```

**Notes**

As can be seen in the third example above, care should be taken when checking numbers against their printed representations. Where both arguments are given, the number is written out to an "ideal" representation, and an attempt is then made to unify this with the given representation.

This predicate is one of a trio of three: between them, *number_atom/2*, *number_chars/2* and *number_string/2* convert between numbers and their atom, char list and string representations respectively.
numbervars/3

**instantiate all variables in a given term**

\[ \text{numbervars}( \text{Term}, \text{First}, \text{Last} ) \]

- **Term**: <term>
- **First**: <integer>
- **Last**: <integer> or <variable>

**Comments**
This takes a given \textit{Term}, and replaces all unbound variables in that term with tuples of the form `'\$VAR'(N)'`, where "N" is an integer; the first such integer used must be specified as \textit{First}, while a value one greater than the last integer used is eventually unified with \textit{Last}.

**Examples**
The following call binds variables in the term, "foo(A,B,A)", starting with the integer "81":

\[ \text{?- numbervars( foo(A,B,A), 81, L ).} \]

\[ A = '\$VAR'(81) , \]
\[ B = '\$VAR'(82) , \]
\[ L = 83 \]

**Notes**
The \textit{numbervars/3} predicate provides a useful, if somewhat primitive method of binding all variables in a term in a format which allows them to be subsequently identified. In particular, a term or subterm of the form `'\$VAR'(N)'`, cannot later be distinguished from grounded variables. \textbf{WIN-PROLOG} provides an alternative, more powerful predicate for handling variables in terms: \textit{vars/2} returns a flat list of variables, together with suitable “pretty print” names, which can be used in conjunction with the original term to perform many kinds of advanced structure processing.
occurs_chk/2

test whether a term includes the given variable

\[
\text{occurs_chk}( \text{Term, Var})
\]

?Term <term>
?Var <variable>

Comments
This is used to check whether a given variable, \( \text{Var} \), occurs in the given \( \text{Term} \).

Examples
The following call succeeds, because the specified variable, "B", is in the term "foo(A,B,A)"

\[
?- \text{occurs_chk}(\text{foo(A,B,A)}, \text{B}).
\]

\[
A = _ ,
B = _
\]

Conversely, the next call fails, because the variable, "C", is not present in the term:

\[
?- \text{occurs_chk}(\text{foo(A,B,A)}, \text{C}).
\]

no

Notes
The \text{occurs_chk/2} predicate is slightly unusual, in that unlike most other Prolog predicates, it is not designed to find matches for a given variable. Contrast the behaviour of the second example above with that of \text{member/2}, which in a similar call can return three "bindings" to "C", albeit to other variables:

\[
?- \text{member}(\text{C}, \text{[A,B,A]}).
\]

\[
C = A = _ ,
B = _ ;
\]

\[
C = B = _ ,
A = _ ;
\]

\[
C = A = _ ,
B = _ ;
\]
The process of "occurs checking" is useful in theorem provers, and in advanced unification routines where it is desired to avoid self-referential, "infinite" terms.
**one/1**

*call a goal once, preventing backtracking on failure*

```prolog
one( Goal )
+Goal <goal>
```

**Comments**

This calls the given Goal, and succeeds or fails depending upon whether the Goal itself succeeds or fails; any choice points introduced by the Goal are removed upon completion, in order to avoid backtracking into the call on subsequent failure. It is defined simply as the program:

```prolog
one( Goal ) :-
    Goal,
    !.
```

**Examples**

The following command invokes the member/3 predicate from within one/1, thereby limiting it to a single solution:

```prolog
?- one( member(X,[1,2,3]) ).
X = 1
```

**Notes**

In many respects, this predicate is similar to call/1: the only difference is that it makes a local call to "cut" (/!) immediately after executing the Goal. The following call is effectively the same as the above example:

```prolog
?- call( (member(X,[1,2,3]),!), ).
X = 1
```

The "if:then" predicate, -/2, also performs a local cut, and is sometimes used to perform the same job as one/1, as shown in the next command:

```prolog
?- member( X, [1,2,3] ) -> true.
X = 1
```
### op/3

**set or clear the definition of an operator**

\[
\text{op( Precedence, Type, Name )}
\]

+Precedence \[<\text{integer}> \text{ in the range } [0..1200]\]
+Type \[<\text{atom}>\]
+Name \[<\text{atom}>\]

**Comments**

This predicate can be used to set, reset or clear the definition of an operator, using the given `Precedence`, `Type` and `Name`. Setting a `Precedence` of zero (0) clears an operator's definition for the given `Type`.

**Examples**

The following call defines the word "plus" as an infix operator:

```prolog
?- op( 500, yfx, plus ).
yes
```

Once this command has been entered, "plus" can now be used to write terms such as the following:

```prolog
?- write( 1 plus 2 plus 3 ), nl.
1 plus 2 plus 3
yes
```

**Notes**

Operators are critical components of Prolog syntax, and define how terms are assembled internally during calls to `read/1`, and how they are output by `write/1`, `writeq/1` and so on. In expressions with mixed operators, those with lower precedences bind more tightly than those with higher ones.

For example, the "+" operator has an infix precedence of 500; the precedence of "*" is lower, at 400. With this relationship, the term "(1 * 2 + 3)" is translated to "+(*(1,2),3)" during input. Suppose, however, that the following call was made to `op/3`, to increase the infix precedence of "*" to 600:

```prolog
?- op( 600, yfx, * ).
yes
```

Because this call has given "*" a higher precedence than "+", the term "(1 * 2 + 3)" will now be translated to "*(1,+(2,3))" during input.
**open/2**

*open a file with the given access mode*

\[
\text{open( FileSpec, Mode )}
\]

+FileSpec <file_spec>
+Mode <atom>

**Comments**
This opens the disk file specified by `FileSpec` in preparation for subsequent input or output. The `Mode` defines how the file is opened, as shown in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>if the file already exists, open it for output and append new data to its end; otherwise, create a new file and open it for output</td>
</tr>
<tr>
<td>read</td>
<td>if the file already exists, open it for input; otherwise, raise an error or fail according to the setting of the file errors flag</td>
</tr>
<tr>
<td>write</td>
<td>if the file already exists, delete it; then create a new file and open it for output</td>
</tr>
</tbody>
</table>

**Examples**
The following command creates a new file, called "foo", preparing it for subsequent output:

\[
?\text{- open( foo, write ).}
\]

yes

In order to write to this file, a call must be made to `tell/1`:

\[
?\text{- tell( foo ), write( 'Hello World' ), nl, told.}
\]

yes

**Notes**
Files opened with `open/2` can be closed by calling `close/1`; normally `see/1` and `tell/1` handle the opening of files relatively automatically, but the present predicate allows, for example, an existing file to be rewritten from scratch, or reopened for reading from the top.

A weakness of `open/2` is that it does not allow for files with simultaneous read and write access. For such files, use should be made of the `fcreate/5` predicate, which also allows memory files to be defined for high-speed local applications.
This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used. This automatic expansion of file names can be shown by the following command:

```prolog
?- open( foo, write ), fdict( 0, X ), close( foo ).
X = ['c:\pro386w\foo']
```

Both open/2 and close/1 take logical file names; here "foo" is automatically expanded to "c:\pro386w\foo", as shown in the value of "X" as returned by fdict/2.

Please see the "notes" section of absolute_file_name/2 for further discussion about file names.
open/3

open a file with the given mode and character encoding

\[ \text{open( FileSpec, Mode, Code )} \]

+FileSpec  <file_spec>
+Mode      <atom>
+Code      <atom>

Comments
This opens the disk file specified by FileSpec in preparation for subsequent input or output. The Mode defines how the file is opened, as described in the entry for open/2; the Code defines which of eleven character encodings to use while accessing the file, as shown in the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>7-bit ASCII text</td>
</tr>
<tr>
<td>iso</td>
<td>8-bit ISO/IEC 8859-1 text (or 8-bit binary)</td>
</tr>
<tr>
<td>raw_16_be</td>
<td>16-bit raw (binary) data, big endian</td>
</tr>
<tr>
<td>raw_16_le</td>
<td>16-bit raw (binary) data, little endian</td>
</tr>
<tr>
<td>raw_32_be</td>
<td>32-bit raw (binary) data, big endian</td>
</tr>
<tr>
<td>raw_32_le</td>
<td>32-bit raw (binary) data, little endian</td>
</tr>
<tr>
<td>utf_16_be</td>
<td>20.1-bit Unicode text as UTF-16, big endian</td>
</tr>
<tr>
<td>utf_16_le</td>
<td>20.1-bit Unicode text as UTF-16, little endian</td>
</tr>
<tr>
<td>utf_32_be</td>
<td>20.1-bit Unicode text as UTF-32, big endian</td>
</tr>
<tr>
<td>utf_32_le</td>
<td>20.1-bit Unicode text as UTF-32, big endian</td>
</tr>
<tr>
<td>utf_8</td>
<td>20.1-bit Unicode text as UTF-8</td>
</tr>
<tr>
<td>automatic</td>
<td>20.1-bit Unicode or 8-bit ISO/IEC 8859-1 text</td>
</tr>
</tbody>
</table>

Examples
The following command creates a new file, called "foo", preparing it for subsequent output in Unicode UTF-16LE:

\[ \text{?- open( foo, write, utf}_16\text{le ).} \]

yes
In order to write to this file, a call must be made to \texttt{tell/1}:

\begin{verbatim}
?- \texttt{tell( foo ), write( 'Hello World' ), nl, told.}
yes
\end{verbatim}

\textbf{Notes}

Files opened with \texttt{open/3} can be closed by calling \texttt{close/1}; normally \texttt{see/1} and \texttt{tell/1} handle the opening of files relatively automatically, but the present predicate allows, for example, an existing file to be rewritten from scratch, or reopened for reading from the top.

A weakness of \texttt{open/3} is that it does not allow for files with simultaneous read and write access. For such files, use should be made of the \texttt{fcreate/5} predicate, which also allows memory files to be defined for high-speed local applications.

Unlike \texttt{fcreate/5}, the present predicate opens or creates Unicode files with a "Byte Order Mark", or "BOM", at the beginning: this reserved character behaves as a "mini-header", which subsequently can be used to determine whether a file contains Unicode, and if so, in which of the five UTF formats it is encoded. When an existing file is opened with the "automatic" setting, it is the presence or not of the BOM that allows \texttt{open/3} to determine the correct file character encoding: if the BOM is not present in any of the five Unicode encodings, the file is assumed to be in ISO/IEC 8859-1 format. When used to create a new file, "automatic" is treated as being synonymous with "utf_16_be".

This is one of a series of file handling predicates which make use of "logical" file names: \texttt{FileSpec} can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to \texttt{file_search_path/2}, and are expanded into atom file names by the \texttt{absolute_file_name/3} predicate before being used. This automatic expansion of file names can be shown by the following command:

\begin{verbatim}
?- \texttt{open( foo, write, iso ), fdict( 0, X ), close( foo ).}
X = ['c:\pro386w\foo']
\end{verbatim}

Both \texttt{open/3} and \texttt{close/1} take logical file names; here "foo" is automatically expanded to "c:\pro386w\foo", as shown in the value of "X" as returned by \texttt{fdict/2}.

Please see the "notes" section of \texttt{absolute_file_name/2} for further discussion about file names.
**opnbox/5**

*display the "open file" common dialog box*

```prolog
opnbox( Title, Filters, Name, Extension, Files )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>&lt;term&gt;</td>
</tr>
<tr>
<td>Filters</td>
<td>&lt;list_of(&lt;term&gt;,&lt;term&gt;)&gt;</td>
</tr>
<tr>
<td>Name</td>
<td>&lt;term&gt;</td>
</tr>
<tr>
<td>Extension</td>
<td>&lt;term&gt;</td>
</tr>
<tr>
<td>Files</td>
<td>&lt;list&gt; or &lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**

This displays a standard Windows "open box" ("open file") common dialog box, with the given Title, list of Filters and default Name and Extension, and then allows the user to browse for or enter the name(s) of one or more files. The Filters comprise a list of comma pairs of terms, normally strings or atoms, which describe file types and their extensions. The dialog closes when the "OK" button is clicked, if a file is double clicked, or if `<enter>` is pressed while the dialog's edit window has focus: at this point, a list of all selected or named files is unified with Files.

**Examples**

The following program shows a typical call to opnbox/5:

```prolog
foo( Files ) :-
    opnbox( `Open File(s)...`,
        [ (`Source`,`*.pl`),
          (`Object`,`*.pc`),
          (`Prolog`,`*.pl *.pc`) ],
        `default`,
        `pl`,
        Files )
```

Once compiled, a call to this program will display the "open file" common dialog box, with the title "Open File(s)...", and a set of filters which give the options of listing "Source" (*.pl), "Object" (*.pc) or "Prolog" (*.pl and *.pc) files. The name "default" will be preloaded into the edit window, and any filename entered without an extension will be given ".pl" (note that the dot (.) must not be included explicitly). Upon completion, the list of one or more selected file names is returned:
?- foo( Files ).
    Files = [\'c:\pro386w\examples\lunar.pl\']

Notes

Three common dialog box predicates are dedicated to obtaining filenames from the user: brsbox/2, opnbox/5, savbox/5. The first of these is used to return a folder name, while the latter pair share the majority of their features, but as their names suggest, one is used when opening files for loading (input), and the other when creating files for saving (output). This leads to the main differences in their respective behaviours, as detailed in the following paragraphs.

When loading, compiling or otherwise processing existing files, it is often desirable to be able to select several files at once; furthermore, it is essential that any such files are known to exist, and to be available for read access: opnbox/5 both allows multiple file selection, and performs its own existence and access permission checks on all selected files.

When "saving as" or otherwise creating new files, it is usually desirable to specify just one file at a time; furthermore, it is helpful to display a warning if a named (selected) file already exists, to avoid the user inadvertently overwriting important data: savbox/5 both limits selections to one file at a time, and displays a warning if the chosen file exists, giving the user the option to try again.

When searching for file locations other than when saving or loading, it can be handy to be able to select a folder directly, rather than one or more files contained within it: brsbox/2 provides exactly this feature.

None of these predicates actually opens the file: it is up to the application to pass the resulting file names to predicates such as open/2, fcreate/5, see/1, tell/1, and so on.
optimize/1

compile an existing predicate with the optimising compiler

\[ \text{optimize( Pred )} \]

+Pred <pred_spec>

Comments
This invokes the optimising compiler to compile an existing predicate, that has previously been compiled by the incremental compiler. When files are "consulted" (loaded with consult/1), their predicates are incrementally compiled: optimising such predicates improves their run-time performance.

Examples
The following command loads one of the example programs, "easter.pl", which contains a single predicate, easter/3:

\[ ?- \text{consult( examples(easter) ).} \]
\[ # 0.000 \text{ seconds to consult easter.pl [c:\pro386w\examples\]} \]
\[ \text{yes} \]

The easter/3 predicate can be optimised as follows:

\[ ?- \text{optimize( easter/3 ).} \]
\[ \text{yes} \]

Notes
The optimize/1 predicate can only optimise programs which are currently incrementally compiled, and furthermore are defined neither as "dynamic" (see dynamic/1) nor "multifile" (see multifile/1). Both these latter classes of predicate need to remain incrementally compiled, so that clauses can be added to or removed from their definitions when required. If an attempt is made to optimise anything other than a static, incrementally compiled predicate, an error will be signalled.

Code compiled with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed. Another benefit of optimised compilation is the possibility of using multiple-argument indexing: this is described in the entry for index/2; in general, use of this latter predicate together with optimize/1 provides a useful method of "tuning" programs at run time.
optimize_files/1

compile one or more files with the optimising compiler

optimize_files( Files )
  +Files <file_spec> or <list>

Comments
This invokes the optimising compiler to compile one or more Files; any predicates declared as "dynamic" or "multifile" are automatically compiled by the incremental compiler. Files may comprise a single file specification, in which case the source (.PL) file of the given name is compiled into the matching object (.PC) file; alternatively, by giving a pair of file specifications separated by a hyphen, the source and object files can be separately defined. Finally, a list of one or more of the above two types can be specified.

Examples
The following command compiles one of the example programs, "easter.pl" with the optimising compiler:

?- optimize_files( examples(easter) ).  
  yes

Notes
The optimize_files/1 predicate compiles directly from file to file, but does not load the programs: in order to use the resulting object file(s), they must be subsequently loaded with ensure_loaded/1 or load_files/1. Because these latter two predicates have a "load most recent" feature that uses file names to match pairs of source and object files, it is best to avoid the "source-object" option in optimize_files/1.

Code compiled with the incremental compiler can be debugged and listed, while code loaded with the optimising compiler runs up to three times faster, but cannot be debugged or listed. Another benefit of optimised compilation is the possibility of using multiple-argument indexing: this is described in the entry for index/2.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
**otherwise/0**

An alternative "no-op" predicate that simply succeeds

```
otherwise
```

**Comments**
This predicate simply succeeds.

**Examples**
The following call succeeds:

```
?- otherwise.
yes
```

**Notes**
The `otherwise/0` predicate is a historical curiosity, included for compatibility with Quintus Prolog, that does exactly the same job as the more widely known predicate, `true/0`. Like the latter, it is defined simply as a fact that is true:

```
otherwise.
```
outpos/1

get or set the output stream pointer

\[
\text{outpos}( \text{Position} )
\]

?Position \text{<integer>, <float> or <variable>}

Comments
This gets or sets the Position of the output (write) pointer in the current output stream. When called with an unbound variable, the existing Position is returned as an integer; if called with an integer, the output pointer is set to the given Position.

Examples
The following call writes three characters, then skips back to the start and writes one character, putting the result into a string with the help of the output redirection predicate, \~>/2:

?- (write( foo ), outpos( 0 ), write( b )) \~> S.  
S = `boo`

The next call writes a first term into a string, and then returns the new output position before writing a second:

?- (write( foo ), outpos( P ), write( bar )) \~> S.  
P = 3 ,  
S = `foobar`

Notes
The inpos/1 and outpos/1 predicates can be used respectively to reposition the input (read) and output (write) pointer for any disk file, memory file or input or output string, but neither can be used on serial devices like the user input/output stream.

Please note that Windows does not support independent read and write pointers in files, so that in any file opened simultaneously for input and output, inpos/1 and outpos/1 will both manipulate the same, read/write pointer.

Starting with Win-Prolog 6.000, file sizes are no longer limited to $2^{32}$-1 bytes in size: "Fat" files of up to $2^{53}$-1 bytes can be created, and randomly positioned using inpos/1 and outpos/1 for reading and writing respectively. This number equates to 9,007,199,254,740,991 bytes, or just over 9 petabytes (9000 terabytes) in size, and is the current limit since this is the biggest whole number than can be stored and computed with integer precision in a 64-bit floating point number.
**output/1**

*get or set the output stream*

```
output( Stream )
```

?Stream <stream> or <variable>

**Comments**

This gets or sets the current output Stream. When called with an unbound variable, the existing Stream is returned; if called with a stream identifier, this is set as the current output Stream. Streams may be specified as integers (for various types of user output), atoms (naming currently open disk or memory files; see `fcreate/5`), or conjunctions of a string and an integer, specifying an output string and its output pointer. These are summarised in the following table:

<table>
<thead>
<tr>
<th>Stream</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>buffered console (user) output</td>
</tr>
<tr>
<td>1</td>
<td>buffered console output (effectively as above)</td>
</tr>
<tr>
<td>2</td>
<td>direct console output (no processing of bell)</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>the name of a currently open disk or memory file</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>a pair of (&lt;string&gt;,&lt;integer&gt;) where the string is &quot;written&quot; like an output device, and the integer specifies the current output (write) pointer</td>
</tr>
</tbody>
</table>

**Examples**

The following call writes a term to the direct console output device, which displays "bell" character (<ctrl-G>), rather than producing a bell tone:

```
?- output( 2 ), write(`bell-G`), nl, output( 0 ).
```

Bell

yes

The next call writes the character "b" into the string "foo", starting at its first character:

```
?- output( (`foo`,0) ), write( b ).
```

yes

The result of the above call can be picked up, and the output stream reset, as follows:
?- output( X ), output( 0 ).
X = (`boo`,1)

Assuming a file called "foo" has previously been opened for output (see fcreate/5), the following call allows the letter "a" to be written to file (note the use of output/1 both to preserve the current output stream and set the new one):

?- output( C ), output( foo ), write( a ), output( C ).
C = 0

Notes

The output/1 predicate provides low-level control over the output stream. It is similar in many ways to the traditional tell/1 and telling/1 predicates, although with several more powerful features. Unlike tell/1, the output/1 predicate only works with existing files and streams: it will not automatically open a file that is named for the first time. By leaving the opening of files to an explicit predicate, fcreate/5, and the positioning of its output (write) pointer to another predicate, outpos/1, output/1 is considerably more transparent than tell/1.

Every type of output stream supported by WIN-PROLOG, from user devices through disk and memory files to string output, can be correctly managed by output/1. Any program that wishes to temporarily switch output streams should simply "bracket" the switch by picking up the current stream, and resetting it when finished. Consider the following example:

write_to_stream( Stream, Term ) :-
    output( Current ),
    output( Stream ),
    write( Term ),
    output( Current ).

The existing output stream is picked up in the variable "Current", before a switch is made to the desired new output stream, "Stream". A "Term", is then written to this stream, before output is restored to the original stream, "Current".
pause/1

pause for the given number of milliseconds

\[
\text{pause}(\ Time\ )
\]

+\text{Time} <\text{integer}>

**Comments**  
This pauses execution for a \text{Time} expressed as an integer number of milliseconds.

**Examples**  
The following call performs writes to the console, before outputting a new line and pausing for 5 seconds; the call to flush/0 between the output of "hello" and the output of "world", ensures that the former is displayed immediately; the latter (and the new line) will only be displayed when the 5 second delay is up, and WIN-PROLOG is waiting for console input:

\[
\text{?- write( \text{hello} ), flush, write( \text{world} ), nl, pause( 5000 ).}<\text{enter>}
\]

eelloworld

**Notes**  
The \text{pause/0} predicate provides a convenient way to hold back execution for a given period, for example to display some information in the console long enough for it to be read, or to time a slideshow application or suchlike. This predicate is new in WIN-PROLOG 5.000; previously, the only way to perform a pause was to contrive a program such as:

\[
\text{my_pause( \text{Time} ) :-}
\begin{align*}
\text{ms( \text{repeat, Elap },}
\text{Elap >= Time}
\text{-> true}
\end{align*}
\]

Like the above soft solution, \text{pause/1} can be interrupted and resumed by normal Windows messages and programmable timers, allowing the main evaluation thread to be held for a period while regular background process can continue unhindered.
pdict/4

return a list of predicates matching the given type and mask

\[
pdict( \text{Flag}, \text{Type}, \text{Mask}, \text{Dict} )
\]

- **Flag** <integer> in the domain \{-1, 0, 1\}
- **Type** <integer> in the range \{-1..5\}
- **Mask** <integer>
- **Dict** <variable>

Comments

This succeeds by binding \text{Dict} to a list of all predicates, as specified in the given \text{Flag} (see \text{dict/2}), which are of the given \text{Type} and whose predicate flags are selected by the given \text{Mask}. Each predicate in the list is represented as a conjunction of (atom, integer), where the atom represents a predicate name and the integer represents its arity. Acceptable values of predicate \text{Type} are shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Any predicate</td>
</tr>
<tr>
<td>0</td>
<td>No definition, but one or more predicate flags</td>
</tr>
<tr>
<td>1</td>
<td>Written in Prolog, and incrementally compiled</td>
</tr>
<tr>
<td>2</td>
<td>Written in Prolog, and hashed</td>
</tr>
<tr>
<td>3</td>
<td>Written in Prolog, and optimised</td>
</tr>
<tr>
<td>4</td>
<td>Written in Assembler, and built-in</td>
</tr>
<tr>
<td>5</td>
<td>Written in Assembler, and dynamically loaded</td>
</tr>
</tbody>
</table>

Five predicate flags have predefined meanings within the system, which leaves eleven available for user-defined purposes (the cross reference utility, "xrefs.pc", for example, borrows the top four flags). The \text{Mask} encodes those file flags to include or exclude in a search, by adding together the values in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000001</td>
<td>include locked predicates (static)</td>
</tr>
<tr>
<td>16'00000002</td>
<td>include predicates which fail if empty (dynamic)</td>
</tr>
<tr>
<td>16'00000004</td>
<td>include predicates with spypoints set</td>
</tr>
<tr>
<td>16'00000010</td>
<td>include overlay predicates</td>
</tr>
</tbody>
</table>
**Examples**

The following call creates a single incrementally compiled clause for a dynamic user predicate, `foo/2`:

```prolog
?- assert( foo(hello,world) ).
yes
```

This, and any other predicates with flag "2" set, and flag "16" clear, corresponding to user-level dynamic predicates, can be returned by the call:

```prolog
?- pdict( 0, 1, 16'00100002, D ).
D = [(foo,2)]
```

**Notes**

Predicate types are also used in `def/3`, which checks or returns the type of any predicate associated with a given name and arity. Predicate flags themselves are set by various predicates, such as `spy/1` and `dynamic/1`, as well as being handled by the special `addflg/3`, `delflg/3` and `getflg/3` predicates. A list of arities for a given predicate name can be retrieved with the `defs/2` predicate.

The flag mask used by `pdict/4` is a 32-bit integer that really comprises two separate 16-bit masks. The low-order 16 bits define any predicate flags which must be present in order for a predicate to qualify for inclusion; the high-order 16 bits define any such flags which must not be present in order for a predicate to qualify. The more bits that are set, the more specific will be the predicate selection; conversely, if no bits are set, all predicates will qualify. In order to see all predicates matching the given `Type`, simply specify a `Mask` of zero ("0").

An error will be raised if any given bits are set both in the low and high halves of the mask, because it is not possible both to include and exclude predicates based on the same flag.
phrase/2

parse or generate an exact phrase for a given grammar

phrase( Grammar, List )

+Grammar <atom> or <compound>
?List <list> or <variable>

Comments This applies any user-defined grammar rules to the term given in Grammar and attempts to unify List with an appropriate phrase; alternative phrases can be generated on backtracking.

Examples Consider the following grammar rules:

sentence --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun.
verb_phrase --> verb ; verb, noun_phrase.
determiner --> [the].
noun --> [boy] ; [house].
verb --> [likes].

Once compiled, various queries can be made:

?- phrase( sentence, [the,boy,likes,the,house] ). yes

?- phrase( verb_phrase, [the,boy] ). no

?- phrase( verb_phrase, S ).
S = [likes] ;
S = [likes, the, boy] ;
S = [likes, the, house]
Grammar rules are a feature only in some Prolog systems, and are designed to facilitate the parsing of natural language. The `phrase/2` and `phrase/3` predicates operate simply by applying term expansion to the given Grammar rule, and then calling the resulting program to test or generate the List. The former predicate requires that the given List is a complete phrase in the specified Grammar; the latter returns the tail of the list for further processing. See the entries for `expand_term/2` and `expand_dcg/2` for further information on term expansion.
phrase/3

parse or generate a partial phrase for a given grammar

\textit{phrase}( \text{Grammar}, \text{List}, \text{Tail} )

+Grammar \hspace{1cm} <atom> \text{ or } <compound>
?List \hspace{1cm} <list> \text{ or } <variable>
?Tail \hspace{1cm} <list> \text{ or } <variable>

Comments
This applies any user-defined grammar rules to the term given \text{Grammar} and attempts to unify \text{List} with an appropriate phrase; alternative phrases can be generated on backtracking. After the parser has processed as much of the \text{List} as possible, its remainder is unified with \text{Tail}.

Examples
Consider the following grammar rules:

\text{sentence} \quad \text{-->} \quad \text{noun\_phrase, verb\_phrase.}
\text{noun\_phrase} \quad \text{-->} \quad \text{determiner, noun.}
\text{verb\_phrase} \quad \text{-->} \quad \text{verb \mid verb, noun\_phrase.}
\text{determiner} \quad \text{-->} \quad \text{[the].}
\text{noun} \quad \text{-->} \quad \text{[boy] \mid [house].}
\text{verb} \quad \text{-->} \quad \text{[likes].}

Once compiled, various queries can be made:

?- \text{phrase}( \text{sentence}, \text{[the,boy,likes,the,house]}, \text{T} ). <enter>
T = \text{[the,house]} ; <space>
T = [ ] ; <space>
no

?- \text{phrase}( \text{verb\_phrase}, \text{[the,boy]}, \text{T} ). <enter>
no

Notes
Grammar rules are a feature only in some Prolog systems, and are designed to facilitate the parsing of natural language. The \text{phrase}/2 and
phrase/3 predicates operate simply by applying term expansion to the given Grammar rule, and then calling the resulting program to test or generate the List. The former predicate requires that the given List is a complete phrase in the specified Grammar; the latter returns the tail of the list for further processing. See the entries for expand_term/2 and expand_dcg/2 for further information on term expansion.
**popup_menu/4**

*display and track a popup menu in a given window*

```
popup_menu( Window, Menu, Xpos, Ypos )
```

**Comments**
This displays a Menu in the given Window at the specified client coordinates, \((Xpos, Ypos)\), and tracks the selection of a single item from this menu.

**Examples**
The following call will create an empty menu called "foo":

```
?- wmcCreate( foo ).
yes
```

The next call adds an item to "foo", with the label "Bar", where "B" is the hotkey, and menu item number "1000":

```
?- wmnuAdd( foo, 0, `&Bar`, 1000 ).
yes
```

This command displays the single item menu at position \((200,300)\) in the console edit window, \((1,1)\):

```
?- popup_menu( (1,1), foo, 200, 300 ).
yes
```

**Notes**
Messages from menus are processed in a window handler (see window_handler/2); in the case of popup menus, the reported handle will be that specified in the call to popup_menu/4.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**portray/1**

*user-definable predicate for extending term printing*

```prolog
portray( Term )
```

?Term <term>

**Comments**

If defined by the user, this predicate is called by `print/1` to try to output a `Term`: if the call succeeds, output is considered to be complete; otherwise, `write/1` is used to output the term.

**Examples**

Consider the following definition of `portray/1`:

```prolog
portray( 81 ) :-
    write( `eighty-one` ).
```

Once compiled, it will modify any attempt to write the term "81" using `print/1`, but will output other terms in the same way as `write/1`:

?- print( 80 ), nl.
80
yes

?- print( 81 ), nl.
eighty-one
yes

**Notes**

By default (in the absence of any user definition of `portray/1`), `print/1` outputs terms using "unquoted" Edinburgh syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally (see `ewrite/1` for further details).

When `portray/1` has been defined by the user, any terms or subterms for which this predicate succeeds are assumed to have be printed successfully. Any remaining terms or subterms for which `portray/1` fails are output using `write/1`. The supervisor prompt and system debuggers use a quoted variant `print/1`, so any definition of `portray/1` directly affects their output, for example:

?- X is 9 * 9 .
X = eighty-one
**portray_clause/**

*write a term in listing format to the current output stream*

```prolog
portray_clause( Term )
```

**Comments**

This writes the given *Term* to the current output stream, in quoted Edinburgh syntax, using a pretty-printed format based on that used by *listing/0* and *listing/1*.

**Examples**

The following command outputs the given term with quotes where necessary:

```
?- portray_clause( (foo(X):-bar(X)) ).
```

```
foo( A ) :-
  bar( A ).
X = _
```

**Notes**

This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (""), and any atoms beginning with the underscore (\_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see *eprint/1* for further information).

Both *portray_clause/1* and *portray_clause/2* perform sophisticated formatting of the output term, laying out subterms which represent nested code blocks using a carefully-designed indentation scheme which makes the resulting output easy to read. Unlike other term output predicates (*write/1*, *writeq/1*, etc), the two clause portrayal predicates also output the terminating period symbol (.) that allows terms to be input with the *read/1* predicate.

In addition to normal quoted output characteristics, *portray_clause/1* assigns default alphabetic names for any multiple-use variables, and a name of underscore (\_) for any single use (anonymous) variables (see *vars/2* for information about how these names are generated).
**portray_clause/2**

write a term in listing format and named variables

```prolog
portray_clause( Term, Vars )
```

**Comments**

This writes the given `Term` to the current output stream, in quoted Edinburgh syntax, using a pretty-printed format based on that used by `listing/0` and `listing/1`, naming any variables listed in `Vars` with their corresponding names.

**Examples**

The following command outputs the given term with quotes where necessary, using the given named variable:

```prolog
?- portray_clause( (foo(X):-bar(X)), [('Var',X)] ).
```

foo( Var ) :-
    bar( Var ).

X = _

**Notes**

This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes ('), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see `eprint/1` for further information).

Both `portray_clause/1` and `portray_clause/2` perform sophisticated formatting of the output term, laying out subterms which represent nested code blocks using a carefully-designed indentation scheme which makes the resulting output easy to read. Unlike other term output predicates (`write/1`, `writeq/1`, etc), the two clause portrayal predicates also output the terminating period symbol (.) that allows terms to be input with the `read/1` predicate.

The ability to preserve or create variable names during input and output is a key feature of WIN-PROLOG: predicates such as `eread/2` return the list of variable names used in a term, and these can then be used in predicates like `portray_clause/2`. See the entries for `eread/2` and `eprint/2` for further information.
**predicate_property/2**

*get or check the properties of current predicates*

```prolog
predicate_property( Head, Property )
```

<table>
<thead>
<tr>
<th>?Head</th>
<th>&lt;atom&gt;, &lt;tuple&gt; or &lt;variable&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Property</td>
<td>&lt;atom&gt; or &lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**

This matches a predicate to its properties. The predicate is specified as a *Head*, comprising a dummy goal which includes the predicate name and arbitrary argument values; the *Property* is one of the atoms in the table below. If either or both arguments is an unbound variable, successive solutions can be found on backtracking.

<table>
<thead>
<tr>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>built_in</td>
<td>the predicate is built in to the system</td>
</tr>
<tr>
<td>compiled</td>
<td>the predicate has been compiled with the incremental compiler</td>
</tr>
<tr>
<td>dynamic</td>
<td>the predicate has been declared dynamic</td>
</tr>
<tr>
<td>foreign</td>
<td>the predicate is implemented in a dynamically loaded assembler module</td>
</tr>
<tr>
<td>hashed</td>
<td>the predicate has been compiled with the hashing compiler</td>
</tr>
<tr>
<td>index(Arg)</td>
<td>the predicate will be indexed on the given argument when compiled with the optimising compiler</td>
</tr>
<tr>
<td>multifile</td>
<td>the predicate has been declared as multifile</td>
</tr>
<tr>
<td>optimized</td>
<td>the predicate has been compiled with the optimising compiler</td>
</tr>
<tr>
<td>spy</td>
<td>the predicate has a spypoint currently set</td>
</tr>
<tr>
<td>static</td>
<td>the predicate is static (not dynamic)</td>
</tr>
<tr>
<td>vestigial</td>
<td>the predicate exists but has no clauses</td>
</tr>
<tr>
<td>volatile</td>
<td>the predicate has been declared as volatile</td>
</tr>
</tbody>
</table>

**Examples**

On backtracking, the following call returns successive properties for the `consult/1` predicate (note that the given argument is a dummy placeholder only, and can have any value):

```prolog
?- predicate_property( consult(_), P ).
P = built_in ;
```
The following calls create a single clause for each of a pair of dynamic predicates, foo/2 and bar/1, and then return these predicates successively on backtracking (please note that the returned arguments are arbitrary dummy variables):

?- assert( foo(hello,there) ), assert( bar(world) )
  yes

?- predicate_property( P, dynamic )
  P = foo(_1,_2) ;
  P = bar(_3) ;
  no

**Notes**

The `predicate_property/2` predicate provides information about currently defined predicates, and simply fails if called with arguments that do not match any existing predicate. The format of its first argument is somewhat quirky, since elsewhere, predicates are usually specified as name and arity. For example, consider the output from `current_predicate/1`:

?- current_predicate( P )
  P = xrefs / 0 ;
  no

Further information about current predicates can be found using `def/3`, `defs/2`, `getflg/3` and `pdict/4`. 
prgbox/1

update the "progress" dialog box

\[ \text{prgbox}( \text{Setting} ) \]
+Setting <integer>

Comments
This updates or closes a previously initialised standard system "progress box" dialog, adjusting its progress bar and remaining time estimate according to the given Setting:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..Limit</td>
<td>adjust the progress bar between zero and &quot;Limit&quot;, where &quot;Limit&quot; has been set with prgbox/3</td>
</tr>
<tr>
<td>-1</td>
<td>close the progress box dialog</td>
</tr>
</tbody>
</table>

Examples
The following call displays and initialises the progress box, using the title, "Progress", setting its limit to 100, and enabling the "Cancel" button, before setting its position to 50, exactly half way to completion:

\[ \text{?- prgbox}( \text{`Progress`}, 100, 1 ), \text{prgbox}( 50 ). \]  
\[ <\text{enter}> \]

\[ \text{yes} \]

Any number of calls to prgbox/1 may follow, the value passed needing to be in the range [0..Limit], where "Limit" is the limit set in the initial call to prgbox/3. When finished, the following call destroys the progress box:

\[ \text{?- prgbox}( -1 ). \]  
\[ <\text{enter}> \]

\[ \text{yes} \]

Notes
The prgbox/1 and prgbox/3 predicates provide a convenient method for allowing users to monitor lengthy processes, with the option of allowing them to abort by clicking a "Cancel" button: if enabled, and this button is clicked, subsequent calls to prgbox/1 will fail, indicating to the process that it should stop. Should this happen, the process should call prgbox/1 one last time, with a value of "-1", to close the progress box.

Please note that the progress box is timer-sensitive, and to obtain the best estimate of "time remaining", it is essential to call prgbox/3 as near as possible to the start of the process that is to be monitored by subsequent calls to prgbox/1. As an example, consider the following program which "processes" all files matching a given patten (in this case, simply using a 100ms time delay):
process( Pattern ) :-
    dir( Pattern, 16'00100000, Files ),
    len( Files, Length ),
    prgbox( `Processing Files`, Length, 1 ),
    ( process( Files, 0 )
    -> write( `Successful!` )
    ; prgbox( -1 ),
      write( `Cancelled!` )
    ),
    nl.

process( [], _ ) :-
    prgbox( -1 ).

process( [File|More], Data ) :-
    prgbox( Data ),
    write( `Processing` - File ),
    nl,
    delay( 100 ),
    Next is Data + 1,
    process( More, Next ).

delay( Time ) :-
    ms( repeat, Value ),
    Value > Time,
    !.

Once compiled, the following command will display all the file names in "c:\windows", with a progress box that allows the sequence to be monitored and interrupted if required:

?- process( 'c:\windows\*.*' ).
Processing - c:\windows\ACCESSOR.GRP
...
Processing - c:\windows\ShellIconCache
Successful!
yes
Unlike certain other programming environment dialogs, such as those accessed through `prnbox/4`, `opnbox/5` and `savbox/5`, the present predicate is not a true "common dialog", being implemented directly in terms of **WIN-PROLOG**'s window handling predicates.
**prgbox/3**

*display the "progress" dialog box*

```
prgbox( Title, Limit, Mode )
```

<table>
<thead>
<tr>
<th>+Title</th>
<th>&lt;string&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Limit</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>+Mode</td>
<td>&lt;integer&gt; in the domain {0,1}</td>
</tr>
</tbody>
</table>

**Comments**

This displays and initialises a standard system "progress box" dialog, with the given Title, setting its and Limit and preparing its "Cancel" button according to the specified Mode:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>disable the &quot;Cancel&quot; button</td>
</tr>
<tr>
<td>1</td>
<td>enable the &quot;Cancel&quot; button</td>
</tr>
</tbody>
</table>

At the point this predicate is called, the system elapsed time is noted, and recorded along with the specified Limit; subsequent calls to the sister predicate, `prgbox/1`, cause a "progress bar" to be drawn, together with an estimate of remaining time.

**Examples**

The following call displays and initialises the progress box, using the title, "Progress", setting its limit to 100, and enabling the "Cancel" button, before setting its position to 50, exactly half way to completion:

```
?- prgbox( `Progress`, 100, 1 ), prgbox( 50 ).
yes
```

Any number of calls to `prgbox/1` may follow, the value passed needing to be in the range [0..Limit], where "Limit" is the limit set in the initial call to `prgbox/3`. When finished, the following call destroys the progress box:

```
?- prgbox( -1 ).
yes
```

**Notes**

The `prgbox/1` and `prgbox/3` predicates provide a convenient method for allowing users to monitor lengthy processes, with the option of allowing them to abort by clicking a "Cancel" button: if enabled, and this button is clicked, subsequent calls to `prgbox/1` will fail, indicating to the process that it should stop. Should this happen, the process should call `prgbox/1` one last time, with a value of "-1", to close the
progress box.

Please note that the progress box is timer-sensitive, and to obtain the best estimate of "time remaining", it is essential to call `prgbox/3` as near as possible to the start of the process that is to be monitored by subsequent calls to `prgbox/1`. Please see the "notes" section of `prgbox/1` for a worked example.

Unlike certain other programming environment dialogs, such as those accessed through `prnbox/4`, `opnbox/5` and `savbox/5`, the present predicate is not a true "common dialog", being implemented directly in terms of WIN-PROLOG's window handling predicates.
print/1

write an unquoted term with user syntax extensions

\[ \text{print( Term )} \]

?Term <term>

Comments  
This writes the given Term to the current output stream, in unquoted Edinburgh syntax, with special syntax extensions specified in the optional user-defined predicate, portray/1.

Examples  
Consider the following definition of portray/1:

\[ \text{portray( 81 ) :- write( `eighty-one` )}. \]

Once compiled, it will modify any attempt to write the term "81" using print/1, but will output other terms in the same way as write/1:

?- \text{print( 80 ), nl}. 
80
yes

?- \text{print( 81 ), nl}. 
eighty-one
yes

Notes  
By default (in the absence of any user definition of portray/1), print/1 outputs terms using "unquoted" Edinburgh syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally (see ewrite/1 for further details).

When portray/1 has been defined by the user, any terms or subterms for which this predicate succeeds are assumed to have be printed successfully. Any remaining terms or subterms for which portray/1 fails are output using write/1. The supervisor prompt and system debuggers use a quoted variant print/1, so any definition of portray/1 directly affects their output, for example:

?- \text{X is 9 * 9}. 
X = eighty-one
prnbox/4

display the "print/print setup" common dialog box

    prnbox( Mode, Name, Driver, Port )

+Mode        <atom> or <empty_list>
-Name        <variable>
-Driver      <variable>
-Port        <variable>

Comments
This displays a standard Windows "print box" ("print/print setup") common dialog box, in the given Mode, and then allows the user to modify printer settings, before returning the Name, Driver and Port of the selected printer. If Mode is the empty list ("[]"), then the "print setup" dialog is shown, and no print job is originated; otherwise, the "print" dialog is shown, and on success, initiates a print job named by Mode.

Examples
The following call initiates the print setup dialog, allowing settings to be modified without starting a print job (the predicate succeeds when the user clicks on the "OK" button or presses <enter> while the dialog is in focus):

    ?- prnbox( [], N, D, P ).
    N = 'HP DeskJet Plus' ,
    D = 'HPDSKJET' ,
    P = 'LPT1:'

The following call will initiate a print job called "foo", assuming that the user clicks "OK" or presses <enter>:

    ?- prnbox( foo, N, D, P ).
    N = 'HP DeskJet Plus' ,
    D = 'HPDSKJET' ,
    P = 'LPT1:'

Notes
There are two predicates in WIN-PROLOG that can initiate a print job: prnbox/4 and prnini/4. The latter directly initiates printing on the named printer, device driver and port, while the former displays a dialog that allows the user to adjust settings as appropriate. Once started, print jobs are controlled using three other predicates: prnpag/1, prnend/1 and prnstt/1; printer output itself is performed by the gfx*/n predicates (see gfx/1 for further information).
prnend/1

terminate a print job

prnend( Flag )

+Flag <integer> in the domain {0,1}

Comments

This is used to terminate a print job and release the printer for other applications. The value of Flag determines whether to close gracefully, or abruptly:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if a page is active (printer status is &quot;3&quot; (see prnss1/1)), then give an error; else print the current page and close the print job gracefully</td>
</tr>
<tr>
<td>1</td>
<td>abandon any unfinished pages, irrespective of status, and abort the print job abruptly</td>
</tr>
</tbody>
</table>

Examples

The following short program sets up a print job, called "Square", through a common dialog (prnbox/4), begins a page (prnpag/1), uses the gfx*/n predicates to draw a grey square on the printer, and then terminates the print job (prnend/1):

```prolog
print_square( Printer, Driver, Port, Page ) :-
  prnbox( 'Square', Printer, Driver, Port ),
  prnpag( Page ),
  gfx_begin( [] ),
  gfx( rectangle(400,400,800,800) ),
  gfx_end( [] ),
  prnend( 0 ).
```

Once compiled, the following command will print a single page containing a grey square near its top left corner:

```prolog
?- print_square( N, D, P, C ).
N = 'HP DeskJet Plus' ,
D = 'HPDSKJET' ,
P = 'LPT1:' ,
C = 1
```
Print jobs are set up using either `prnbox/4` or `prnini/4`, and are then controlled by `prnpag/1`, `prnend/1` and `prnstt/1`; printer output itself is performed by the `gfx*/n` predicates (see `gfx/1` for further information).
prnini/4

_Initialise the printer using a given setup_

\[
\text{prnini( Title, Name, Driver, Port )}
\]

+Title <atom>
+Name <atom>
+Driver <atom>
+Port <atom>

**Comments**

This initialises a print job with the given `Title`, and using the given printer `Name`, `Driver` and `Port`.

**Examples**

The following short program sets up a print job, called "Circle", with the specified, fixed setup (`prnini/4`), begins a page (`prnpag/1`), uses the `gfx*/n` predicates to draw a grey circle on the printer, and then terminates the print job (`prnend/1`):

\[
\begin{align*}
\text{print_circle( Page ) :-} \\
\quad \text{prnini( 'Circle', 'HP DeskJet Plus', 'HPDSKJET', 'LPT1:' ),} \\
\quad \text{prnpag( Page ),} \\
\quad \text{gfx_begin( [] ),} \\
\quad \text{gfx( ellipse(400,400,800,800) ),} \\
\quad \text{gfx_end( [] ),} \\
\quad \text{prnend( 0 ).}
\end{align*}
\]

Once compiled, the following command will print a single page containing a grey circle near its top left corner:

\[
?- \text{print_circle( C ).}
\]

\[
C \ = \ 1
\]

**Notes**

There are two predicates in **WIN-PROLOG** that can initiate a print job: `prnbox/4` and `prnini/4`. The latter directly initiates printing on the named printer, device driver and port, while the former displays a dialog that allows the user to adjust settings as appropriate. Once started, print jobs are controlled using three other predicates: `prnpag/1`, `prnend/1` and `prnstt/1`; printer output itself is performed by the `gfx*/n` predicates (see `gfx/1` for further information).
prnpag/1

begin a new page on a print job

\[\text{prnpag( Page )}\]

-Page <variable>

Comments
This is used to print and eject any current page, before preparing to print on a new page; the latter's number is returned as Page. If the current page is still active (printer status is "3" (see prnstt/1)), an error is generated.

Examples
The following short program sets up a print job, called "Square", through a common dialog (prnbox/4), begins a page (prnpag/1), uses the gfx*/n predicates to draw a grey square on the printer, and then terminates the print job (prnend/1):

\[\text{print_square( Printer, Driver, Port, Page ) :-}\]
\[\text{prnbox( 'Square', Printer, Driver, Port ),}\]
\[\text{prnpag( Page ),}\]
\[\text{gfx_begin( [] ),}\]
\[\text{gfx( rectangle(400,400,800,800) ),}\]
\[\text{gfx_end( [] ),}\]
\[\text{prnend( 0 ).}\]

Once compiled, the following command will print a single page containing a grey square near its top left corner:

\[?- \text{print_square( N, D, P, C ).}\]
\[\text{N = 'HP DeskJet Plus',}\]
\[\text{D = 'HPDSKJET',}\]
\[\text{P = 'LPT1:',}\]
\[\text{C = 1}\]

Notes
Print jobs are set up using either prnbox/4 or prmini/4, and are then controlled by prnpag/1, prnend/1 and prnstt/1; printer output itself is performed by the gfx*/n predicates (see gfx/1 for further information).
prnstat

get or check printer status

prnstat( Status )

?Status <variable> or <integer> in the range [0..3]

Comments
This is used to get or check the Status of the printer or current print job; the four possible values are described below:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>there is no current print job (neither prnbox/4 nor prnini/4 have been called, at least since any call to prnend/1)</td>
</tr>
<tr>
<td>1</td>
<td>a print job has been initiated by a call to either prnbox/4 or prnini/4, but is not yet underway (prnpag/1 has not been called to start a page)</td>
</tr>
<tr>
<td>2</td>
<td>a print job is underway, and the current page is inactive (any calls to gfx_begin/1 have been matched by calls to gfx_end/1)</td>
</tr>
<tr>
<td>3</td>
<td>a print job is underway, and the current page is active (one or more gfx_begin/1 calls have yet to be matched by calls to gfx_end/1)</td>
</tr>
</tbody>
</table>

Examples
The following call checks the current status of any print job, in this case returning "0", implying that there is no such job:

?- prnstat( S ).
S = 0

Notes
Print jobs are set up using either prnbox/4 or prnini/4, and are then controlled by prnpag/1, prnend/1 and prnstat/1; printer output itself is performed by the gfx*/n predicates (see gfx/1 for further information).

The "print status" determines when certain actions can take place. For example, a status of "1" means that the first page of the print job has yet to be requested, so it is not yet possible to perform any output; a status of "3", on the other hand, means that the current page is actively being written to, and until this operation is complete (status returns to "2"), it is not possible to request a new page or terminate the print job gracefully.
profile/1

```
get the name of the WIN-PROLOG initialisation file

    profile( -File )

-File <variable>
```

Comments

This returns the name of the WIN-PROLOG initialisation File, for subsequent use with the profile/4 predicate.

Examples

The following call picks up the name of the WIN-PROLOG initialisation file, and stores an entry, "bar=sux", in a section called "foo", in this file:

```
?- profile( File ), profile( File, foo, bar, sux ).
```

File = 'C:\PRO386W\PRO386W.INI'

Notes

Initialisation files are processed entirely by Windows itself, and profile/4 therefore makes no use of the WIN-PROLOG file input/output system. Where a given File name is specified without a path, it is assumed to refer to a file in the Windows directory, rather than in the current working directory.

The WIN-PROLOG system initially assumes its initialisation file resides in its own directory, and shares its name, apart from the file extension, which is "INI" rather than "EXE". To enable this file to be positioned elsewhere on a computer or network, a special entry can be in this file; for example, the following entry would state that all further processing should be done in a file called "C:\ANOTHER.INI":

```
[pro386w]
profile=c:\another.ini
```

The profile/1 implements this redirection, just once: it will not follow a chain of "profile=" entries between multiple initialisation files.

This is one of a series of file-oriented predicates which are implemented directly in terms of the Windows API. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.
**profile/4**

*get or set an initialisation file profile string*

```
profile( File, Section, Entry, String )
```

+File <atom>
+Section <atom> or <string>
+Entry <atom> or <string>
?String <variable>, <atom> or <string>

**Comments**

This reads or writes a String into the named initialisation File, using the given Section and Entry names. If any of the given File, Section or Entry does not exist, it is created automatically. If Entry is given as an empty atom or string, the named Section is deleted; similarly, if String is given as an empty atom or string, the named Entry is deleted. The Section and Entry must either be of the same data type (both may be atoms, or both may be strings): where String is an unbound variable, it will be returned using this data type; if String is given, then it must use the same data type as both Section and Entry.

**Examples**

The following call returns the current "wallpaper" setting in the "[desktop]" section of "win.ini":

```
?- profile( 'win.ini', `desktop`, `wallpaper`, S ).
S = `willow.bmp`
```

The next call picks up the name of the WIN-PROLOG initialisation file, and stores an entry, "bar=sux", in a section called "foo", in this file:

```
?- profile( File ), profile( File, foo, bar, sux ).
File = 'C:\PRO386W\PRO386W.INI'
```

**Notes**

Initialisation files are processed entirely by Windows itself, and profile/4 therefore makes no use of the WIN-PROLOG file input/output system. Where a given File name is specified without a path, it is assumed to refer to a file in the Windows directory, rather than in the current working directory.

This is one of a series of file-oriented predicates which are implemented directly in terms of the Windows API. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for `absolute_file_name/2` for further discussion about file names.
prolog_flag/2

get or check the value of a prolog flag

prolog_flag( Flag, Value )

?Flag <atom> or <variable>
?Value <atom> or <variable>

Comments
This predicate retrieves or checks the value of a "Prolog flag", a system setting that modifies the behaviour of Prolog. Where Flag is given, the corresponding value is retrieved and unified with Value; where Flag is an unbound variable, successive Prolog flags are tried on backtracking, and their names and values unified with Flag and Value respectively.

Examples
The following call returns the value of the "debugging" flag:

?- prolog_flag( debugging, V ).
V = debug

On backtracking, the following call returns the names of successive flags whose current value is "off":

?- prolog_flag( F, off ).
F = save_settings ;
F = print_log_file ;
F = status_box

Notes
Prolog flags are maintained by two predicates, prolog_flag/2 and prolog_flag/3: the former is used to get or check flag values, and can provide different solutions on backtracking, while the latter does not support backtracking, and is used to set a new value for a specific flag while returning its previous setting. A number of Prolog flags are also manipulated by a curious collection of other predicates, including debug/0, nodebug/0, trace/0 and so forth.

Each Prolog flag controls the behaviour of some aspect of the behaviour of WIN-PROLOG. For example, the "debug" flag defines how queries interact with the system debugger, while the "fileerrors" flag determines how open/2 and similar file handling predicates deal with nonexistent files. The names of all Prolog flags, their default values and alternative settings can be found in the entry for prolog_flag/3.
**prolog_flag/3**

set and get the value of a prolog flag

```prolog
prolog_flag( Flag, Old, Value )
```

+Flag <atom>
?Old <atom> or <variable>
+Value <atom>

**Comments**

This predicate is used to set the value of a "Prolog flag", a system setting that modifies the behaviour of Prolog. The existing value of the given Flag is retrieved, and an attempt is made to unify it with Old: if the match succeeds, and the given Value is a valid setting for the Flag, then the latter is set to the new Value accordingly. The following table shows the complete list of Prolog flags, together with their default settings:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>consult_errors</td>
<td>retry</td>
</tr>
<tr>
<td>consult_prompt</td>
<td>'</td>
</tr>
<tr>
<td>debugging</td>
<td>debug</td>
</tr>
<tr>
<td>debug_file</td>
<td>srcbug</td>
</tr>
<tr>
<td>discontiguous</td>
<td>off</td>
</tr>
<tr>
<td>display_unify</td>
<td>bindings</td>
</tr>
<tr>
<td>fileerrors</td>
<td>on</td>
</tr>
<tr>
<td>flex_extension</td>
<td>'.ksl'</td>
</tr>
<tr>
<td>gc</td>
<td>on</td>
</tr>
<tr>
<td>gc_collect</td>
<td>heap</td>
</tr>
<tr>
<td>hash_files</td>
<td>on</td>
</tr>
<tr>
<td>hash_limit</td>
<td>'100'</td>
</tr>
<tr>
<td>hash_quota</td>
<td>'100'</td>
</tr>
<tr>
<td>multiple</td>
<td>off</td>
</tr>
<tr>
<td>object_extension</td>
<td>'.pc'</td>
</tr>
<tr>
<td>overlay_extension</td>
<td>'.ovl'</td>
</tr>
<tr>
<td>Flag</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
</tr>
<tr>
<td>discontiguous</td>
<td>on, off</td>
</tr>
<tr>
<td>fileerrors</td>
<td>on, off</td>
</tr>
<tr>
<td>gc</td>
<td>on, off</td>
</tr>
<tr>
<td>hash_files</td>
<td>on, off</td>
</tr>
<tr>
<td>multiple</td>
<td>on, off</td>
</tr>
<tr>
<td>print_log_file</td>
<td>on, off</td>
</tr>
<tr>
<td>save_format</td>
<td>on, off</td>
</tr>
</tbody>
</table>

Several Prolog flags behave as toggles for system features, and may be set to one of two values, namely "on" or "off":

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>discontiguous</td>
<td>on, off</td>
</tr>
<tr>
<td>fileerrors</td>
<td>on, off</td>
</tr>
<tr>
<td>gc</td>
<td>on, off</td>
</tr>
<tr>
<td>hash_files</td>
<td>on, off</td>
</tr>
<tr>
<td>multiple</td>
<td>on, off</td>
</tr>
<tr>
<td>print_log_file</td>
<td>on, off</td>
</tr>
<tr>
<td>save_format</td>
<td>on, off</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>discontiguous</td>
<td>on, off</td>
</tr>
<tr>
<td>fileerrors</td>
<td>on, off</td>
</tr>
<tr>
<td>gc</td>
<td>on, off</td>
</tr>
<tr>
<td>hash_files</td>
<td>on, off</td>
</tr>
<tr>
<td>multiple</td>
<td>on, off</td>
</tr>
<tr>
<td>print_log_file</td>
<td>on, off</td>
</tr>
<tr>
<td>save_format</td>
<td>on, off</td>
</tr>
</tbody>
</table>
Another set of Prolog flags are used to store items like default file extensions and so forth, and may be set to any atom:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>consult_prompt</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>debug_file</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>flex_extension</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>object_extension</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>overlay_extension</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>project_extension</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>source_extension</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>supervisor_prompt</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>text_extension</td>
<td>&lt;atom&gt;</td>
</tr>
<tr>
<td>user_prompt</td>
<td>&lt;atom&gt;</td>
</tr>
</tbody>
</table>

There is a set of Prolog flags can each be set to a fixed range of descriptive settings:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>consult_errors</td>
<td>abort, retry</td>
</tr>
<tr>
<td>debugging</td>
<td>debug, off, trace</td>
</tr>
<tr>
<td>display_unify</td>
<td>bindings, instantiation, off</td>
</tr>
<tr>
<td>gc_collect</td>
<td>heap, text</td>
</tr>
<tr>
<td>print_message</td>
<td>error, help, informational, interrupt, modify, warning</td>
</tr>
<tr>
<td>print_stream</td>
<td>console, message_log</td>
</tr>
<tr>
<td>Flag</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>recompile</td>
<td>always, when_changed</td>
</tr>
<tr>
<td>resave</td>
<td>always, when_changed</td>
</tr>
<tr>
<td>syntax_errors</td>
<td>dec10, error, fail, quiet</td>
</tr>
<tr>
<td>unknown</td>
<td>error, fail</td>
</tr>
</tbody>
</table>

A final set of Prolog flags can each be set to an atom whose contents denotes an integer value, for example '100':

<table>
<thead>
<tr>
<th>Flag</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_limit</td>
<td>&lt;atom&gt; denoting &lt;int&gt;</td>
</tr>
<tr>
<td>hash_quota</td>
<td>&lt;atom&gt; denoting &lt;int&gt;</td>
</tr>
</tbody>
</table>

Examples

The following call sets the value of the "debugging" flag to "off", while at the same time returning its existing value (this could be saved to allow the original value to be restored later):

```
?- prolog_flag( debugging, V, off ).
# Debugging mode switched to off
V = debug
```

Notes

Prolog flags are maintained by two predicates, `prolog_flag/2` and `prolog_flag/3`: the former is used to get or check flag values, and can provide different solutions on backtracking, while the latter does not support backtracking, and is used to set a new value for a specific flag while returning its previous setting. A number of Prolog flags are also manipulated by a curious collection of other predicates, including `debug/0`, `nodebug/0`, `trace/0` and so forth.

Each Prolog flag controls the behaviour of some aspect of the behaviour of WIN-PROLOG. For example, the "debug" flag defines how queries interact with the system debugger, while the "fileerrors" flag determines how open/2 and similar file handling predicates deal with nonexistent files.
prolog_load_context/2

*get or check information about the current load context*

`prolog_load_context( Key, Value )`

?Key <atom> or <variable>
?Value <atom> or <variable>

**Comments**
This gets or checks information about a file currently in the process of being loaded, and can find successive values for `Key` and `Value` on backtracking. The names of the keys and their meanings are described in the table below:

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory</td>
<td>the directory containing the file being compiled</td>
</tr>
<tr>
<td>file</td>
<td>the full name of the file being compiled</td>
</tr>
<tr>
<td>module</td>
<td>the module into which code is being compiled</td>
</tr>
<tr>
<td>stream</td>
<td>the stream from which the file is being read</td>
</tr>
</tbody>
</table>

**Examples**
The `prolog_load_context/2` only succeeds when called from within a file being compiled, and simply fails from the command line. Suppose the following command has been saved in the file, "foo.pl", in the **WIN-PROLOG** home directory:

```prolog
:- prolog_load_context( Key, Value ),
   writeq( Key = Value ),
   nl,
   fail.
```

When this file is consulted, it will display the following output:

```prolog
?- consult( foo ).
module = user
file = 'c:\pro386w\foo.pl'
stream = 'c:\pro386w\foo.pl'
directory = 'c:\pro386w\'
# 0.000 seconds to consult foo.pl [c:\pro386w]\yes
```
Notes  This predicate is provided for Quintus Prolog compatibility: the load context cannot distinguish between WIN-PROLOG program windows and disk files, and modules are not supported.
prompt/2

get and set the Prolog initial read prompt

prompt( Old, New )

?Old <atom> or <variable>
+New <atom>

Comments This is used to change the prompt displayed when a program attempts to read input from the console (user) stream. The existing value of the prompt is retrieved, and an attempt is made to unify it with Old: if the match succeeds, the prompt is set to the New value accordingly.

Examples The following call sets the initial read prompt to "==> ", reads a term from the console, and then restores the original prompt:

?- prompt( P, '==> ' ), read( T ), prompt( _, P ).<enter>
==> new(prompt).<enter>
P = '|: ',
T = new(prompt)

Notes The prompt/2 predicate works by setting one of the Prolog flags, "user_prompt" (see prolog_flag/3), and is implemented as follows:

prompt( Old, New ) :-
    prolog_flag( user_prompt, Old, New ).

There are, in fact, two prompts in WIN-PROLOG: an initial prompt (which is the subject of this predicate), and a continuation prompt, which is used when an input predicate has began to read data, but requires more to complete its operation. The prompts/2 predicate provides access to both prompts.
prompts/2

get or set the Prolog initial and continuation read prompts

\[
\text{prompts( Init, Cont )}
\]

?Init \<string> or <variable>
?Cont \<string> or <variable>

Comments
This gets or sets the initial \(\text{(Init)}\) and continuation \(\text{(Cont)}\) prompts that are used when a program attempts to read input from the console (user) stream. Either or both prompts may be specified as a variable, in which case the current prompt is returned, or as a string, in which case this sets a new prompt.

Examples
The following call sets the initial read prompt to "ready: ", and the continuation prompt to "more.. ", and reads a term from the console. In this example, the term is entered over several lines in order to display both prompts:

\[
?- \text{prompts( `ready: `, `more.. ` ), read( T ).}
\]

Notes
Unlike \text{prompt/2}, the \text{prompts/2} does not modify the "user_prompt" Prolog flag (see \text{prolog_flag/3}): as such, its effects are fairly transient, lasting at most for the duration of a single query at the "?-" prompt. It does, however, provide a mechanism for specifying both the initial and continuation prompts, which can be useful in applications when reading from the console: the continuation prompt can contain an error message explaining that the input was not properly completed.
**put/1**

*evaluate and write a character to the current output stream*

```
put( Expression )
```

+Expression <expr>

**Comments**

This evaluates the given *Expression* by calling *is/2*, and writes the result to the current output stream, as if it were a character code; an error is generated if the *Expression* does not evaluate to an integer.

**Examples**

In most cases, this predicate is used to output characters which are already represented as character codes, for example:

```
?- put( 64 ), nl.
@ yes
```

The following call illustrates the fact that *put/1* performs arithmetic, and computes an expression before outputting the resulting integer as a character code:

```
?- put( "a" - 32 ), nl.
A yes
```

**Notes**

Although similarly named, the *put/1* predicate is not symmetrically opposed to *get/1*: the latter never performs arithmetic, only dealing with variables or integers; furthermore, *get/1* does not process non-printable characters (those with character codes of 20h or less).
**putb/1**

*put a character directly to the console (user) output device*

```
putb( Char )
```

+Char <char>

**Comments**

This outputs a single Char, represented as its integer character code, directly to the console window (user output stream).

**Examples**

In a simple call, the behaviour of this predicate seems similar to that of `put/1`:

```
?- put( 64 ), putb( 64 ), nl.
@@
```

```
yes
```

The main feature of `putb/0` is that it always outputs directly to the console, irrespective of the current setting of the output stream; the next examples use the output redirection predicate, `~>/2`, to write to a string, "S". A call to `putb/1` is surrounded by a pair of calls to `put/1`: notice that while the latter predicate's output ("A" and "C") goes to the string, the former's output ("B") goes directly to the console:

```
?- (put( 65 ), putb( 66 ), put( 67 )) ~> S, nl.
```

```
B
S = `AC`
```

**Notes**

This predicate is one of a special family of three which is principally aimed at helping the debugging of applications. Each of `getb/1`, `grab/1` and `putb/1` performs its input or output directly from or to the lowest-level of user I/O.

The `getb/1` will return immediately if a byte is stored in the keyboard or mouse type-ahead buffer; otherwise, it will suspend program execution until a key or mouse button is pressed. The related `grab/1` predicate also returns immediately any character stored in type-ahead, but simply fails if nothing is there: this allows programs to peek into type-ahead without risking being suspended if nothing is present. Finally, `putb/1` outputs the given byte directly to the console window, irrespective of whichever output stream is currently set.
**putx/2**

*write a word to the current output stream*

```
putx( Size, Word )
```

+Size   <integer> in the domain {-1,[1..4]} (X86) or {-1,[1..8]} (X64)
+Word   <integer> or <float>

**Comments**

This writes the next *Word* of the given *Size* to the current output stream. Where *Size* is in the range [1..4] (X86) or [1..8] (X64), *Word* must be an integer which is disassembled and output as the given number of bytes, assuming "little endian" encoding. If *Size* is minus one (-1), *Word* may be an integer or a float, and is written out in eight bytes as a floating point number.

**Examples**

The following call writes an integer to a string, in the form of a four-byte (32-bit) word:

```prolog
?- putx( 4, 256 ) ~> S.
S = `~@~A~@~@
```

**Notes**

The *putx/2* predicate is useful for writing binary data to files to be exported to other applications. It creates files that contain data in the "little endian" format, where the least significant byte of a long word is stored first, followed successively by each byte of greater significance. This model is employed by the Intel 80386 and all its successors, and is therefore inherent to Windows file and data structures. The related *putx/2* predicate can be used to write integers in little endian binary format.

Used with a *Size* of one (1), *putx/2* is similar to but considerably more efficient than *put/1*: the present predicate does not perform arithmetic evaluation each time it is called. The other two sizes most commonly of interest are two (2) and four (4), which relate to standard 16-bit and 32-bit words respectively: the size of three (3) is also supported, although 24-bit little endian integers are not normally used in data files in the Windows environment. The special case of minus one (-1) allows a number, whether an integer or a float, to be written as an IEEE-format 64-bit floating point number.
**read/1**

read and unify a term from the current input stream

```prolog
read( Term )
```

**Comments**
This reads an Edinburgh syntax term from the current input stream, and then attempts to unify it with `Term`, succeeding or failing accordingly.

**Examples**
The following command reads a term from the console:

```prolog
?- read( T ).
|: foo(VAR1,VAR2).
T = foo(_1,_2)
```

**Notes**
This predicate inputs terms that have been written in "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes ("), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes (’) (see `writeq/1`). Terms must be terminated by a period (.) and at least one space or control character.

If `read/1` encounters the end of a file while reading, it returns an atom, "end_of_file". If a syntax error is encountered during input, a call is made to `skip_term/0`, to consume the remainder of the damaged term. Further processing depends upon the setting of the "syntax_errors" Prolog flag (see `prolog_flag/3`), as described in the following table:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec10</td>
<td>print a warning message before trying to read another term (default)</td>
</tr>
<tr>
<td>error</td>
<td>abort reading and generate an error</td>
</tr>
<tr>
<td>fail</td>
<td>print a warning message and fail</td>
</tr>
<tr>
<td>quiet</td>
<td>fail without printing a warning message</td>
</tr>
</tbody>
</table>

Two related predicates provide further support for Edinburgh input, including the ability to read and manipulate variable names: see `eread/1` and `eread/2` for further information.
reconsult/1

reload source files using the incremental compiler

reconsult( FileSpec )

+FileSpec    <file_specs>

Comments
This predicate loads the programs contained in the disk file or files specified in FileSpec, using the incremental compiler. If the file has previously been loaded, the existing copy is abolished before reloading.

Examples
The following call loads the example program, "BENCHMRK.PL", into memory:

?- reconsult( examples(benchmrk) ).  <enter>
# 0.000 seconds to consult benchmrk.pl [c:\pro386w\examples\]
yes

To see that it was compiled with the incremental compiler, type the call:

?- predicate_property( suite, P ).  <enter>
P = compiled ;  <space>
P = static ;  <space>

no

Notes
This predicate is identical in every respect to consult/1: it is included simply for historical compatibility. In many early Prolog systems, "consulting" the same file more than once resulted in multiple copies of its predicates being loaded into memory, and reconsult/1 was introduced to overcome this limitation, allowing files to be reloaded from scratch as often as needed, for example to restore initial data values. Later Prolog systems have tended to redefined consult/1 so that it, too, removes a previously-loaded version of a program before loading the new image: thus consult/1 and reconsult/1 are now synonymous.

It is recommended to load programs with consult/1 or its optimised equivalent, compile/1, rather than with reconsult/1: the latter, historical relic is likely to be removed from the system at some point in the future.
remove/3

remove an element from a list

remove( Term, List, Rest )

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

Comments  This predicate succeeds when its first argument is bound to a Term that is an element in a particular List, and Rest is bound to another list that contains all the elements of List except for Term. Either of the last two arguments may be fully or partially instantiated lists, or simply variables; remove/3 can backtrack to generate alternative solutions where appropriate.

Examples  The following command simply removes the element, "2", from a list, "[1,2,3]", to give the remainder:

?- remove( 2, [1,2,3], R ).
R = [1,3] ;
no

The next example runs this predicate in reverse, inserting the given element into a list; on backtracking, each possible solution is offered in turn:

?- remove( a, L, [1,2] ).
L = [a,1,2] ;
L = [1,a,2] ;
L = [1,2,a] ;
no

Notes  The remove/3 predicate is a classic Prolog program, and is widely used for removing items from or inserting them into lists: however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of remove/3. In order to avoid errors, such references must be renamed or removed before loading files into WIN-PROLOG.
removeall/3

remove all matches of an element from a list

\[ \text{removeall( Term, List, Rest )} \]

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

Comments
This removes all elements from a List which can be individually unified with the given Term, and attempts to unify a list of the remaining elements to Rest; any variables in Term are left unbound at the end of execution.

Examples
The following command simply removes all elements which unify with "2" from a list, ",[1,2,3,2,4]", to give the remainder:

\[ ?- \text{removeall( 2, } [1,2,3,2,4], \text{ R ).} \]
\[ \text{R} = [1,3,4] \]

The next call removes all elements in a list that individually match the pattern, "(X,X)". Notice that two entries, "(3,3)" and "(6,6)" are removed, even though "3" and "6" are different numbers, because each element in the List is tested in turn, without binding variables in the Term; this is why "X" is not bound by the call:

\[ ?- \text{removeall( (X,X), } [(1,2),(3,3),(4,5),(6,6)], \text{ L ).} \]
\[ X = \_ , \]
\[ L = [(1,2),(4,5)] \]

Notes
The removeall/3 predicate is useful for the deterministic removal of single or multiple items from a list: it does not, however, provide a mechanism for knowing which terms were removed. A related predicate, remove/3, is a classic Prolog program that is widely used for removing items from or inserting them into lists one at a time, and may be more flexible in certain circumstances.

One use for removeall/3 is in duplicate removal programs, which recurse through a list, at each iteration removing all instances of the list "head" from its "tail". Unfortunately, such programs become inefficient as list sizes grow, exhibiting classic "n-squared" execution characteristics. Pre-sorting a list with sort/3 allows duplicate removal to be performed linearly, while sort/2 removes duplicates automatically.
ren/2

rename a file

\[ \text{ren}( \text{Old}, \text{New} ) \]

+Old <atom>
+New <atom>

Comments
This renames the file or directory with the given \textit{Old} name with the given \textit{New} name. If the original file does not exist, or if a file with the new name already exists, an error is generated.

Examples
Assuming that there is a file called "foo" in the current working directory, the following call renames it "bar":

\[ ?- \text{ren}( \text{foo}, \text{bar} ) . \]

\text{yes} <enter>

Notes
This is one of a series of file handling and operating system interface predicates which are implemented directly in the \texttt{WIN-PROLOG} "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for \texttt{absolute_file_name/2} for further discussion about file names.

Please note that \texttt{ren/2} attempts to rename files immediately, but cannot check whether the file concerned is currently open for input and/or output. Depending upon the version of Windows being run, it might be possible to rename a file which is still active: in such cases, disk space will need to be recovered subsequently by calls to the CHKDSK or SCANDISK utilities.
**repeat/0**

*succeed indefinitely on subsequent failure and backtracking*

repeat

**Comments**
This predicate succeeds when called, and leaves a choice point to a self-recursive call so that it will re-satisfy indefinitely as a result of subsequent failure and backtracking.

**Examples**
The following command repeatedly calls `get/1` until it reads the character code 67 ("C"), finally performing a cut (`!/0`) to remove the choice point introduced by `repeat/0`:

```prolog
?- repeat, get( 67 ), !.
| : A
| : B
| : C
yes
```

**Notes**
The `repeat/0` predicate provides Prolog with a very powerful control structure, effectively beginning a "repeat until" block. In conjunction with `fail/0`, it even provides "repeat forever": For example, many WIN-PROLOG stand-alone applications use the following "idle loop" to suspend active processing while waiting for messages (see `wait/1` for further information):

```prolog
wait_for_messages :-
    repeat,
    wait( 0 ),
    fail.
```

Because `repeat/0` always leaves a choicepoint, it is usually necessary to perform a cut (call to `!/0`) after the "until" test of a "repeat until" structure has succeeded. This predicate is implemented as a simple Prolog program, as follows:

```prolog
repeat.
repeat :-
    repeat.
```
repeat/1

_succeed a limited number times on subsequent failure and backtracking_

\[ \text{repeat}(\text{Count}) \]

+Count <integer> in the range \([1..2147483647]\)

Comments This predicate succeeds when called, and leaves a choice point that will re-satisfy one less than Count times as a result of subsequent failure and backtracking.

Examples The following command calls \(\text{put}/1\) until the given count ("5") has expired, before outputting a new line:

\[ ?- \text{repeat}(5), \text{put}(64), \text{fail} ; \text{nl}. \]

An equivalent, but "cleaner" version of the above can be achieved with the help of the \(\text{forall}/2\) predicate:

\[ ?- \text{forall}(\text{repeat}(5), \text{put}(64)), \text{nl}. \]

Notes The \(\text{repeat}/1\) predicate provides Prolog with a simple control structure, effectively beginning a count-limited "repeat until" block. Another predicate, \(\text{repeat}/2\), takes this a stage further, by returning successive integers at each iteration:

\[ ?- \text{forall}(\text{repeat}(5,X), \text{write}(X)), \text{nl}. \]

Please note that \(\text{repeat}/2\) leaves a choice point one less than Count times, as shown in this example which uses \(\text{call}/2\):

\[ ?- \text{call}(\text{repeat}(2), P). \]

P = true ;

P = !
repeat/2

succeed a limited number times on subsequent failure and backtracking, returning iteration count

\[ \text{repeat( Count, Index )} \]

+Count <integer> in the range [1..2147483647]
-Index <variable>

Comments
This predicate succeeds when called, and leaves a choice point that will re-satisfy one less than Count times as a result of subsequent failure and backtracking, each time returning the current Index.

Examples
The following command calls write/1 until the given count ("5") has expired, before outputting a new line:

?- repeat( 5, I ), write( I ), write( ` ` ), fail ; nl.
       1 2 3 4 5
I = _

Notes
The \text{repeat/2} predicate provides Prolog with a simple control structure with automatic counting from 1 to a given number, effectively beginning a count-limited "repeat until" block. Another predicate, \text{repeat/3}, takes this a stage further still, by allowing you to specify arbitrary start and end points for the loop:

?- forall( repeat(0'A,0'E,X), put(X) ), nl.
       ABCDE
X = _

Please note that \text{repeat/3} leaves a choice point one less than Count times, as shown in this example which uses \text{call/2}:

?- call( repeat(2,I), P ).
       I = 1 ,
       P = true ;
       I = 2 ,
       P = !
repeat/3

succeed a for a range of integers on subsequent failure and backtracking, returning current integer

repeat( Start, Final, Index )

+Start <integer> in the range [1..2147483647]
+Final <integer> in the range [Start..2147483647]
-Index <variable>

Comments This predicate succeeds when called, and leaves a choice point that will re-satisfy one less than Count times as a result of subsequent failure and backtracking.

Examples The following command calls put/1 for all integers between "65" and "90", before outputting a new line:

?- repeat( 65, 90, I ), put( I ), fail ; nl.
ABCDEFGHIJKLMNOPQRSTUVWXYZ
I = _

Notes The repeat/3 predicate provides Prolog with a simple control structure with automatic counting from between any two given numbers, and replaces the non-standard and quirkily named predicate, integer_bound/3, found in versions of many WIN-PROLOG up to 5.000. The simpler repeat/2 predicate performs a similar process, but always counts from 1 to the given number. Effectively, the latter predicate can be specified in terms of the former as follows:

repeat( Count, Index ) :-
    repeat( 1, Count, Index ).

Please note that repeat/3 leaves a choice point one less than Count times, as shown in this example which uses call/2:

?- call( repeat(8,9,I), P ).
I = 8 ,
P = true ;
I = 9 ,
P = !
retract/1

*delete the first matching clause from a dynamic predicate*

retract( Clause )

+Clause <clause>

**Comments**

This deletes the first matching Clause from a dynamic predicate. The given Clause can be partially or fully instantiated, although as a minimum its head must include the predicate name, and any arguments must be fully enumerated: attempts are made to unify Clause with successive program clauses, and if a match is found, the clause in question is deleted. On backtracking, attempts are made to find and delete further matching clauses. When no (more) matching clauses are found, the call fails; if the target predicate did not previously exist, it is declared "dynamic" and again the call fails.

**Examples**

Consider the following program, which describes various cases of "who" likes "what":

```
:- dynamic likes/2.
likes( pooh, hunny )).
likes( brian, wine ).
likes( brian, X ) :-
    likes( pooh, X ).
```

Once compiled, this program can be called as follows:

```
?- likes( Who, What ).
Who = pooh ,
What = hunny ;

Who = brian ,
What = wine ;

Who = brian ,
What = hunny
```
Now we'll delete any clauses of the form, "likes(brian,W):-G", returning the values of "W" and "G" in the process; once we have done this, we'll test the predicate again:

```prolog
?- retract( likes(brian,W):-G ) .
```

```
W = wine ,
G = true ;
```

```
W = _ ,
G = likes(pooh,W) ;
```

```
no
```

```prolog
?- likes( Who, What ).
```

```
Who = pooh ,
What = hunny
```

**Notes**

When deleting a rule, it is necessary to parenthesise it, as shown in the second example above, in order to avoid a syntax error. This is because the ".-" operator has a higher precedence (1200) than a normal argument list (1000). Notice that facts can also be deleted when a rule is specified: a dummy body goal of "true" is automatically added, again as shown above.

Predicates can only be retracted from when they are declared "dynamic": if retract/1 is called to delete from a previously nonexistent predicate, this declaration is performed automatically before the call fails. However, if some clauses for predicate have already been loaded from a file, the predicate may be considered "static", and any attempt to retract from it will result in errors. If a source file contains the definition of a predicate which is to be further modified at run-time, this predicate should be declared dynamic explicitly with a call to dynamic/1; for example, consider the following file:

```prolog
:- dynamic foo/1.
foo( hello ).
bar( world ).
```

If this file were to be loaded with a call to consult/1, it would be possible to retract clauses from the foo/1 predicate, but not from bar/1.

It is never possible to retract from optimised predicates (those which have been compiled with the optimising compiler); were the above file to be optimised, for example with a call to optimize_files/1, the foo/1 predicate would be bypassed by the optimiser, leaving it dynamic.
Note that retracting a clause actually invokes a very efficient incremental decompiler, which converts low-level Prolog virtual machine instructions back into a representation of the clause. This means that even dynamically modified code is able to run at compiled code speeds.

Extreme care should be taken when deleting or adding to predicates: in order to maintain efficiency and maximise the utilisation of memory, WIN-PROLOG immediately removes the code following a deletion, even if that code is still being executed. Similarly, when clauses are added to a program, they are inserted into memory immediately.

Executing a program which suddenly gets overwritten with garbage or new code can produce unexpected results. In general, therefore, it is not a good idea to write programs which call a routine, and then modify it while it is still running or contains untried choice points.

For safety, WIN-PROLOG contains a feature that allows programs to be treated as "logical dynamic" code, which effectively allows Prolog to call a safely-frozen copy of a routine, rather than the routine itself. In this case, any amount of mutilation can be applied to the program being called: it is a copy which is actually running. See the entry for dynamic/1, as well as Appendix B, to find out more about this feature.
retract/2

delete the first matching clause at a known position

retract( Clause, Position )

+Clause <clause>
?Position <integer> or <variable>

Comments
This deletes the first matching Clause from a dynamic predicate, returning its Position, or deletes a specific Clause at a given Position. The given Clause can be partially or fully instantiated, although as a minimum its head must include the predicate name, and any arguments must be fully enumerated. Where Position is an unbound variable, attempts are made to unify Clause with successive program clauses, and if a match is found, the clause in question is deleted and its clause number is bound to Position. If Position is given, only the indicated Clause is matched and, if successful, deleted. In either case, on backtracking, attempts are made to find and delete further matching clauses. When no (more) matching clauses are found, the call fails; if the target predicate did not previously exist, it is declared "dynamic" and again the call fails.

Examples
Consider the following program, which describes various cases of "who" likes "what":

:- dynamic likes/2.
likes( pooh, hunny ).
likes( brian, wine ).
likes( brian, X ) :-
    likes( pooh, X ).

Once compiled, this program can be called as follows:

?- likes( Who, What ).
Who = pooh ,
What = hunny ;

Who = brian ,
What = wine ;

<enter>
<space>
<space>
Who = brian ,
What = hunny

The following command deletes the second clause from this program, and uses a cut (call to !/0) to commit to this solution; once we have done this, we'll test the predicate again:

?- retract( (likes(X,W):-G), 2 ), !.
   X = brian ,
   W = wine ,
   G = true

?- likes( Who, What ).
   Who = pooh ,
   What = hunny ;

   Who = brian ,
   What = hunny

Now we'll delete any clauses of the form, "likes(brian,W):-G", returning the values of "W" and "G" together with the clauses's position, before testing the predicate a final time:

?- retract( (likes(brian,W):-G), P )..
   W = _ ,
   G = likes(pooh,W) ,
   P = 2 ;

   no

?- likes( Who, What ).
   Who = pooh ,
   What = hunny

Notes
When Position is an unbound variable, retract/2 behaves very much like retract/1; however, the former is less efficient than the latter at backtracking through large relations to delete successive matching clauses. Because it works by clause number, rather than address, retract/2 exhibits "n-squared" behaviour with respect to relation size: retract/1, on the other hand, always behaves linearly. Where Position
is given at the time of call, it is still possible to delete further clauses on backtracking: if, say, the second clause of a relation is matched and deleted, what was previously the third clause now becomes the second, and is therefore eligible for deletion on backtracking, so long as it also matches the original Clause.

Please see the "notes" section in the entry for retract/1 for important additional information about adding clauses to and deleting them from dynamic predicates.
retractall/1

delete all clauses matching the given head

retractall( Head )

+Head       <goal>

Comments  This deletes all clauses, whose heads match the given Head, from a dynamic predicate; no variables are bound during this call. The given Head can be partially or fully instantiated, although as a minimum it must include the predicate name, and any arguments must be fully enumerated. Even where no matching clauses are found, the call succeeds; if the target predicate did not previously exist, it is declared "dynamic" and again the call succeeds.

Examples  Consider the following program, which describes various cases of "who" likes "what":

:- dynamic likes/2.
likes( pooh, hunny ).
likes( brian, wine ).
likes( brian, X ) :-
    likes( pooh, X ).

Once compiled, this program can be called as follows:

?- likes( Who, What ).  <enter>
Who = pooh ,
What = hunny ;  <space>
Who = brian ,
What = wine ;  <space>
Who = brian ,
What = hunny
Now we'll delete all clauses whose heads are of the form, "likes(brian,W)" (note that "W" is not bound in this call); once we have done this, we'll test the predicate again:

```prolog
?- retractall( likes(brian,W) ).
W = _

?- likes( Who, What ).
Who = pooh ,
What = hunny
```

**Notes**

The `retractall/1` predicate is very efficient, exhibiting linear behaviour with respect to the size of the relation from which it is deleting clauses, and is deterministic, meaning that it does not create choicepoints; it provides a mechanism of filtering unwanted clauses out of a dynamic relation. Unless an error occurs by trying to delete clauses from a static relation, this predicate always succeeds. Effectively, `retractall/1` is implemented in terms of `retract/1`, as follows:

```prolog
retractall( Head ) :-
    retract( (Head:-Body) ),
    fail.

retractall( Head ).
```

The related predicates, `abolish/1` and `dynamic/1`, provide even quicker methods of deleting an entire relation; each can delete static predicates as well as dynamic ones, but the latter subsequently declares its target predicate(s) dynamic.

Please see the "notes" section in the entry for `retract/1` for important additional information about adding clauses to and deleting them from dynamic predicates.
reverse/2

reverse the order of elements in a list

\[
\text{reverse}( \text{List}, \text{Reverse} )
\]

?List <list> or <variable>
?Reverse <list> or <variable>

Comments
This predicate succeeds when both its first argument is bound to a List, and its second argument contains a list comprised of the same set of elements, but in Reverse order. Either of the two arguments may be fully or partially instantiated lists, or simply variables; reverse/2 can backtrack to generate alternative solutions where appropriate.

Examples
The following command simply reverses the list, 
\([1,2,3]\)
, to give a second list:

\[
?- \text{reverse}( [1,2,3], R ).
\]

\[
R = [3,2,1]
\]

Working "backwards", the following command also generates a solution, but leaves a choicepoint: on backtracking, an infinite search for a second solution leads to a "heap full" error:

\[
?- \text{reverse}( R, [1,2,3] ).
\]

\[
R = [3,2,1] ;
\]

Error 4, Heap Space Full, Trying ewrite/1

Notes
The reverse/2 predicate is a classic Prolog program, and is widely used for reversing the order of elements in lists; however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of reverse/2. In order to avoid errors, such references must be renamed or removed before loading files into WIN-PROLOG.

When its first argument is given as a list, reverse/2 is very efficient (its behaviour is linear with respect to list length), because it uses a "difference list" algorithm; however, where the first argument is a variable, the algorithm reverts to "generate and test", which explains the eventual "heap full" error on backtracking. For this reason, it is always best to specify the first argument in calls to reverse/2.
**reverse/3**

*reverse the order of elements in a list onto another*

\[
\text{reverse( List, Tail, Reverse )}
\]

- **List**: <list> or <variable>
- **Tail**: <term> or <variable>
- **Reverse**: <list> or <variable>

**Comments**
This predicate succeeds when both its first argument is bound to a *List*, and its third argument contains a list comprised of the same set of elements, but in *Reverse* order, followed by the given *Tail*. Either of the first and last arguments may be fully or partially instantiated lists, or simply variables, while the middle argument may be any term; *reverse/3* can backtrack to generate alternative solutions where appropriate.

**Examples**
The following command simply reverses the list, "[1,2,3]", to give a second list:

\[
?\text{- reverse( [1,2,3], [a,b,c], R ).}
\]

\[
R = [3,2,1,a,b,c]
\]

**Notes**
The *reverse/3* predicate is a classic Prolog program, and is widely used for reversing the order of elements in lists: however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of *reverse/3*. In order to avoid errors, such references must be renamed or removed before loading files into WIN-PROLOG.

This predicate uses a "difference list" algorithm to perform efficient reversing without the user of *append/3*: as a result, its execution time is linear with respect to the length of list being processed. In WIN-PROLOG, it is used to implement the more common *reverse/2* predicate:

\[
\text{reverse( List, Reverse ) :-}
\]

\[
\text{reverse( List, [], Reverse ).}
\]

Note that this implementation works best when at least the first argument is specified (see *reverse/2* for more details).
display the "choose colour" dialog box

\[ \text{rgbbox}( \text{Title}, \text{Initial}, \text{Default}, \text{Final} ) \]

+Title <term>
+Initial <integer> or (<integer>,<integer>)
+Default <integer> or (<integer>,<integer>)
-Final <variable> or <list>

Comments
This displays a standard system "font box" (or "choose colour") dialog box, with the given Title, and Initial and Default colours or colour pairs, allowing the user to view and select a new colour or colour pair. The dialog closes when the "OK" button is clicked or if <enter> is pressed, and the user's chosen colour or colour pair is unified with Final. If Initial and Default specified as integers, the dialog displays a single RGB colour; if Initial and Default specified as pairs of integers, the dialog expands to show a foreground/background pair of RGB values. The integers used for Initial should be as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>commence with the appropriate system default</td>
</tr>
<tr>
<td>0..16'00ffffff</td>
<td>commence with the given RGB colour value</td>
</tr>
</tbody>
</table>

The integers used for Default should be as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>disable the &quot;default&quot; button</td>
</tr>
<tr>
<td>0..24</td>
<td>use the given system default colour if requested</td>
</tr>
</tbody>
</table>

Examples
The following command displays the colour box, using the title, "Choose a Colour", and initialising it with bright blue (16'ff0000), and the "color_btnface" system default (15); after the user has adjusted the controls and selected a colour, the final choice is returned (in this example, bright red (255)):

```
?- rgbbox( `Choose a Colour`, 16'ff0000, 15, RGB ).
```

```
RGB = 255
```

The next command displays the colour box, using the title, "Choose a Pair", and initialising it with the system default foreground and back-
ground of "color_windowtext" (8) and "color_window" (5) respectively; after the user has adjusted the controls and selected a colours, the
final choice is returned (in this example, a black foreground (0) and bright red background (255)):

?- rgbbox('Choose a Pair', (-1,-1), (8,5), RGB ).  <enter>
RGB = (0,255)

**Notes**

The rgbbox/3 predicate provides a convenient method for allowing users to chose a colour or colour pair for use in an application; it does
not permanently create the chosen colour or colour pair in any sense, but simply returns its specification as one or a pair of integers
representing RGB values which may then be decoded and passed to graphics predicates such as gfx_brush_create/5 and others. Unlike
certain other programming environment dialogs, such as those accessed through prnbox/4, opnbox/5 and savbox/5, the present predicate
is not a true "common dialog", being implemented directly in terms of WIN-PROLOG's window handling predicates.
rich_format/3

get or set text format in a rich edit control

rich_format( Window, Range, Format )

+Window <window_handle>
+Range <atom>
?Format <variable> or <list>

Comments
This obtains or applies a Format from or to a Range of text in the given rich edit control Window. When Format is a variable, a list of all common formats for the specified text Range is returned as a list, when Format is a list, those entries that are specified are applied to the given text Range, as follows:

<table>
<thead>
<tr>
<th>Range</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>set the format of all text</td>
</tr>
<tr>
<td>default</td>
<td>get or set the default text format</td>
</tr>
<tr>
<td>selection</td>
<td>get or set format of selected text</td>
</tr>
<tr>
<td>word</td>
<td>set format to word boundary</td>
</tr>
</tbody>
</table>

The Format is specified as a list of "=" pairs, containing words and values, including any combination of the following:

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>allcaps</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>animation</td>
<td>&lt;integer&gt; in the range [0..255]</td>
</tr>
<tr>
<td>backlight</td>
<td>RGB colour object (see below)</td>
</tr>
<tr>
<td>bold</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>charset</td>
<td>&lt;integer&gt; in the range [0..255]</td>
</tr>
<tr>
<td>color</td>
<td>RGB colour object (see below)</td>
</tr>
<tr>
<td>disabled</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>emboss</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>face</td>
<td>&lt;string&gt; of up to 32 characters</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>hidden</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>imprint</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>italic</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>kerning</td>
<td>&lt;integer&gt; in the range [0..65535]</td>
</tr>
<tr>
<td>lcid</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>link</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>offset</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>outline</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>protected</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>revauthor</td>
<td>&lt;integer&gt; in the range [0..255]</td>
</tr>
<tr>
<td>revised</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>shadow</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>size</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>smallcaps</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>spacing</td>
<td>&lt;integer&gt; in the range [0..65535]</td>
</tr>
<tr>
<td>strikeout</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>style</td>
<td>&lt;integer&gt; in the range [0..65535]</td>
</tr>
<tr>
<td>underline</td>
<td>&quot;0&quot; (off) or &quot;1&quot; (on)</td>
</tr>
<tr>
<td>underlinetype</td>
<td>&lt;integer&gt; in the range [0..255]</td>
</tr>
<tr>
<td>weight</td>
<td>&lt;integer&gt; in the range [0..65535]</td>
</tr>
</tbody>
</table>

Two of the items above use an "RGB Colour Object", which comprises a conjunction of three integers, each in the range [0..255], corresponding to the Red, Green and Blue channels respectively. For example, the format setting describes a purplish blue colour, of 50% red, 0% green and 100% blue:

```
color=(128,0,255)
```

**Examples**

The following call returns a complete list of the default format settings for the console window, ")(1,1)":

```
?- rich_format( (1,1), default, List ).
List = [bold = 0,italic = 0,underline = 0 ... revauthor = 0]
```
The next call will the change "face" and "size" of all text in the console window into "Arial 10pt":

?- rich_format( (1,1), all, [face='arial',size=200] ). <enter>
yes

Notes

The rich_format/3 predicate provides total control over the appearance of text in rich edit control windows, using the special window messages, "es_getcharformat" and "es_setcharformat". While most aspects of its behaviour are self-evident, it is worth noting that "size" values are quoted in "twips", or twentieths of a point: hence the use of the value "200" to obtain 10pt text in the above example.

Although the font predicates, such as wfcreate/4 and wfont/1, more or less work with rich edit controls, far greater control can be achieved by using the current predicate, rich_format/3.
rich_print/5

print the contents of a rich edit control

rich_print( Window, Title, Footer, Font, Margin )

+Window    <window_handle>
+Title      <string>
+Footer     <string>
+Font       <atom>, <integer> or <tuple>
+Margin     <integer> or <float>

Comments  This prints the contents of the given rich edit control Window, using the given Footer in the given Font, and using the page Margin, specified in inches. The given Title is used in the print progress dialog.

Examples   The following call creates a Font, "foo", that will be used to display the Footer on each page of printout:

   ?- wfcreate( foo, arial, 20, 0 ).          <enter>
   yes

   The next command prints out the contents of the console window, "(1,1)", using the dialog title, "Console", and the footer, "Footer" in the font, "foo", and a margin 0.5 inches:

   ?- rich_print( (1,1), `Console`, `Footer`, foo, 0.5 ).  <enter>
   yes

Notes      The present predicate prints the contents of any "Rich" window, whether it belongs to the console window, a "Text" window, or has been created as a control in a "Dialog" window. The text is printed using the same fonts, layout and colours as the window itself, and so provides a useful extension to the "Rich Syntax Colouring" of WIN-PROLOG, providing a means to generate coloured program listings (see rich_syntax/1).

   Apart from being able to change the wording and font used for the page footer, and also the page margin, any remaining printer or layout control is left to the print driver itself. A 3-digit page number is automatically printed to the left of the specified footer, again using the footer's font.
**rich_syntax/1**

*get or set the rich syntax colouring status*

rich_syntax( Status )

?Status <variable> or <integer>

**Comments**

This gets or sets the rich syntax Status, which controls the automatic "rich syntax colouring" of program windows, the console, and other features. When specified as a variable, Status is bound to an integer giving the current setting; when specified as an integer, it defines the new Status, causing windows to be re-coloured if necessary, as follows:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>turn off rich syntax colouring</td>
</tr>
<tr>
<td>1..167fffffff</td>
<td>turn on rich syntax colouring, with a delay of the given number of milliseconds in program windows</td>
</tr>
</tbody>
</table>

**Examples**

The following call returns the current rich syntax status:

?- rich_syntax( Status ).

Status = 250

The next call turns off rich syntax colouring, and redraws all program windows in their default colours:

?- rich_syntax( 0 ).

yes

**Notes**

"Rich Syntax Colouring" is a powerful feature of WIN-PROLOG, providing immediate feedback about the program being developed. Unfortunately, it also imposes a considerable processor overhead, and so it might be desirable to disable it at times, or arrange for its onset to be delayed for a given period of inactivity, to tailor keyboard response on any given platform.

The rich_syntax/1 predicate provides the required control, and in general, a setting of about "250" is recommended, corresponding to a 0.25 second delay before rich syntax colouring commences in program windows. This delay does not affect the console or debugger windows. Independently of this setting, the rich_syntax/2 predicate can be used to perform immediate rich syntax colouring on a given window.
rich_syntax/2

perform rich syntax colouring on a given window

    rich_syntax( Window, Mode )

+Window <window_handle>
+Mode <integer> in the range [0..3]

Comments  This performs "rich syntax colouring" on the given rich edit control Window, controlled by the given Mode, as follows:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>remove rich syntax colouring from the window</td>
</tr>
<tr>
<td>1</td>
<td>perform rich syntax colouring on the visible portion of the window only</td>
</tr>
<tr>
<td>2</td>
<td>perform rich syntax colouring on the entire window, without a progress box</td>
</tr>
<tr>
<td>3</td>
<td>perform rich syntax colouring on the entire window, with a progress box</td>
</tr>
</tbody>
</table>

Examples  The following call performs rich syntax colouring on the entire contents of a "text" window, "foo", displaying a "progress box" that can be used to interrupt processing if desired:

    ?- rich_syntax( (foo,1), 3 ).
    yes

Notes  "Rich Syntax Colouring" is a powerful feature of WIN-PROLOG, providing immediate feedback about the program being developed. Unfortunately, it also imposes a considerable processor overhead, and so it might be desirable to disable it at times, or arrange for its onset to be delayed for a given period of inactivity, to tailor keyboard response on any given platform.

The rich_syntax/2 predicate provides a mechanism to perform elective rich syntax colouring, independently of the automatic rich syntax status set by the rich_syntax/1 predicate. Its Mode parameter allows the colouring to be tailored to the application, with or without a "progress box" (see prgbox/1 and prgbox/3) which can provide feedback, and even allow the user to abort the process.
rmdir/1

delete a file directory

\texttt{rmdir( Directory )}

+Directory <atom>

Comments
This removes (deletes) the named, empty \texttt{Directory} from the disk or network. If the directory does not exist, or if it is not empty, an error is generated.

Examples
Assuming that there is an empty subdirectory called "foo" in the current working directory, the following call removes it:

\texttt{?- rmdir( foo ).}\hspace{1cm}<\texttt{enter>}

yes

Notes
This is one of a series of file handling and operating system interface predicates which are implemented directly in the \texttt{WIN-PROLOG "kernel"}, a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for \texttt{absolute_file_name/2} for further discussion about file names.

New directories can be created with \texttt{mkdir/1}, and the current working directory can be changed or queried with \texttt{chdir/1}; curiously, the ordinary file renaming predicate, \texttt{ren/2}, can be used to rename directories.
compute a reverse polish notation integer expression

\[
\text{rpn}( \text{Expression}, \text{Result} )
\]

\[
\begin{align*}
+ & \quad \text{Expression} \quad \langle \text{list} \rangle \\
? & \quad \text{Result} \quad \langle \text{integer} \rangle \text{ or } \langle \text{variable} \rangle \\
\end{align*}
\]

Comments

This evaluates the given reverse polish notation (RPN) arithmetic \( \text{Expression} \) and unifies the solution with \( \text{Result} \). The \( \text{Expression} \) comprises a list of integers and functions, which are processed from left to right in order to compute the result. The \( \text{Expression} \) list may contain anything from a single integer to large sequence of integers and functions.

The functions can be split into two groups: binary and unary functions. Binary functions are simply those that take two arguments, while unary functions take just one.

The following table shows the integer binary functions:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X, Y, + )</td>
<td>adds ( X ) to ( Y )</td>
</tr>
<tr>
<td>( X, Y, - )</td>
<td>subtracts ( Y ) from ( X )</td>
</tr>
<tr>
<td>( X, Y, * )</td>
<td>multiplies ( X ) by ( Y ), returns product</td>
</tr>
<tr>
<td>( X, Y, / )</td>
<td>divides ( X ) by ( Y ), returns dividend</td>
</tr>
<tr>
<td>( X, Y, # )</td>
<td>multiplies ( X ) by ( Y ), returns overflow</td>
</tr>
<tr>
<td>( X, Y, \backslash )</td>
<td>divides ( X ) by ( Y ), returns remainder</td>
</tr>
<tr>
<td>( X, Y, a )</td>
<td>computes the bitwise &quot;and&quot; ( X ) and ( Y )</td>
</tr>
<tr>
<td>( X, Y, l )</td>
<td>computes the left rotation of the ( X ) by ( Y ) bits</td>
</tr>
<tr>
<td>( X, Y, o )</td>
<td>computes the bitwise &quot;inclusive or&quot; ( X ) and ( Y )</td>
</tr>
<tr>
<td>( X, Y, r )</td>
<td>computes the right rotation of the ( X ) by ( Y ) bits</td>
</tr>
<tr>
<td>( X, Y, x )</td>
<td>computes the bitwise &quot;exclusive or&quot; ( X ) and ( Y )</td>
</tr>
</tbody>
</table>

The following table shows the integer unary functions:
<table>
<thead>
<tr>
<th>Sublist</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, ?</td>
<td>computes a linear congruential pseudo random integer between zero and X</td>
</tr>
<tr>
<td>X, @</td>
<td>computes a Marsaglia Zaman pseudo random integer between zero and X</td>
</tr>
<tr>
<td>X, $</td>
<td>sign-extends a 32-bit integer to 64-bit</td>
</tr>
<tr>
<td>X, &amp;</td>
<td>truncates a 64-bit integer to 32-bit</td>
</tr>
</tbody>
</table>

### Examples

The following calls show a variety of expressions being evaluated or tested:

```
?- rpn( [2,2,+], X ).
X = 4

?- rpn( [10,2,/] , 5 ).
yes

?- rpn( [22,7,/] , X ).
X = 3

?- rpn( [3.14,2,+], X ).
Error 23 : Type Error
```

### Notes

The `rpn/2` predicate is the low-level integer "maths engine" in WIN-PROLOG, and is used in a lot of internal routines for simple counting operations where maximum efficiency is desired. It is also used for bitwise flag manipulations throughout the development environment and many built-in predicates. Together with its floating-point sibling, `fpn/2`, this predicate remained undocumented until the release of the X64 version of WIN-PROLOG, at which point to of its functions, "$" and "&", began to play a lead role in writing portable Windows API code that would run on both the 32-bit (X86) and 64-bit (X64) platforms. See Appendix U for further discussion of these functions in relation to code portability.

The reverse polish notation model is very simple to understand, and entirely unambiguous. The given list is processed from left to right, and if a number is found, it is pushed onto an internal stack; if an atom is found, it is executed as a function. A binary function will attempt to pop two arguments from the internal stack, operate on them, and then push the result back on the stack. A unary function will pop just one value off the stack, process it, and then push the result back on the stack. The computation succeeds when the full list has been processed, and exactly one result is left on the internal stack: this result is unified with the predicate's second argument.

Conventional "infix" maths expressions have two problems, one human, the other computational. Ask someone to calculate the following expression in their heads, or even with the aid of a basic calculator, and many a time you will be given the wrong answer:
\[2 + 3 * 4 = ???\]

A lot of people will give you the wrong answer, "20", because first they will compute, "2 + 3 = 5", and then "5 * 4 = 20". The correct answer is, of course, "14", because the in arithmetic, you perform multiplications before additions: "3 * 4 = 12, 12 + 2 = 14. Just as you get with Prolog:

\[?- X \text{ is } 2 + 3 * 4\]
\[X = 14\]

The is/2 predicate itself has no knowledge of the rules of arithmetic: the above works only because "+" has been declared as an operator with a higher numerical precedence than "*" (500 yfx vs 400 yfx). If you were to redefine "+" as a lower precedence, with a call to op/3, then is/2 will appear to return the same, wrong result, as many ordinary folks do:

\[?- \text{op}(400, \text{yfx}, +)\].
\[yes\]

\[?- X \text{ is } 2 + 3 * 4\]
\[X = 20\]

Again, it should be stressed that this is not a bug in is/2: it is simply processing a Prolog term that has been created by read/1; before the call to op/3, "2+3*4" is read as the term, \((2+(3*4))\); after the call, it is read as \((2+3)*4\). The beauty of reverse polish notation, is that is not only unambiguous, but it cannot be affected by operator definitions:

\[?- \text{rpn}([2,3,4,*,+], X)\].
\[X = 14\]

\[?- \text{rpn}([2,3,+,4,*], X)\].
\[X = 20\]

There are two reverse polish predicates in WIN-PROLOG: fpn/2 (for all numbers) and rpn/2 (for integers only).
**safe/1**

*get or set the Windows shutdown flag*

```prolog
safe( Flag )
```

?Flag <variable> or <integer> in the domain {0,1}

**Comments**

This gets or sets the state of the Windows shutdown `Flag`. When clear, this flag permits a Windows shutdown sequence to proceed unhindered; when set, it causes shutdown to terminate, and a "msg_close" message to be sent to the "main window" ("0"). The flag may have either of the following states:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shutdown permitted, no warning received</td>
</tr>
<tr>
<td>1</td>
<td>shutdown not permitted, warning received</td>
</tr>
</tbody>
</table>

**Examples**

The following call picks the current value of the Windows "shutdown flag", then clears the flag (sets it to "0"), before calling some program "foo/0". Upon completion, it resets the flag to its original value:

```prolog
?- safe( F ), safe( 0 ), foo, safe( F ).
F = 1
```

**Notes**

Initially, the Windows shutdown flag is clear (has the value "0"): in this state it allows any Windows shutdown sequence to continue unhindered, without notification. If it is desired to intercept a Windows shutdown sequence, the flag should be set to "1": in this state, it prevents the continuation of shutdown, and notifies the "main window" ("0") of the attempted shutdown with a special "msg_close" message. An application can use this signal to perform any necessary housekeeping, before calling `exit/1` to resume the shutdown if desired.

The development environment uses `safe/1` to ensure that unsaved program windows are not lost in the event of a Windows shutdown, setting this flag to "1" whenever a change is made in a "program window", and clearing it to "0" whenever all such windows have been saved or closed.
display the "save as" common dialog box

savbox( Title, Filters, Name, Extension, Files )
+Title <term>
+Filters <list_of(<term>,<term>)>
+Name <term>
+Extension <term>
?Files <list> or <variable>

Comments
This displays a standard Windows "save box" ("save as") common dialog box, with the given Title, list of Filters and default Name and Extension, and then allows the user to browse for or enter the name of a single file. The Filters comprise a list of comma pairs of terms, normally strings or atoms, which describe file types and their extensions. The dialog closes when the "OK" button is clicked, if a file is double clicked, or if <enter> is pressed while the dialog's edit window has focus: at this point, a list of all selected or named files is unified with Files.

Examples
The following program shows a typical call to savbox/5:

foo( File ) :-
    savbox( `Save As...`,
        [ (`Source`,`*.pl`),
          (`Object`,`*.pc`),
          (`Prolog`,`*.pl *.pc`)
        ],
        `default`,
        `pl`,
        [File]
    ).

Once compiled, a call to this program will display the "save as" common dialog box, with the title "Save As...", and a set of filters which give the options of listing "Source" (*.pl), "Object" (*.pc) or "Prolog" (*.pl and *.pc) files. The name "default" will be preloaded into the edit window, and any filename entered without an extension will be given ".pl" (note that the dot (.) must not be included explicitly). Upon completion, the single selected file name is returned:
Notes

Three common dialog box predicates are dedicated to obtaining filenames from the user: brsbox/2, opnbox/5, savbox/5. The first of these is used to return a folder name, while the latter pair share the majority of their features, but as their names suggest, one is used when opening files for loading (input), and the other when creating files for saving (output). This leads to the main differences in their respective behaviours, as detailed in the following paragraphs.

When loading, compiling or otherwise processing existing files, it is often desirable to be able to select several files at once; furthermore, it is essential that any such files are known to exist, and to be available for read access: opnbox/5 both allows multiple file selection, and performs its own existence and access permission checks on all selected files.

When "saving as" or otherwise creating new files, it is usually desirable to specify just one file at a time; furthermore, it is helpful to display a warning if a named (selected) file already exists, to avoid the user inadvertently overwriting important data: savbox/5 both limits selections to one file at a time, and displays a warning if the chosen file exists, giving the user the option to try again.

When searching for file locations other than when saving or loading, it can be handy to be able to select a folder directly, rather than one or more files contained within it: brsbox/2 provides exactly this feature.

None of these predicates actually opens the file: it is up to the application to pass the resulting file names to predicates such as open/2, fcreate/5, see/1, tell/1, and so on.
**save_predicates/2**

save the specified predicates to the named object file

```
save_predicates( Predicates, File )
```

+Predicates <atom>, <pred_spec>, <list> or <conjunction>
+File <file_spec>

**Comments**

This saves the object code for one or more *Predicates* to the given *File*. The given *Predicates* may be any of the following types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;atom&gt;</td>
<td>save all predicates of the given name, irrespective of their arity</td>
</tr>
<tr>
<td>&lt;pred_spec&gt;</td>
<td>this takes the form, &quot;atom/integer&quot; (name/arity), and the single specified predicate is saved</td>
</tr>
<tr>
<td>&lt;list&gt;</td>
<td>one or more of the first two options may be combined in a list</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>one or more of the first two options may be combined in a conjunction</td>
</tr>
</tbody>
</table>

**Examples**

Consider the following program, which comprises the four predicates, *foo/1*, *foo/2*, *bar/1* and *bar/2*:

```
foo( one ).
foo( two, three ).
bar( four ).
bar( five, six ).
```

Once compiled, both "foo" predicates (*foo/1* and *foo/2*) can be saved into a file called "file1.pc" as follows:

```
?- save_predicates( foo, file1 ).
yes
```

If desired, the "foo/2" predicate can be saved by itself:

```
?- save_predicates( foo/2, file2 ).
yes
```

The next call saves *foo/1* together with both "bar" predicates:
?- save_predicates( [foo/1,bar], file3 ).
yes

Notes

The `save_predicates/2` predicate provides a method for saving existing programs of all types. It is the only mechanism available for saving programs that have been compiled with the optimising compiler, but is equally able to save incrementally compiled and dynamic predicates, both of which will have been compiled with the incremental compiler. The files created by this predicate are in a binary format which is effectively non-readable by humans; however, in conjunction with `tell/1` and `told/0`, the `listing/1` predicate provides a simple way to save incrementally compiled programs as source files.

There is a curious anomaly in `save_predicates/2`: Edinburgh Prolog systems lay great emphasis on the use of "name/arity" predicate specifications. A predicate called "foo/1" is considered to be entirely unrelated to and independent from another called "foo/2"; however, `save_predicates/2` allows predicates of different arities to be referred to by a single name!

This is one of a series of file handling predicates which make use of "logical" file names: `FileSpec` can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to `file_search_path/2`, and are expanded into atom file names by the `absolute_file_name/3` predicate before being used.
scan/3

scan or copy up to first best string in current input stream

\[
\text{scan( Scan, Mode, Scanned )}
\]

+Scan <list_of <string>>
+Mode <integer> in the range [0..3]
-Scanned <variable>

Comments

This searches the current input stream for the first, best match of any one string in the given Scan list, either ignoring or exactly matching the case and optionally sending mismatched text to the current output stream, according to the given Mode. The result of the scan operation is returned as a string in Scanned. If Scan is an empty list, this predicate searches for the next non-white-space character on the current input stream. Mode may have any of the following settings:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Match exact case, perform no output</td>
</tr>
<tr>
<td>1</td>
<td>Ignore case, perform no output</td>
</tr>
<tr>
<td>2</td>
<td>Match exact case, copy mismatch to output</td>
</tr>
<tr>
<td>3</td>
<td>Ignore case, copy mismatch to output</td>
</tr>
</tbody>
</table>

If a match is found in the input stream, Scanned is bound to a string representing the exact text of that match. In modes 1 and 3, this may be of a different case to the original string in the Scan list. If no match is found, an empty string (``) is returned.

Examples

Assume that a file called "foo" exists in the current directory, containing the text:

```
the quick brown fox jumps over a lazy dog
```

This file can be opened for reading by the call:

```
?- fcreate( i, foo, 0, 0, 0 ).
yes
```

Now we will use scan/3 to search for the strings, "A" and "fox", returning a string that indicates the first, best match:
?- input( i ), scan( ['A', 'fox'], 0, S ), input( 0 ).          <enter>
S = `Fox`

All characters in the file, up to and including that matched by the first matching string in the Scan list, are treated as having been read, as can be shown by the following call to inpos/1:

?- input( i ), inpos( P ), input( 0 ).              <enter>
P = 19

Using modes 2 or 3, scan/3 can output mismatched characters along the way, as shown by the next example:

?- input( i ), scan( ['A', 'fox'], 3, S ), nl, input( 0 ).       <enter>
jumps over
S = `a`

Here, all characters have been output, from the space character after the word, "fox", and up to, but not including the first occurrence of the string, "a". Because the chosen Mode (3) also includes case insensitivity, the lowercase word, "a", was found, even though the string in the Scan list was specified in uppercase, "A". Finally, let's search for something that is not in the file, outputting as we go:

?- input( i ), scan( ['123'], 2, S ), nl, input( 0 ).          <enter>
lazy dog
S = ``

The remainder of the file, in this case the phrase " lazy dog", is output, but the Scanned string returns as an empty string (``). This string can only occur when the none of the items being searched for are present on the input stream.

In the next example, scan/3 will search for and return the whole of the next block of white space, outputting any characters before them because it is in echo mode Mode (2):

?- scan( [], 2, S ) <- `abc-M-Jdef` ~> W.          <enter>
S = `~M~J` ,
W = `abc`

Notes
The scan/3 predicate is very powerful, performing a parallel search for any number of strings simultaneously, and returning the first, best match. For example, consider a file, "bar", containing the nonsense text:
the fool was not fooled by his food fooler

The following call can be used to open this file and set it for subsequent input:

```prolog
?- fcreate( i, bar, 0, 0, 0 ), input( i ).
yes
```

Let's search from the start of the file for the first and best match out of the strings, "foo", "fool", and "fooled":

```prolog
?- scan( ['foo','fool','fooled'], 0, S ).
S = 'fool'
```

At first, this might look like the wrong answer: after all, "fooled" was in the list alongside "fool"; however, as described earlier, scan/3 searches for the first and best match: as it reads the file from the beginning, it first encounters the match, "foo"; however, when it looks further ahead, it finds that the next character, "l", allows for the better match, "fool". One more character is read, but this time it is found to be a space, so the first best match, "fool", is returned. Let's make the same call a second time:

```prolog
?- scan( ['foo','fool','fooled'], 0, S ).
S = 'fooled'
```

Much as before, "foo" is quickly found; a peek ahead finds another matching character, "l", suggesting that "fool" is a better match than "foo"; two more looks ahead find "e" and "d" respectively, so the even better match, "fooled", is discovered, and it is this one which is returned. Let's make yet another call:

```prolog
?- scan( ['foo','fool','fooled'], 0, S ).
S = 'foo'
```

Again, "foo" is found quickly, but the next peek ahead sees "d", which does not match any of the other strings, so "foo" is returned as the best match this time. Here's another call:

```prolog
?- scan( ['foo','fool','fooled'], 0, S ).
S = 'fool'
```

As ever, "foo" is found at once: look-ahead discovers another matching character, "l", and a further peek finds an even better match, "e"; however, the next character is found to be "r", which doesn't match anything, so it turned out that the "e" was irrelevant, and the best
match this time is "fool". A final call reveals that there are no further matches:

?\- scan([`foo`, `fool`, `fooled`], 0, S).
S = ``

The empty string, "``", indicates that none of the input strings could be matched, so we can close the file:

?\- fclose(i).
yes

The scan/3 predicate performs its apparent magic using tries, a special form of tree structure which allows an arbitrary number of strings to be represented from their common roots. Its initial implementation was inspired by the specific requirements of the efficient parsing of Extensible Markup Language (XML), buts its applications extend far beyond this.

Just one example is the common need to swap two strings in a file, so that, say, all instances of "foo" become "bar", and vice versa. The common algorithm requires three steps, where "$$" is some unique text which does not appear in the file:

1) Find all instances of "foo", replacing with "$$"
2) Find all instances of "bar", replacing with "foo"
3) Find all instances of "$$", replacing with "bar"

Thanks to scan/3 and its parallel searching, this can be done in a single pass:

1) Find all instances of "foo" or "bar", replacing with the other

In fact, traditional algorithms for swapping or shuffling require N+1 steps, where "N" is the number of items being exchanged, and in every instance, scan/3 can always perform the same task in a single, efficient pass.

There are two text search predicates in WIN-PROLOG; find/3 and scan/3: the former searches for a match for a single specified string, while the latter scans for the first, best instance of one string in a given list. Where only one string is the target of the search, find/3 provides a slightly faster solution; additionally, it has a special feature which allows for fast scanning for non-white text, but in all other respects, scan/3 provides a considerably more flexible solution to text searching applications.

Scanning for white space is a feature that was added in WIN-PROLOG 6.000, to simplify the parsing of free text text blocks.
sckhdl/2

convert between logical and raw windows socket handle

\hspace{1em} sckhdl( Socket, Raw )

?Socket <atom> or <variable>
?Raw <integer> or <variable>

Comments
This converts between a Windows Sockets (Winsock) socket handle and its Raw, integer counterpart. When Socket is an unbound variable, Raw must be an integer: an attempt is made to check that this integer identifies a socket, and if not the predicate fails. If successful, a further test is made to see whether the socket was created by WIN-PROLOG: if so, its logical window handle is bound to Socket as an atom; otherwise, the raw socket handle is returned. If Socket is an atom, a check is made to see if it defines a currently-existing socket; if so, the raw handle of that window is unified with Raw. In both cases, sckhdl/2 fails if the specified socket does not exist.

Examples
The following call uses screate/2 to create a client socket that attempts to connect to the LPA website at www.lpa.co.uk; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

\hspace{1em} ?- screate( foo, `www.lpa.co.uk` ). <enter>
yes

Assuming the connection has succeeded, we can send a very simple HTTP request; normally, we would use ssend/2 for this operation, but for the purpose of illustration, we'll do it the long way this time, using sckhdl/2 to retrieve the raw socket handle, and winapi/4 to call the Winsock DLL directly:

\hspace{1em} ?- sckhdl( foo, Sock ),
    winapi( (wsock32,send), [Sock,`GET /index.htm~M~J~M~J`,18,0], 0, Sent ). <enter>
Sock = 28 ,
Sent = 18

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "/index.htm"; we can receive this with a call to srecv/2:

\hspace{1em} ?- srecv( foo, Text ). <enter>
Text = `<HTML> ... </HTML>~M~J`
where "..." will depend upon the contents of the file. Once the page has been received, we can close the socket with \texttt{sclose/1}, releasing its resources:

\begin{verbatim}
?- sclose( foo ).
yes
\end{verbatim}

\section*{Notes}

The \texttt{sckhdl/2} predicate performs two primary functions: firstly, it provides a quick test for the existence of any given socket: \texttt{"sckhdl( Socket, _ )"} will simply succeed or fail depending upon whether or not the socket exists. Secondly, it converts window handles to and from their "raw" integer state. The socket predicates within \texttt{WIN-PROLOG} work with "logical" socket handles; however, external functions called with \texttt{winapi/4} cannot interpret such handles.

Starting in version 4.600, \texttt{WIN-PROLOG} has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via \texttt{winapi/4}.

The full set of Winsock predicates includes \texttt{sckhdl/2}, \texttt{sclose/1}, \texttt{screate/2}, \texttt{sdict/2}, \texttt{srecv/2}, \texttt{ssend/2} and \texttt{sstat/2} together with \texttt{socket_handler/1} and \texttt{socket_handler/2}; see Appendix P for a detailed overview.
sclose/1

close a windows sockets (winsock) tcp/ip socket

sclose( Name )
+Name <atom>

Comments
This closes a Windows Sockets (Winsock) socket of the given Name, gracefully terminating any connection that may associated with it.

Examples
The following call uses screate/2 to create a client socket that attempts to connect to the LPA website at www.lpa.co.uk; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

?- screate( foo, `www.lpa.co.uk` ).
yes

Assuming the connection has succeeded, we can send a very simple HTTP request using ssend/2:

?- ssend( foo, `GET /index.htm~M~J~M~J` ).
yes

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "index.htm"; we can receive this with a call to srecv/2:

?- srecv( foo, Text ).
Text = `&lt;HTML&gt; ... &lt;/HTML&gt;~M~J`

where "..." will depend upon the contents of the file. Once the page has been received, we can close the socket with sclose/1, releasing its resources:

?- sclose( foo ).
yes

Notes
Starting in version 4.600, WIN-PROLOG has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications;
datagram sockets can still be used, but will require direct Winsock calls via *winapi/4*.

The full set of Winsock predicates includes `sckhdl/2`, `sclose/1`, `screate/2`, `sdict/2`, `srecv/2`, `ssend/2` and `sstat/2` together with `socket_handler/1` and `socket_handler/2`; see Appendix P for a detailed overview.
\textbf{screate/2}

\textit{create a windows sockets (winsock) tcp/ip socket}

\begin{verbatim}
screate( Name, Param )
+Name <atom>
+Param <integer>, <atom>, <string> or <conjunction>
\end{verbatim}

\textbf{Comments} This creates a Windows Sockets (Winsock) socket of the given \texttt{Name}, in one of four modes, depending on the data type of \texttt{Param}, as shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{&lt;integer&gt;}</td>
<td>a listening socket is created at the TCP/IP port indicated by the given integer, which should be in the range 0..65535</td>
</tr>
<tr>
<td>\texttt{&lt;atom&gt;}</td>
<td>this must name an existing listening socket, and is used to create a new server socket to accept a connection request from the given socket</td>
</tr>
<tr>
<td>\texttt{&lt;string&gt;}</td>
<td>this identifies a host machine, using either its domain name or its IP address, requesting a client connection at port 80 of that machine</td>
</tr>
<tr>
<td>\texttt{&lt;conjunction&gt;}</td>
<td>this contains two values, a string and an integer, respectively which identify a host machine and a port number. The string may be either a domain name or an IP address, and the integer should be in the range 0..65535</td>
</tr>
</tbody>
</table>

\textbf{Examples} The following call uses \texttt{screate/2} to create a client socket that attempts to connect to the LPA website at \texttt{www.lpa.co.uk}; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

\begin{verbatim}
?- screate( foo, `www.lpa.co.uk` ).
yes
\end{verbatim}

Assuming the connection has succeeded, we can send a very simple HTTP request using \texttt{ssend/2}:

\begin{verbatim}
yes
\end{verbatim}

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "/index.htm"; we can receive this
with a call to \texttt{srecv/2}:

\begin{verbatim}
?- \texttt{srecv( foo, Text ).}
Text = `\text少HTML> ... </HTML>`~M~J`
\end{verbatim}

where "..." will depend upon the contents of the file. Once the page has been received, we can close the socket with \texttt{sclose/1}, releasing its resources:

\begin{verbatim}
?- \texttt{sclose( foo ).}
yes
\end{verbatim}

\textbf{Notes} Starting in version 4.600, \textbf{WIN-PROLOG} has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via \texttt{winapi/4}.

The full set of Winsock predicates includes \texttt{sckhdl/2, sclose/1, screate/2, sdict/2, srecv/2, ssend/2 and sstat/2} together with \texttt{socket_handler/1} and \texttt{socket_handler/2}; see Appendix P for a detailed overview.
**sdict/2**

return a list of windows sockets (winsock) tcp/ip sockets

\[ sdict( \text{Flag, Dict} ) \]

+Flag <integer> in the domain \{-1,0,1\}

-Dict <variable>

**Comments**

This returns a list of all the currently open Windows Sockets (Winsock) sockets, as specified in the given \text{Flag} (see \text{dict/2}), and binds it to the variable \text{Dict}.

**Examples**

The following call uses \text{screate/2} to create a client socket that attempts to connect to the LPA website at \text{www.lpa.co.uk}; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

\[ ?- \text{screate( foo, `www.lpa.co.uk`)}. \]

yes

Now we'll check for a list of all currently open sockets (assuming that "foo" is the only file created so far):

\[ ?- \text{sdict( 0, D )}. \]

D = [foo]

The following call can be used to close all sockets:

\[ ?- \text{sdict( 0, D ), forall( member(S,D), sclose(S) )}. \]

D = [foo],

S = _

**Notes**

Starting in version 4.600, \text{WIN-PROLOG} has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via \text{winapi/4}.

The full set of Winsock predicates includes \text{skhdl/2, sclose/1, screate/2, sdict/2, srecv/2, ssend/2 and sstat/2} together with \text{socket_handler/1} and \text{socket_handler/2}; see Appendix P for a detailed overview.
**see/1**

set the current input stream

```prolog
see( Stream )
```

+Stream <file_spec>

**Comments**
This sets the current input Stream, which may be a disk file or the special atom, "user", the latter which refers to the console input device. If Stream refers to a disk file which is not yet open, it is automatically opened in "read" mode (see open/2); if the file was already open, input continues from the point immediately after the previously-read character.

**Examples**
Assuming that a file called "foo" is present in the Prolog home directory, the following call will open it (if not already open), read a term, say "hello(world)", from the file, and finally close it and reset the input stream to the console (user) device:

```prolog
?- see(prolog(foo)), read(T), seen.
T = hello(world)
```

**Notes**
Three related predicates, see/1, seeing/1 and seen/0, support high-level control over the input stream, and provide for the easy handling of sequential access files: however, it does not have direct access to special numerically named devices or to memory files. Another predicate, input/1, provides low-level control over all types of WIN-PROLOG input stream.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
\textbf{seed/1}

\textit{get or set the pseudo random number generator seed}

\begin{verbatim}
seed( Seed )

?Seed          <variable>, <number> or <conjunction>
\end{verbatim}

\textbf{Comments} This gets or sets the 64-bit integer \texttt{Seed} used by the \textit{pseudo random number generator ("PRANG")}. If \texttt{Seed} is an unbound variable, then it is bound to a conjunction of two integers: between them, these make up the two halves of the 64-bit integer seed. If \texttt{Seed} is an integer, it is converted to a 64-bit floating point number, whose bit pattern is used as the 64-bit integer seed; if \texttt{Seed} is already a floating point number, its bit pattern is used directly. Finally, if \texttt{Seed} is a conjunction of two integers, they are used to set the two respective halves of the 64-bit integer seed.

\textbf{Examples} The following call picks up the current seed, sets the new seed to a 64-bit integer whose bit pattern is the same as that of the floating point number, "0.81e2", generates a pseudo random number using \texttt{is/2}, before restoring the original seed:

\begin{verbatim}
?- seed( S ), seed( 81 ), R is rand(100), seed( S ).
\end{verbatim}

\begin{verbatim}
S = (2084826112,1117618496),
R = 35.3748761117458
\end{verbatim}

\textbf{Notes} The PRANG used in \textbf{WIN-PROLOG} utilises a fast, carefully researched, 64-bit integer "linear congruential" algorithm that generates a sequence of $2^{64}$ distinct numbers; each number ("N") is computed from its predecessor ("P") as follows:

\begin{equation}
N = (P \times 6364136223846793005 + 8110242468091) \mod 2^{64}
\end{equation}

When \textbf{WIN-PROLOG} starts up, the PRANG's seed is set from a mixture of date and time, resulting in a unique sequence of numbers on every execution. Sometimes, however, such as when running simulations, it may be desirable to generate a given "random" sequence on several successive occasions: it is with this in mind that \texttt{seed/1} has been provided. Note that it is a good idea to preserve the existing seed around any such "forced" sequence, as shown in the example above, to avoid corrupting other random sequences that may be running.

Pseudo random numbers are obtained with the "rand(\texttt{X})" function in \texttt{is/2} and related arithmetic evaluation predicates.
seeing/1

get the current input stream

seeing( Stream )

?Stream <file_spec> or <variable>

Comments
This gets the current input Stream, which may be a disk file or the special atom, "user", the latter which refers to the console input device.

Examples
Assuming that a file called "foo" is present in the Prolog home directory, the following call will pick up the current input stream, open the file (if not already open), read a term, say "hello(world)", from the file, and finally reset input to the original stream, without closing the file:

?- seeing( C ), see( prolog(foo) ), read( T ), see( C).
C = user,
T = hello(world)

Notes
Three related predicates, see/1, seeing/1 and seen/0, support high-level control over the input stream, and provide for the easy handling of sequential access files: however, it does not have direct access to special numerically named devices or to memory files. Another predicate, input/1, provides low-level control over all types of WIN-PROLOG input stream.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
**seen/0**

*close the current input stream and revert to user input*

**seen**

**Comments**
This closes any file associated with the current input stream, and resets input to the "user" (console input) device.

**Examples**
Assuming that a file called "foo" is present in the Prolog home directory, the following call will open it (if not already open), read a term, say "hello(world)", from the file, and finally close it and reset the input stream to the console (user) device:

```prolog
?- see(prolog(foo)), read(T), seen.  
    T = hello(world)  
```

**Notes**
Three related predicates, `see/1`, `seeing/1` and `seen/0`, support high-level control over the input stream, and provide for the easy handling of sequential access files: however, it does not have direct access to special numerically named devices or to memory files. Another predicate, `input/1`, provides low-level control over all types of WIN-PROLOG input stream.
setof/3

return sorted sets of terms in solutions of a given goal

\text{setof( Term, Goal, List )}

?Term <term>
+Goal <goal>
?List <variable> or <list>

Comments

This predicate returns sorted sets of solutions that would normally be obtainable only by failing and backtracking through a query. It succeeds if \text{List} can be unified to a non-empty list of instances of \text{Term} such that \text{Goal} is true. The \text{Term} may be any Prolog term, and \text{Goal} may be any Prolog goal, with or without calls to the existential quantifier predicate, \^{/2}, which is used to alter the partitioning of solutions (see "notes" below).

Examples

Consider the database relation:

\begin{verbatim}
foo( 1, c ).
foo( 2, a ).
foo( 1, b ).
foo( 2, a ).
\end{verbatim}

The following call returns sets of solutions of "X" in the goal "foo(Y,X)", partitioned according to different values of "Y":

\begin{verbatim}
?- setof( X, foo(Y,X), Z ).
<enter>
X = _,
Y = 1 ,
Z = [b,c] ;

X = _,
Y = 2 ,
Z = [a] ;

no
\end{verbatim}

If it is desired to get all solutions irrespective of the bindings of "Y", then "existential quantification" can be used:
The `setof/3` predicate is very closely related to `bagof/3`. The only difference in behaviour is that the former sorts the returned sets, removing duplicate solutions in the process.

The "existential quantification" feature of both `bagof/3` and `setof/3` often causes confusion. In general, solutions to both predicates are simply partitioned according to the bindings of those variables mentioned in the Goal but not in the Term. Consider the command:

```
?- setof( (X,Y), foo(X,Y,A,B), Z ).
```

This would return a series of lists of (X,Y) solutions for the hypothetical predicate, `foo/4`, with each list corresponding to a different binding combination of (A,B). Supposing it was desired still to partition according to bindings of B, but not A, all that is required is to mention "A" in the Goal as the left operand of a call to `^/2`:

```
?- setof( (X,Y), A^foo(X,Y,A,B), Z ).
```

This time, the series of lists of (X,Y) solutions will be split only on different bindings of B: effectively, A will be ignored during the partitioning.

The "^" atom is defined as an infix, left-associate operator, so a series of variables can be mentioned simply by extending (deepening) the sequence of calls to `^/2`:

```
?- setof( (X,Y), A^B^foo(X,Y,A,B), Z ).
```

Please note that outside of its special interpretation in the `bagof/3` and `setof/3` predicates, `^/2` has no meaning apart from that of the goal that constitutes its second argument; because of this, the command:

```
?- A ^ foo(A,B).
```

is in every respect identical to the command:

```
?- call( foo(A,B) ).
```
One remaining solution set predicate, \texttt{findall/3}, returns all solutions to a goal, without partitioning or sorting: in this case, use of existential quantification is meaningless.
**sha/3**

*perform a secure hash algorithm on data from current input*

```prolog
sha( Wanted, Hashed, Result )
```

**Comments**
This hashes up to the number of characters *Wanted*, from the current 8-bit input stream, computes the 256-bit secure hash algorithm (SHA-256) on the data, and returns the number of characters successfully *Hashed* as well as the *Result* SHA-256 message digest as a list of eight 32-bit integers.

**Examples**
The following command hashes all characters in a string, returning its length and SHA-256:

```prolog
?- sha( -1, Count, SHA ) <= `Hello World`.
```

```
Count = 11 ,
SHA = [-1517181228,200548416,1241585459,-810045040,
       -701733441,198026027,1471313881,-1382083474] % X86
SHA = [2777786068,200548416,1241585459,3484922256,
       3593233855,198026027,1471313881,2912883822] % X64
```

**Notes**
An input value of "-1" interpreted is interpreted as "process the entire file", using a 53-bit (X86) or 64-bit (X64) counter. Because this predicate digests directly from input, without using Prolog's internal data structures, it can handle large amounts of data very efficiently.

There are three data checking predicates in **WIN-PROLOG**, providing differing levels of security. The first is *crc/3*, which performs the 32-bit "Cyclic Redundancy Check", or "CRC32", a reliable checksum a given data stream. The other two predicates, *mdf/3* and *sha/3*, perform the 128-bit "Message Digest Five", or "MD5", and 256-bit "Secure Hash Algorithm", or "SHA-256" message digests respectively.

While CRC32 is fine for quick checks, it is not "secure": there is a $2^{-32}$ chance of two data streams having the same CRC32, and it is easy to create files with a given CRC32 value. Meanwhile, MD5 is relatively secure, with only a $2^{-128}$ chance of a clash: it is also not thought possible to design a file with a specific MD5, although recent attacks are making it look less than ideal. For maximum security, use SHA-256, which has only a $2^{-256}$ chance of a collision, and is considered computationally impossible to crack.

Note that *sha/3* computes the SHA-256 secure hash as a set of 8, 32-bit integers: normally, such hashes are represented as a 64-char-
acter hexadecimal string, simply by concatenating the big-endian hexadecimal representations of these integers. Consider the following program:

```prolog
sha( File ) :-
  fcreate( input, File, 0, 0, 0 ),
  input( Current ),
  input( input ),
  sha( -1, _, List ),
  input( Current ),
  fclose( input ),
  forall( member( Int, List ),
    fwrite( r, 8, 16, Int )
  ),
  nl.
```

This takes as its input the name of a single file, which is opened in "ISO" (8-bit, ISO/IEC 8859-1) mode, and then hashed using `sha/3`; each of the integers in the resulting hash is then written to the current output stream in big-endian hexadecimal format by `fwrite/4`. Suppose we had a text file, "HELLO.TXT", containing just the characters, "hello world~M~J"; once the above program is compiled, the following call will display the conventional SHA-256 signature of the file:

```prolog
?- sha( 'hello.txt' ).
```

Note that both CRC-32 and SHA-256 values are traditionally output in big-endian format, making direct use of `fwrite/4` possible when displaying the results of `crc/3` and `sha/3` respectively, but MD5 values are conventionally output in little-endian format, requiring some "bit twiddling" to display the output of `mdf/3` correctly.
show_dialog/1

show a modeless dialog

show_dialog( Window )

+Window <window_handle>

Comments

This shows the given dialog Window, immediately continuing execution of the calling program. The behaviour of the dialog itself is governed by its attached "window handler".

Examples

The following program creates a very simple dialog with two buttons, labelled "Hello" and "World" respectively:

create_foo :-
    Dstyle = [ws_popup,ws_caption,dlg_ownedbyprolog],
    Bstyle = [ws_child,ws_visible,bs_pushbutton],
    wdcreate( foo, `foo`, 100, 100, 170, 75, Dstyle ),
    wccreate( (foo,1), button, `Hello`, 10, 10, 70, 30, Bstyle ),
    wccreate( (foo,2), button, `World`, 90, 10, 70, 30, Bstyle ).

When compiled, and the following command is entered, the dialog "foo" is created, but not yet displayed:

?- create_foo.
yes

To show the dialog modelessly, make the following call:

?- show_dialog( foo ).
yes

The dialog will appear, with its two buttons: at this point, clicking the buttons will have no effect. To wait for and retrieve a button press, enter the following command:

?- wait_dialog( foo, X ).

If the user presses the "Hello" button, its lowercase text will be returned as follows:
X = hello

Unlike modal dialogs, which handled by `call_dialog/2`, this dialog remains on the screen even after a button click. It is therefore possible to call the dialog again to retrieve further button clicks:

?- `wait_dialog( foo, X )`.  
X = world

When it is finally desired to hide the dialog, the following call can be made:

?- `hide_dialog( foo )`.  
yes

Notes

The example shown above uses the default window handler, `window_handler/4`, which simply binds the lowercase text of any clicked button to the `Result`. It is the action of binding this variable which actually "completes" the dialog.

Window handlers are simply arity-4 programs which intercept messages destined for a window (including dialogs and their controls), allowing arbitrary actions to take place. Consider the following example:

```prolog
foo_handler( (foo,1), msg_button, _, Result ) :-
  time( _, _, _, H, M, S, _ ),
  Result = finished_at(H,M,S).

foo_handler( Window, Message, Data, Result ) :-
  window_handler( Window, Message, Data, Result ).
```

The first clause detects that a button called "(foo,1)" has been clicked, and completes the dialog by returning the structure "finished_at(H,M,S)" rather than the atom "hello". All other messages are passed to the default window handler. Once compiled, the new window handler is attached to the dialog window with a call to `window_handler/2`, as shown here:

?- `window_handler( foo, foo_handler )`.  
yes

If the dialog is invoked once again, and the "(foo,1)" button (labelled "Hello") is pressed, the time will be returned instead of the atom "hello":

X = hello
?- show_dialog( foo ).
   yes

?- wait_dialog( foo, X ).
   X = finished_at(10,8,42)

?- hide_dialog( foo ).
   yes

Please note that "modal" and "modeless" dialogs have exactly the same window handlers: in fact, in WIN-PROLOG, these two types of dialog are one and the same thing as each other. All that determines whether a dialog behaves in a modal or modeless fashion at run-time is whether it is invoked by call_dialog/2 (modal) or show_dialog/1 (modeless). A third class of dialog, sometimes referred to as a "wizard", behaves semi-modally: like a modal dialog, it waits for input from the user when called with wait_dialog/2, but like a modeless dialog, it remains visible and in focus between successive calls, until finally hidden by invoking hide_dialog/1.
**simple/1**

*test whether a term is a variable, atom, number or string*

```prolog
simple( Term )
```

**Comments**  
This succeeds if `Term` is simple, which means that it is currently an unbound variable, or is bound to a number (integer or float), atom or string.

**Examples**  
The following calls test various cases:

```prolog
?- simple( 123 ).  %
yes

?- simple( [1,2,3] ).  %
no

?- simple( 'foo' ).  %
yes

?- simple( 'bar' ).  %
yes

?- simple( X ).  %
X = _
```

**Notes**  
This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

The term "simple" means non-compound, implying that the term is "indivisible". Simple entities can, however, be split by a variety of term conversion predicates.
size_dialog/2

automatically resize a dialog

size_dialog( Window, Mode )

+Window  <window_handle>
+Mode    <integer> in the domain {0,1}

Comments
This automatically resizes the given dialog Window, ensuring all its controls are visible, according to the given Mode:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>resize dialog, but leave in current position</td>
</tr>
<tr>
<td>1</td>
<td>resize and reposition dialog</td>
</tr>
</tbody>
</table>

Examples
The following program creates a very simple dialog with two buttons, labelled "Hello" and "World" respectively:

create_foo :-
    Dstyle = [ws_popup,ws_caption],
    Bstyle = [ws_child,ws_visible,bs_pushbutton],
    wdcreate( foo, `foo`, 0, 0, 99, 99, Dstyle ),
    wccreate( (foo,1), button, `Hello`, 10, 10, 70, 30, Bstyle ),
    wccreate( (foo,2), button, `World`, 90, 10, 70, 30, Bstyle ).

When compiled, and the following command is entered, the dialog "foo" is created, but not yet displayed:

?- create_foo.
yes

To show the dialog, and return the text of whichever of the two buttons was pressed, enter the following command:

?- show_dialog( foo ).
yes

The dialog will appear, and remain on the screen as control returns to the WIN-PROLOG "console window"; when the user presses either
button, the dialog automatically disappears. However, because the dialog was created with arbitrary size and position parameters (0, 0, 99, 99), it is a small box at the top left of the screen, in which one of its two buttons is largely obscured. The next command corrects this:

```
?- size_dialog( foo, 1 ), show_dialog( foo ).
```

This time the dialog will automatically be sized to fit around its controls, and will then appear, slightly down and to the right of the WIN-PROLOG "main window"; again, when the user presses either button, the dialog automatically disappears.

**Notes**

One of the perennial problems with writing dialogs is that different versions of Windows, or even different "desktop themes" within a single version of Windows, change the proportions of the title bar, menu and client area of dialogs. For example, a perfectly designed Windows 95 dialog, containing nicely spaced controls and an even border around the edge, might lose the bottoms of some of its controls when running under Windows XP.

The present predicate overcomes this problem, by measuring the amount of space to the left and above the left-most and top-most controls respectively, and then resizing the dialog window to ensure that the same amounts of space exist to the right and bottom of the right-most and bottom-most controls respectively. It is thus possible to create dialogs which display in optimal proportions on all versions of Windows and using any desktop theme, however wild.
skip/1

evaluate and skip past a given character on current input

\[ \text{skip}( \text{Expression} ) \]

+Expression <expr>

Comments  
This evaluates the given \textit{Expression} by calling \textit{is/2}, interprets the result as a character code, and then searches for this character on the current input stream. Searching stops either when the character is found, or when end-of-file is reached: in either case, this predicate succeeds. An error is generated if the \textit{Expression} does not evaluate to an integer.

Examples  
In most cases, this predicate is used to search for characters which are already represented as character codes, for example in the following call, where the input, "abc@" is consumed, leaving "write( hello ), nl." on the command line:

\[
\text{?- } \text{skip}( 64 ).
\]

\[
|: \text{abc@write( hello ), nl.}
\]

\[
\text{yes}
\]

\[
\text{hello}
\]

\[
\text{yes}
\]

Notes  
This is one of several "skip" predicates, each of which is designed to scan the current input stream for a specified or implicit pattern. The set includes \textit{skip/1, skip_comments/0, skip_layout/0, skip_line/0 and skip_term/0}. Another predicate, \textit{find/3}, provides considerably more flexible "skipping" features, including being able to search for case sensitive or case insensitive text, and optionally writing of all non-matched characters to the current output stream.
skip_comments/0

skip past leading comments on the current input stream

**skip_comments**

**Comments**
This reads from the current input stream, skipping past any white space characters (those with character code values of less than 21h) and comments. Searching stops either when a non-comment character is found, or when end-of-file is reached: in either case, this predicate succeeds.

**Examples**
In the following call, the leading comment is read from the input string, allowing `fread/4` to return a string, "foo", without the leading comment or spaces:

```prolog
?- (skip_comments, fread( S, 0, -1, S )) <- `/*hello*/ foo`. <enter>
S = `foo`
```

**Notes**
This is one of several "skip" predicates, each of which is designed to scan the current input stream for a specified or implicit pattern. The set includes `skip/1`, `skip_comments/0`, `skip_layout/0`, `skip_line/0` and `skip_term/0`. Another predicate, `find/3`, provides considerably more flexible "skipping" features, including being able to search for case sensitive or case insensitive text, and optionally writing of all non-matched characters to the current output stream.

Unlike the other "skip" predicates, `skip_comments/0` is not able to process the console (user) device. This is because it needs to read up to two characters ahead to determine whether a comment is about to begin, and it needs to be able to "rewind" the input stream if not: the console (user) device does not support the necessary stream repositioning predicates.
skip_layout/0

skip past white space on the current input stream

skip_layout

Comments
This reads from the current input stream, skipping past any white space characters (those with character code values of less than 21h). Searching stops either when a printable character is found, or when end-of-file is reached: in either case, this predicate succeeds.

Examples
In the following call, the leading white space characters are read from the console input stream, allowing fread/4 to return a string, "foo", without leading spaces:

?- skip_layout, fread( S, 0, -1, S ).
   |:     foo
   S = `foo`

Notes
This is one of several "skip" predicates, each of which is designed to scan the current input stream for a specified or implicit pattern. The set includes skip/1, skip_comments/0, skip_layout/0, skip_line/0 and skip_term/0. Another predicate, find/3, provides considerably more flexible "skipping" features, including being able to search for case sensitive or case insensitive text, and optionally writing of all non-matched characters to the current output stream.
skip_line/0

skip past the end of line on the current input stream

skip_line

Comments  This reads from the current input stream, skipping past all characters including the end of line. Searching stops either when the end of a
line is found (<cr> or <cr><lf>), or when end-of-file is reached: in either case, this predicate succeeds.

Examples  In the following call, a whole line of input is discarded from the console input stream, allowing fread/4 to return the string, "bar", from the
following line:

    ?- skip_line, fread( s, 0, -1, S ).
    |: foo
    |: bar
    S = `bar`

Notes  This is one of several "skip" predicates, each of which is designed to scan the current input stream for a specified or implicit pattern. The
set includes skip/1, skip_comments/0, skip_layout/0, skip_line/0 and skip_term/0. Another predicate, find/3, provides considerably more
flexible "skipping" features, including being able to search for case sensitive or case insensitive text, and optionally writing of all non-
matched characters to the current output stream.
skip_term/0

skip past the end of a term on the current input stream

skip_term

**Comments**

This reads from the current input stream, skipping past any characters up to and including the end of a Edinburgh syntax term. Searching stops either when the terminating ".<period><space>" sequence is found, or when end-of-file is reached: in either case, this predicate succeeds.

**Examples**

In the following call, the term "foo(1,2,3)" is skipped, allowing fread/4 to return a string, "bar", from directly after the term:

```
?- skip_term, fread( S, 0, -1, S ).
|: foo(1,2,3). bar
```

**Notes**

This is one of several "skip" predicates, each of which is designed to scan the current input stream for a specified or implicit pattern. The set includes skip/1, skip_comments/0, skip_layout/0, skip_line/0 and skip_term/0. Another predicate, find/3, provides considerably more flexible "skipping" features, including being able to search for case sensitive or case insensitive text, and optionally writing of all non-matched characters to the current output stream.
### sndmsg/5

**send a message to a window**

```prolog
sndmsg( Window, Message, Wparam, Lparam, Result )
```

- **Window**  
  `<window_handle>`
- **Message**  
  `<atom>`
- **Wparam**  
  `<term>`
- **Lparam**  
  `<term>`
- **Result**  
  `<variable>`

#### Comments
This sends the given named `Message` to the given `Window`, passing it the data parameters `Wparam` and `Lparam`: the `Result` of the call is returned as an integer. Each of the two data parameters may be one of the following types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;integer&gt;</code></td>
<td>the integer is passed directly as a data item</td>
</tr>
<tr>
<td><code>&lt;string&gt;</code></td>
<td>the string's text is stored in a temporary buffer, and its address is passed as a pointer</td>
</tr>
<tr>
<td><code>&lt;atom&gt;</code></td>
<td>this is a memory file created with <code>fcreate/5</code>: its address is passed as a pointer</td>
</tr>
</tbody>
</table>

#### Examples
The following call sets up a memory file called "buffer", with a size of 1024 bytes, in preparation for use with `sndmsg/5`:

```prolog
?- fcreate( buffer, [], -1, 1024, 0 ).
```

Once this buffer is created, it can be used as a parameter in calls to `sndmsg/5`. The next call sends a message to `WIN-PROLOG`'s "main window" ("0") which copies the window's title into the buffer, returning the number of characters in the name:

```prolog
?- sndmsg( 0, wm_gettext, 1024, buffer, R ).
```

```
R = 22
```

A call can be made to `wintxt/4` to retrieve the text from the buffer, before the latter is closed with `fclose/1`:

```prolog
?- wintxt( buffer, 0, 0, T ), fclose( buffer ).
```

```
T = `WIN-PROLOG - [Console]`
```
Notes

Many operations in the Windows environment (Windows API) are handled by the passing of messages to windows. Such "window messages" may be sent by the operating system itself, or by applications. At runtime, messages and their parameters are simply numbers, although most Windows compilers provide them with meaningful names and parameter data types at compile time. Because Prolog is a dynamic language, it would be restrictive to limit the naming of messages to compile time: sndmsg/5 has been provided specifically to give easy programming access to window messages.

The sndmsg/5 predicate is, in fact, nothing more than a front end to the all-powerful winapi/4 predicate: the latter allows virtually any Windows API function (as well as the majority of those in any 32-bit dynamic link library (DLL)) to be called. The "SendMessageW" and "SendMessageA" functions are simply part of the WIndows API, the former providing 16-bit WIDE char support in WinNT, and the latter 8-bit ANSI support in all versions of Windows. Effectively, this is how sndmsg/5 is implemented:

```
sndmsg( Window, Message, Wparam, Lparam, Result ) :-
    wndhdl( Window, Handle ),
    message_name( Message, Number ),
    ( running_in_windows_nt
      -> winapi( (user32,'SendMessageW'), [Handle,Number,Wparam,Lparam], 2, Result )
    ;  winapi( (user32,'SendMessageA'), [Handle,Number,Wparam,Lparam], 1, Result )
    ).

message_name( bm_getcheck, 16'00f0 ).
message_name( bm_getstate, 16'00f2 ).
...etc
```

This predicate supports the majority of Windows messages, including all those documented in version 2.0 of the Microsoft Windows SDK; the user is referred to the appropriate SDK manuals and help files for further information.
socket_handler/2

get or set the socket handler for the given socket

socket_handler( Socket, Handler )

+Socket <atom>
?Handler <atom> or <variable>

Comments
This gets or sets the name of an arity-3 predicate that will be used as the socket Handler for the given Windows Sockets (Winsock) Socket. If Handler is an unbound variable, it returns the name of the existing handler; otherwise, it sets the given Handler name to be associated with the Socket.

Examples
By default, all messages to sockets are handled by a predicate, socket_handler/3; you can see this by calling socket_handler/2 with a socket name, even before the socket has been created:

?- socket_handler( foo, H ).
    H = socket_handler

Now consider the following socket handler:

foo( Socket, Event, Error ) :-
    (  write( Socket = Event / Error ),
        nl ) ~> user,
    socket_handler( Socket, Event, Error ).

Once this code has been compiled, we can assign foo/3 to become the socket handler for our forthcoming socket, foo:

?- socket_handler( foo, foo ).
     yes

The following call uses screate/2 to create a client socket that attempts to connect to the LPA website at www.lpa.co.uk; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

?- screate( foo, `www.lpa.co.uk` ).
Assuming the connection has succeeded, momentarily after the above command has completed, our handler will display two events:

?- foo = sck_connect / 0
?- foo = sck_write / 0

In response to the "sck_write" event, we can send a very simple HTTP request using ssend/2:

?- ssend( foo, `GET /index.htm`).
yes

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "index.htm"; after a moment, we will see confirmation of this event as our program reports the following:

?- foo = sck_read / 0

Now we know we can receive the data with a call to srecv/2:

?- srecv( foo, Text ).
Text = `<HTML> ... </HTML>`

where "..." will depend upon the contents of the file. Another event will be reported, caused by the web server asking us to close the connection:

?- foo = sck_close/ 0

In response to this, we can close the socket with sclose/1, releasing its resources:

?- sclose( foo ).
yes

A final piece of housekeeping is to remove the association of the socket name, "foo", with the handler, "foo/0". We do this with socket_handler/2, passing in either the name, "socket_handler" or the empty list, "[]", which is a shorthand for the same:

?- socket_handler( foo, [] ).
Notes

The `socket_handler/2` simply maintains a local database of the names of Windows Sockets (Winsock) sockets and the names of their corresponding "socket handler" predicates: when an event occurs in any given socket, this database is checked for a match. If the top-level window is found, the corresponding handler is called with three arguments:

```
<handler>( Socket, Message, Error )
```

The three arguments are bound as follows: `Socket` contains the handle of the socket that received the event as an atom, `Message` contains the name of the event as an atom, and `Error` contains an error code as an integer; normally this value is zero: any other value indicates a Winsock error.

Starting in version 4.600, **WIN-PROLOG** has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via `winapi/4`.

The full set of Winsock predicates includes `sckhdl/2`, `sclose/1`, `screate/2`, `sdict/2`, `srecv/2`, `ssend/2` and `sstat/2` together with `socket_handler/1` and `socket_handler/2`; see Appendix P for a detailed overview.
**socket_handler/3**

*default windows sockets (winsock) socket handler*

\[
\text{socket_handler}(\text{Socket, Event, Error})
\]

+Socket <atom>
+Event <atom>
+Error <integer>

**Comments**
This provides standard, default processing of Windows Sockets (Winsock) events, specified by the given Socket and Event names and Error code.

**Examples**
By default, all messages to sockets are handled by a predicate, `socket_handler/3`; you can see this by calling `socket_handler/2` with a socket name, even before the socket has been created:

```prolog
?- socket_handler( foo, H ).
```

```
H = socket_handler
```

Now consider the following socket handler:

```prolog
foo( Socket, Event, Error ) :-
    ( write( Socket = Event / Error ),
      nl
    ) ~> user,
    socket_handler( Socket, Event, Error ).
```

Once this code has been compiled, we can assign `foo/3` to become the socket handler for our forthcoming socket, `foo`:

```prolog
?- socket_handler( foo, foo ).
```

```
yes
```

The following call uses `screate/2` to create a client socket that attempts to connect to the LPA website at `www.lpa.co.uk`; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

```prolog
?- screate( foo, `www.lpa.co.uk` ).
```
Assuming the connection has succeeded, momentarily after the above command has completed, our handler will display two events:

?- foo = sck_connect / 0
?- foo = sck_write / 0

In response to the "sck_write" event, we can send a very simple HTTP request using ssend/2:

?- ssend( foo, `GET /index.htm` ).

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "index.htm"; after a moment, we will see confirmation of this event as our program reports the following:

?- foo = sck_read / 0

Now we know we can receive the data with a call to srecv/2:

?- srecv( foo, Text ).

where "..." will depend upon the contents of the file. Another event will be reported, caused by the web server asking us to close the connection:

?- foo = sck_close/ 0

In response to this, we can close the socket with sclose/1, releasing its resources:

?- sclose( foo ).

A final piece of housekeeping is to remove the association of the socket name, "foo", with the handler, "foo/0". We do this with socket_handler/2, passing in either the name, "socket_handler" or the empty list, "[]", which is a shorthand for the same:

?- socket_handler( foo, [] ).

yes

Notes

The `socket_handler/3` predicate provides default processing of socket events, and is attached to all named sockets unless replaced in a call to `socket_handler/2`. The present predicate ignores all events. See `socket_handler/2` for a more detailed discussion.
solution/2

call a goal, returning the number of its solution

solution( Goal, Number )

+Goal <goal>
?Number <integer> or <variable>

Comments
This calls the given Goal, and succeeds or fails depending upon whether the Goal itself succeeds or fails, and whether Number unifies with the number of the solution returned. If Number is an unbound variable, the Goal is allowed to generate additional solutions on backtracking, each time incrementing the integer that is returned as Number; if Number is given, the Goal is forced to backtrack until it reaches the indicated solution, at which point any remaining choice points introduced by the Goal are removed, in order to avoid backtracking into the call on subsequent failure.

Examples
The following command invokes the member/2 predicate, returning the number of each successive solution:

?- solution( member(X,[a,b,c]), S ).
X = a ,
S = 1 ;
X = b ,
S = 2 ;
X = c ,
S = 3 ;
no

The next call deterministically returns solution number "2":

?- solution( member(X,[a,b,c]), 2 ).
X = b

Notes
In many respects, this predicate is similar to call/1: the only difference is that the Number of a solution can be returned or forced. When Number is an unbound variable at the time of call to solution/2, the Goal executes and backtracks normally; where it is given, however,
backtracking is forced one less than a given Number of times, at which point a local call is made to "cut" (!/0) to commit to the solution reached.
sort/2

sort a list into ascending order, removing duplicate terms

\[\text{sort( List1, List2 )}\]

+List1 <list>
?List2 <variable> or <list>

Comments
This sorts the list of terms in \textit{List1} into ascending order according to the \textit{standard ordering} of terms, removes all duplicates, and unifies the result with \textit{List2}.

Examples
The following command sorts the given list of terms into ascending order, removing any duplicates:

\?- sort([the,cat,and,thedog], S).
S = [and,cat,dog,the] <enter>

Notes
Unlike \textit{sort/3} and \textit{keysort/2}, the \textit{sort/2} predicate removes duplicate entries from the sorted list: unfortunately, once removed, data cannot be reinstated, so this "feature" should be treated with caution. The related predicate, \textit{sort/3}, provides a considerably more flexible approach to sorting, including an arbitrary key-based index "path" that allows the user to specify precisely which subterm to use as the sort key on any particular occasion, and without duplicate removal:

\?- sort([3-the,1-quick,4-brown,2-fox], S, [2]).
S = [1 - quick,2 - fox,3 - the,4 - brown] <enter>

\?- sort([3-the,1-quick,4-brown,2-fox], S, [3]).
S = [4 - brown,2 - fox,1 - quick,3 - the] <enter>

See the entry for \textit{sort/3} for further information about sorting, duplicate checking, and other features.
sort a list into ascending order using given key path

\[
\text{sort} ( \text{List1}, \text{List2}, \text{Path} )
\]

\[
\begin{align*}
+\text{List1} & \quad \text{<list>} \\
-\text{List2} & \quad \text{<variable>} \\
+\text{Path} & \quad \text{<list>}
\end{align*}
\]

Comments  
This sorts the list of terms in \text{List1} into ascending order according to the "standard ordering" of terms, using the given \text{Path} to identify the sort key, and unifies the result with \text{List2}.

Examples  
The following command sorts the given list of terms into ascending order, using an empty path ("[]"), and without removing any duplicates:

\[
\text{?- sort} ( [\text{the,cat,and,the,dog}], S, [\text{}]) .
\]

\[
S = [\text{and,cat,dog,the,the}]
\]

Notes  
Unlike \text{sort/2}, the \text{sort/3} predicate does not remove duplicate entries from the sorted list, and unlike \text{keysort/2}, it does not restrict the input list to containing only terms of the form "key-value". Of the three sorting predicates, \text{sort/3} is the most general: in fact, the other two are simply implemented in terms of the present predicate.

The \text{Path} feature of \text{sort/3} is extremely powerful, allowing any arbitrary subterm to be used as the sort key on any particular occasion (see \text{mem/3} for more information about paths):

\[
\text{?- sort} ( [3-\text{the},1-\text{quick},4-\text{brown},2-\text{fox}], S, [2]) .
\]

\[
S = [1 - \text{quick},2 - \text{fox},3 - \text{the},4 - \text{brown}]
\]

\[
\text{?- sort} ( [3-\text{the},1-\text{quick},4-\text{brown},2-\text{fox}], S, [3]) .
\]

\[
S = [4 - \text{brown},2 - \text{fox},1 - \text{quick},3 - \text{the}]
\]

The \text{stir/2} predicate, which stirs lists of arbitrary terms into a randomised order, is effectively the opposite of \text{sort/3}, and can be very useful in a variety of situations, such as shuffling cards in a game program or randomising a sequence of tests. See the entry for \text{stir/2} for further information about shuffling and random sequencing.
source_file/1

get or check the name of a currently loaded program file

source_file( FileSpec )

?FileSpec <file_spec> or <variable>

Comments
This gets or checks the names of currently loaded program files. When FileSpec is an unbound variable, it returns the absolute file name of a currently loaded file; on subsequent failure and backtracking, successive files can be returned. When FileSpec is given as a file specification, a check is made to see whether such a file is currently loaded.

Examples
The following call assumes that two example files, "hanoi.pl" and "lunar.pl", have been loaded; it returns each of their absolute file names in turn on backtracking:

?- source_file( S ).
S = 'c:\pro386w\examples\hanoi.pl';
S = 'c:\pro386w\examples\lunar.pl'

The next call simply checks that the "hanoi" example has been loaded, using a logical file name:

?- source_file( examples(hanoi) ).
yes

Notes
This is one of a family of three related predicates: in spite of their names, source_file/1, source_file/2 and source_file/3 all return information about both source (.PL) and object (.PC) files. If FileSpec does not specify a file extension, each of the "source" and "object" extensions (normally '.PL' or '.PC' respectively) is tested in turn.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
source_file/2

get or check a currently loaded predicate and program file

source_file( Head, FileSpec )

?Head <atom>, <tuple> or <variable>
?FileSpec <file_spec> or <variable>

Comments
This matches a predicate to its program file. The predicate is specified as a Head, comprising a dummy goal which includes the predicate name and arbitrary argument values. When FileSpec is an unbound variable, it returns the absolute file name of a currently loaded file; when it is given as a file specification, a check is made to see whether such a file is currently loaded. If either or both arguments are unbound variables, successive solutions can be found on backtracking.

Examples
The following call assumes that two example files,"hanoi.pl" and "lunar.pl", have been loaded; it returns the name of the file containing the hanoi/1 predicate:

?- source_file( hanoi(_), S ).
    S = 'c:\pro386w\examples\hanoi.pl';
    no

The next call finds the predicates implemented in the "lunar" example, returning successive solutions on backtracking:

?- source_file( P, examples(lunar) ).
    P = lunar(_1,_2,_3,_4); 
    P = lunar(_5,_6,_7,_8,_9,_10); 
    no

Notes
This is one of a family of three related predicates: in spite of their names, source_file/1, source_file/2 and source_file/3 all return information about both source (.PL) and object (.PC) files. If FileSpec does not specify a file extension, each of the "source" and "object" extensions (normally '.PL' or '.PC' respectively) is tested in turn.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple
atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to `file_search_path/2`, and are expanded into atom file names by the `absolute_file_name/3` predicate before being used.

The format of this predicate's first argument is somewhat quirky, since elsewhere, predicates are usually specified as name and arity. For example, consider the output from `current_predicate/1`:

```
?- current_predicate( P ).
P = xrefs / 0 ;
no
```

Further information about current predicates can be found using `def/3`, `defs/2`, `getflg/3` and `pdict/4`. 
**source_file/3**

get or check a predicate, clause count and program file

/source_file( Head, Clauses, FileSpec )/

?Head <atom>, <tuple> or <variable>
?Clauses <integer> or <variable>
?FileSpec <file_spec> or <variable>

Comments This matches a predicate to its program file and the number of clauses that comprise the predicate. The predicate is specified as a *Head*, comprising a dummy goal which includes the predicate name and arbitrary argument values. When *FileSpec* is an unbound variable, it returns the absolute file name of a currently loaded file; when it is given as a file specification, a check is made to see whether such a file is currently loaded; in both cases, the number of clauses that the predicate contains is unified to *Clauses*. If any or all arguments are unbound variables, successive solutions can be found on backtracking.

Examples The following call assumes that two example files,"hanoi.pl" and "lunar.pl", have been loaded; it returns the name of the file containing the hanoi/1 predicate, together with the number of clauses this predicate contains:

```prolog
?- source_file( hanoi(_), C, S ).
C = 1 ,
S = 'c:\pro386w\examples\hanoi.pl' ;
no
```

The next call finds the predicates implemented in the "lunar" example, together with their numbers of constituent clauses, returning successive solutions on backtracking:

```prolog
?- source_file( P, C, examples(lunar) ).
P = lunar(_1,_2,_3,_4) ,
C = 1 ;
P = lunar(_5,_6,_7,_8,_9,_10) ,
C = 1 ;
no
```
Notes

This is one of a family of three related predicates: in spite of their names, `source_file/1`, `source_file/2` and `source_file/3` all return information about both `source` (.PL) and `object` (.PC) files. If `FileSpec` does not specify a file extension, each of the "source" and "object" extensions (normally '.PL' or '.PC' respectively) is tested in turn.

This is one of a series of file handling predicates which make use of "logical" file names: `FileSpec` can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to `file_search_path/2`, and are expanded into atom file names by the `absolute_file_name/3` predicate before being used.

The format of this predicate's first argument is somewhat quirky, since elsewhere, predicates are usually specified as name and arity. For example, consider the output from `current_predicate/1`:

```
?- current_predicate( P ).
P = xrefs / 0 ;

no
```

Further information about current predicates can be found using `def/3`, `defs/2`, `getflg/3` and `pdict/4`. 
**Sprint/1**

*write a quoted term to the current output stream in standard syntax*

```
sprint( Term )

?Term <term>
```

**Comments**
This writes the given `Term` to the current output stream, in quoted Standard syntax.

**Examples**
The following command outputs the given term with quotes where necessary:

```
?- sprint( `String`+'Atom'=Var ), nl.
 (= (+ `String` 'Atom') _1)
Var = _
```

**Notes**
This predicate outputs terms using "quoted" Standard syntax, which means that any strings are delimited by backward quotes (`), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Finally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation:

```
?- sprint( 'hello-M~Jworld' ), nl.
 'hello-M~Jworld'
yes
```

Terms output to a file in quoted syntax are readable provided they are followed by at least one space or control character. This is not provided automatically, and must therefore be output explicitly if readability is required:

```
?- sprint( foo(X) ), nl.
(foo _1)
X = _
```

The `sprint/1` predicate is analogous to the Edinburgh predicate, `writeq/1`. Unlike the latter, however, `sprint/1` forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output in the highly canonical, easily parsable and operator-free "Standard" syntax (see `sprint/2` and `sprint/3` respectively).

For more information about Standard syntax, and how it related to Edinburgh syntax, see Appendix S.
write a quoted term with named variables to current output in standard syntax

\[ \text{sprint}( \text{Term, Vars} ) \]

Comments
This writes the given Term to the current output stream, in quoted Standard syntax, naming any variables listed in Vars with their corresponding names.

Examples
The following command outputs the given term with quotes where necessary, using the given named variable:

\[ \text{?- sprint}( `\text{String}`+`\text{Atom}`=Var, [(`X`,Var)] ), nl. \]
\[ (= (+ `\text{String}` `\text{Atom}`) X) \]
\[ \text{Var} = _ \]

Notes
This predicate outputs terms using "quoted" Standard syntax, which means that any strings are delimited by backward quotes (\`\`), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Finally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see sprint/1).

The ability to preserve or create variable names during input and output is a key feature of WIN-PROLOG: predicates such as sread/2 return the list of variable names used in a term, and these can then be used in predicates like sprint/2:

\[ \text{?- sread( T, V ), sprint( T, V ), nl.} \]
\[ |: (\text{foo VAR1 VAR2}) \]
\[ (\text{foo} \text{ VAR1} \text{ VAR2}) \]
\[ \text{T} = \text{foo}(\_1,\_2) , \]
\[ \text{V} = [(`\text{VAR1}',\_1),(`\text{VAR2}',\_2)] \]

It is this feature that allows WIN-PROLOG to display program listings with the variable names, and even to display the results of queries to the console! A special predicate, vars/2, can be used to identify and name all variables in a term using the sequence, [A..Z], [A1..Z1], [A2..Z2], and so forth, with all single-use variables being given the "anonymous variable" name of "\_" (underscore). This list can be used to name variables neatly during output.
**sprint/3**

write a quoted term with named variables with indented or truncated output in standard syntax

\[
\text{sprint}( \text{Term}, \text{Vars}, \text{Indent} )
\]

Term \text{<term>}
Vars \text{<varlist>}
Indent \text{<integer> in the range \([-1..253]\) or <integer> with other negative value}

**Comments**
This writes the given Term to the current output stream, in quoted Standard syntax, naming any variables listed in Vars with their corresponding names, pretty-printed over multiple lines starting using the given initial Indent, or on a single line with automatic truncation, by specifying a negative Indent other than -1 whose absolute value represents the maximum number of characters to output.

**Examples**
The following command outputs the given term with quotes where necessary, using the given initial Indent:

```prolog
?- sprint( [`String`, [`Atom`|Var]], [], 2 ).
[ `String`
  [ `Atom`
    | _1
  ]
] Var = _
```

By using an "indent" of -1, pretty printing is suppressed:

```prolog
?- sprint( [`String`, [`Atom`|Var]], [], -1 ), nl.
[ `String` [ `Atom`|_1]]
Var = _
```

The following command outputs a term, automatically truncated to just 9 characters:

```prolog
?- sprint( `the quick brown fox`, [], -10 ).
`the quickno
```
Notice that \texttt{sprint/3} has "failed" in this instance, resulting in "no" being displayed by the supervisor: see "Notes" below for an explanation.

\textbf{Notes}

This predicate outputs terms using "quoted" Standard syntax, which means that any strings are delimited by backward quotes (``), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see \texttt{sprint/1}).

The "pretty printing" feature automatically lays out nested lists, conjunction and disjunctions over a series of lines, intending 2 spaces for each additional level of recursion, and placing a final new line after the term. The \textit{Indent} parameter specifies the number additional of 2-space "tabs" to use on each line, to facilitate the use of automatic indentation within user programs that are recursively outputting terms to apply customised layout styles.

Truncated output is a new feature in \texttt{WIN-PROLOG 5.000}, and was introduced to handle the output of long or infinite terms more gracefully than before, allowing a limit to be imposed on the number of characters displayed during any given piece of term output.

The \texttt{sprint/3} truncation feature is designed to fail the predicate if truncation occurs, and to succeed if not, giving calling programs the chance to handle truncation. For example, consider this program:

\begin{verbatim}
trunc( Term, Size ) :-
    Prec is - Size,
    ( \texttt{sprint( Term, [], Prec )} -> true
    ; \texttt{write( `---TRUNCATED!' )} )
).
\end{verbatim}

The following calls show how the failure can be used to display a truncation warning message:

\begin{verbatim}
?- \texttt{trunc( hello_world, 11 ), nl.} \hspace{1cm} <enter>
\texttt{hello_world}
\texttt{yes}

?- \texttt{trunc( hello_world, 10 ), nl.} \hspace{1cm} <enter>
\texttt{hello_world<---TRUNCATED!}
\texttt{yes}
\end{verbatim}
Spy/1

add spypoints from one or more predicates

spy( Prefs )

+Prefs <pred_specs>

Comments
This adds spypoints to the predicates specified in Prefs.

Examples
Assuming a predicate exists, called foo/1, and that it has no currently set spypoint, the following call adds a spypoint:

?- spy( foo/1 ).
# Spypoint placed on predicate foo / 1
yes

A number of predicates can be "spied" in a single call, using either a list or a conjunction:

# Spypoint placed on predicate bar / 2
# Spypoint placed on predicate sux / 3
# Spypoint placed on predicate you / 4
yes

?- spy( (bar/2,sux/3,you/4) ).
yes

Because "spy" is defined as a prefix operator, it is also possible to declare predicates using the following syntax:

?- spy foo/1.
yes

Notes
When the debugging mode is set to "debug" (see debug/0), and a predicate is called for which a spypoint is currently set (a "spied" predicate), the currently selected system debugger is automatically invoked to trace the predicate's execution. The spy/1 predicate is used to add a spypoint to one or more predicates, and nospy/1 is used to remove them once these predicates have been successfully debugged; meanwhile, nospyall/0 can remove spypoints from all predicates at once.
sread/1

read a term from the current input stream in standard syntax

\[ \text{sread}( \text{Term} ) \]

-Term <variable>

Comments
This reads an Standard syntax Term from the current input stream.

Examples
The following command reads a term from the console:

\[
\text{?- sread( T ).} \\
|: (\text{foo VAR1 VAR2}) \\
T = \text{foo(_1,_2)}
\]

Notes
This predicate inputs terms that have been written in "quoted" Standard syntax, which means that any strings are delimited by backward quotes ("), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes (' ) (see sprint/1). Terms must be terminated at least one space or control character.

The sread/1 predicate is more or less analogous to the Edinburgh predicate, read/1. Unlike the latter, however, sprint/1 forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output in the highly canonical, easily parsable and operator-free "Standard" syntax (see sprint/1).

Error handling in the "sread" family is simpler than in read/1: firstly, if the input stream is at end of file, a call to sread/1 or sread/2 simply fails; read/1 returns the atom, "end_of_file", for which programs must explicitly test; secondly, if a syntax error is encountered during input, the "sread" predicates report this directly using the standard error handler, while the read/1 predicate attempts to process the error internally, depending upon the settings of assorted Prolog flags.

For more information about Standard syntax, and how it related to Edinburgh syntax, see Appendix 5.
sread/2

read a term and variables from current input in standard syntax

\[ \text{sread( Term, Vars )} \]

- Term \hspace{1em} <variable>
- Vars \hspace{1em} <variable>

Comments
This reads an Standard syntax Term from the current input stream, returning a list of any variables with their corresponding names in Vars.

Examples
The following command reads a term and variable list:

\[ \text{?- sread( T, V ).} \]
\[ \text{|: (foo VAR1 VAR2)} \]
\[ \text{T = foo(_1,_2)} \]
\[ \text{V = [('VAR1',_1),('VAR2',_2)]} \]

Notes
This predicate inputs terms that have been written in "quoted" Standard syntax, which means that any strings are delimited by backward quotes (\''), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes () (see sprint/1). Terms must be terminated by at least one space or control character.

The ability to preserve or create variable names during input and output is a key feature of \textbf{WIN-PROLOG}: predicates such as sread/2 return the list of variable names used in a term, and these can then be used in predicates like sprint/2:

\[ \text{?- sread( T, V ), sprint( T, V ), nl.} \]
\[ \text{|: (foo VAR1 VAR2)} \]
\[ \text{(foo VAR1 VAR2)} \]
\[ \text{T = foo(_1,_2)} \]
\[ \text{V = [('VAR1',_1),('VAR2',_2)]} \]

It is this feature that allows \textbf{WIN-PROLOG} to display program listings with the variable names, and even to display the results of queries to the console! A special predicate, vars/2, can be used to identify and name all variables in a term using the sequence, [A..Z], [A1..Z1], [A2..Z2], and so forth, with all single-use variables being given the "anonymous variable" name of "\_" (underscore). This list can be used to name variables neatly during output.
srecv/2

receive data from a windows sockets (winsock) tcp/ip socket

\[ \text{srecv( Name, Text )} \]

+Name <atom>
-Text <variable>

Comments

This attempts to receive data from a connection associated with the Windows Sockets (Winsock) socket of the given \text{Name}, returning it as a string bound to \text{Text}. If no text is available on the socket, \text{srecv/2} fails; otherwise it will return one (1) or more bytes of data if the connection is still open, or zero (0) bytes if the connection has been closed.

Examples

The following call uses \text{screate/2} to create a client socket that attempts to connect to the LPA website at \text{www.lpa.co.uk}; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

\[ \text{?- screate( foo, `www.lpa.co.uk` ).} \]
\[ \text{yes} \]

Assuming the connection has succeeded, we can send a very simple HTTP request using \text{ssend/2}:

\[ \text{?- ssend( foo, `GET /index.htm~M~M~J` ).} \]
\[ \text{yes} \]

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "/index.htm"; we can receive this with a call to \text{srecv/2}:

\[ \text{?- srecv( foo, Text ).} \]
\[ \text{Text = `\text{\langle HTML\rangle ... \text{\langle /HTML\rangle~M~J`} \]

where "..." will depend upon the contents of the file. Once the page has been received, we can close the socket with \text{sclose/1}, releasing its resources:

\[ \text{?- sclose( foo ).} \]
\[ \text{yes} \]
Notes

Once a connection has been established on a given socket, `srecv/2` can be called as often as needed, usually in response to a "sck_read" socket event. If the call succeeds, it will return up to about 8kb of data. Successive calls to this predicate should be made until it fails, indicating that no more data currently exists, or until it returns zero (0) bytes, which indicates that the remote end of the connection has been closed.

Starting in version 4.600, **WIN-PROLOG** has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via `winapi/4`.

The full set of Winsock predicates includes `sckhdl/2`, `sclose/1`, `screate/2`, `sdict/2`, `srecv/2`, `ssend/2` and `sstat/2` together with `socket_handler/1` and `socket_handler/2`; see Appendix P for a detailed overview.
ssend/2

send data to a windows sockets (winsock) tcp/ip socket

\[ \text{ssend}( \text{Name}, \text{Text} ) ]

+Name <atom>
+Text <string>

Comments
This attempts to send data, encoded as an 8-bit string in Text, to a connection associated with the Windows Sockets (Winsock) socket of the given Name. If all the text is sent, ssend/2 succeeds; otherwise, it fails.

Examples
The following call uses screate/2 to create a client socket that attempts to connect to the LPA website at www.lpa.co.uk; because a port number is not specified, "80" (the standard HTTP port) is used as a default:

\[- \text{screate}( \text{foo}, `\text{www.lpa.co.uk}` ). \]  
yes

Assuming the connection has succeeded, we can send a very simple HTTP request using ssend/2:

\[- \text{ssend}( \text{foo}, `\text{GET /index.htm~M~J~M~J}` ). \]  
yes

Once this string has been sent, the LPA web server will respond by sending back the contents of its file, "index.htm"; we can receive this with a call to srecv/2:

\[- \text{srecv}( \text{foo}, \text{Text} ). \]  
Text = `\text{`<HTML> ... </HTML>`~M~J}`

where "..." will depend upon the contents of the file. Once the page has been received, we can close the socket with sclose/1, releasing its resources:

\[- \text{sclose}( \text{foo} ). \]  
yes

Notes
Once a connection has been established on a given socket, ssend/2 can be called as often as needed, usually in response to a "sck_write"
socket event. Up to about 8kb of text can be sent in any one call, and successive calls to this predicate should be made until it fails, indicating that no more data can currently be sent. Note that Winsock only supports 8-bit text, so any Unicode characters will need to be encoded, for example as UTF-8, before sending.

Starting in version 4.600, **WIN-PROLOG** has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via `winapi/4`.

The full set of Winsock predicates includes `sckhdl/2`, `sclose/1`, `screate/2`, `sdict/2`, `srecv/2`, `ssend/2` and `sstat/2` together with `socket_handler/1` and `socket_handler/2`; see Appendix P for a detailed overview.
sstat/2

return status of a windows sockets (winsock) tcp/ip socket

\[ sstat( \text{Name}, \text{Status} ) \]

+Name <atom>
-Status <variable>

**Comments**

This returns an integer indicating the current *Status* of the Windows Sockets (Winsock) socket of the given *Name*; this status is bit-encoded to indicate the read, write and other properties of the socket, as shown in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Winsock Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'0001</td>
<td>FD_READ</td>
<td>The socket has data which can be read by srecv/2</td>
</tr>
<tr>
<td>16'0002</td>
<td>FD_WRITE</td>
<td>The socket is able to transmit at least some data written by ssend/2</td>
</tr>
<tr>
<td>16'0008</td>
<td>FD_ACCEPT</td>
<td>The socket has received an incoming connection request, which can be accepted by calling screate/2</td>
</tr>
<tr>
<td>16'0010</td>
<td>FD_CONNECT</td>
<td>The socket is currently or has previously been connected</td>
</tr>
<tr>
<td>16'0020</td>
<td>FD_CLOSE</td>
<td>The socket has been requested to close by the remote connection, suggesting a call to sclose/1</td>
</tr>
<tr>
<td>16'0040</td>
<td>&lt;none&gt;</td>
<td>An error has occurred on the socket, which may not be recoverable, suggesting a call to sclose/1</td>
</tr>
<tr>
<td>16'8000</td>
<td>&lt;none&gt;</td>
<td>This is a listening socket, which can only be used to wait for incoming connection requests</td>
</tr>
</tbody>
</table>

**Examples**

The following call uses screate/2 to create a client socket that attempts to connect to the LPA website at www.lpa.co.uk; because a port number is not specified, “80” (the standard HTTP port) is used as a default:

\[ ?- \text{screate( } \text{foo, } `\text{www.lpa.co.uk}` \text{) }, \text{sstat( } \text{foo, } S \text{) } \text{.} \]

\[ S = 0 \]

Because screate/2 is non-blocking, the predicate has returned immediately, before the connection has been made: this is shown by the value returned by sstat/2. Calling the latter predicate again after a few moments will reveal whether or not the connection has been made:
This time the return value contains two bits, corresponding to "FD_CONNECT" and "FD_WRITE", indicating that the socket is connected and may be written to.

Assuming the connection has succeeded, we can send a very simple HTTP request using ssend/2:

yes

Once this string has been sent, we can check the socket status again:

?- sstat( foo, S ).
S = 19

This shows that socket is still connected, and can still be written to, but in addition, a third bit, corresponding to "FD_READ", indicating that some data is now ready for reading from the socket: this because the LPA web server will have responded by sending back the contents of its file, "/index.htm", we can receive this with a call to srecv/2:

?- srecv( foo, Text ).
Text = `&lt;HTML&gt; ... &lt;/HTML&gt;~M~J`

where "..." will depend upon the contents of the file. Let's check the status one last time:

?- sstat( foo, S ).
S = 51

Yet another bit has been added, this time corresponding to "FD_CLOSE", indicating that the LPA web server has requested that we close the connection. We can now do this, with sclose/1, closing the socket and releasing its resources:

?- sclose( foo ).
yes

Notes

Starting in version 4.600, WIN-PROLOG has built-in support for Windows Sockets (Winsock), providing direct and easy access to TCP/IP network programming. The socket predicates have been designed to work in the "Asynchronous" mode, which means that they are "non-
blocking", and generate messages when ready to process data. They are also set up exclusively for connection-oriented applications; datagram sockets can still be used, but will require direct Winsock calls via \texttt{winapi/4}.

The \texttt{sstat/2} predicate can be used in asynchronous code to check the ongoing current state of a socket, but it can also be used to implement blocking calls simply by calling it repeatedly until a desired socket state is achieved. For example, the following program will create a socket connection, returning only once it has been completed:

\begin{verbatim}
create_socket_blocking( Socket, Host ) :-
   screate( Socket, Host ),
   ( repeat,
       sstat( Socket, Status ),
       Status > 0
   -> ( 16 is Status \(\setminus\) 16
       -> write( `Connected OK~M~J` )
       ;
       sclose( Socket ),
       write( `Cannot Connect Socket~M~J` ),
       fail
   )
).
\end{verbatim}

The full set of Winsock predicates includes \texttt{sckhdl/2}, \texttt{sclose/1}, \texttt{screate/2}, \texttt{sdict/2}, \texttt{srecv/2}, \texttt{ssend/2} and \texttt{sstat/2} together with \texttt{socket_handler/1} and \texttt{socket_handler/2}; see Appendix P for a detailed overview.
set or get a file date and time stamp

\[
\text{\texttt{stamp( File, Stamp )}}
\]

+File <file_name>
?Stamp <variable> or <date_time>

Comments
This sets or gets the date and time Stamp ("timestamp") for the named File; if Stamp is an unbound variable, the existing last-written timestamp is returned; if Stamp is a date_time structure, this is used to set all three timestamps for the given File.

Examples
The following call picks up the current date and time, and uses it to set the timestamp of the file, "foo" (this example assumes that the date is 01-APR-1999, and the time is 10:08:42):

\[
?\text{- time( 1, T ), stamp( foo, T ).} \\
T = (145822,36522000)
\]

The next call simply picks up the time stamp of one file, "bar", and applies it to another, "sux" (this example assumes that the former file has a timestamp of 01-APR-1999, 10:08:42):

\[
?\text{- stamp( bar, T ), stamp( sux, T ).} \\
T = (145822,36522000)
\]

Notes
This is one of a series of file handling and operating system interface predicates which are implemented directly in the WIN-PROLOG "kernel", a low-level, assembler-coded part of the system. At this level, all file names are handled directly as Prolog atoms: there is no support for "logical" file names. Please see the "notes" section in the entry for absolute_file_name/2 for further discussion about file names.

This is also one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date_time" data type provides a uniform structure for handling, processing and storing dates and times throughout WIN-PROLOG. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, time/4 and time/5, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, time/2, is used to return current local or elapsed time.
display a statistics report about the current session

**statistics**

**Comments**
This writes a formatted report containing information about memory usage, time utilisation and garbage collection, to the current output device.

**Examples**
The following call was made early in a Prolog session, reporting initial statistics shortly after loading:

```prolog
?- statistics.
```

<table>
<thead>
<tr>
<th>Memory Statistics</th>
<th>Free Bytes</th>
<th>Total Bytes</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backtrack Space</td>
<td>65366</td>
<td>65536</td>
<td>(100%)</td>
</tr>
<tr>
<td>Local Space</td>
<td>64931</td>
<td>65536</td>
<td>(99%)</td>
</tr>
<tr>
<td>Reset Space</td>
<td>65536</td>
<td>65536</td>
<td>(100%)</td>
</tr>
<tr>
<td>Heap Space</td>
<td>261840</td>
<td>262140</td>
<td>(100%)</td>
</tr>
<tr>
<td>Text Space</td>
<td>1929600</td>
<td>2097152</td>
<td>(92%)</td>
</tr>
<tr>
<td>Program Space</td>
<td>7151744</td>
<td>8388480</td>
<td>(85%)</td>
</tr>
<tr>
<td>Stack Space</td>
<td>65496</td>
<td>65536</td>
<td>(100%)</td>
</tr>
<tr>
<td>Input Space</td>
<td>262144</td>
<td>262144</td>
<td>(100%)</td>
</tr>
<tr>
<td>Output Space</td>
<td>262144</td>
<td>262144</td>
<td>(100%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timer Statistics</th>
<th>Seconds</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Elapsed Time</td>
<td>1787.025</td>
<td>(100%)</td>
</tr>
<tr>
<td>Active Processing</td>
<td>13.493</td>
<td>(1%)</td>
</tr>
<tr>
<td>Waiting for Input</td>
<td>1773.312</td>
<td>(99%)</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>0.220</td>
<td>(0%)</td>
</tr>
</tbody>
</table>

**Notes**
The `statistics/0` predicate provides a convenient method of displaying a comprehensive session report, while `statistics/2` allows individual statistics to be obtained programmatically. The garbage collector is always invoked by the former predicate, and also by the latter when...
it is being used to obtain memory-related statistics. See \texttt{free/9}, \texttt{total/9} and \texttt{stats/4} for information about how to obtain statistics more directly, without invoking the garbage collector, and also see \texttt{gc/1} for information about calling the garbage collector itself.
get or check individual statistics about the current session

\[
\text{statistics}( \text{Flag}, \text{Value} )
\]

- \text{Flag} \quad \text{<atom> or <variable>}
- \text{Value} \quad \text{<integer>, <list> or <variable>}

Comments

This returns specific statistics about memory usage and time utilisation. When \text{Flag} is an unbound variable, it is given one of the names in the table below, and \text{Value} is unified with the appropriate statistical information: on subsequent failure and backtracking, successive statistics are returned. When \text{Flag} is an atom, the specifically requested statistic is obtained deterministically, and unified with \text{Value}. With one exception ("runtime", see below), the \text{Value} is returned as an integer number of bytes of memory. There are nine memory areas that can be queried, and for each one, it is possible to obtain its "free" or "total" space, using a \text{Flag} whose name is one of the following prefixes, followed by the suffix ".free" or ".total":

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>backtrack</td>
<td>returns backtrack stack free or total space</td>
</tr>
<tr>
<td>local</td>
<td>returns local stack free or total space</td>
</tr>
<tr>
<td>reset</td>
<td>returns reset stack free or total space</td>
</tr>
<tr>
<td>heap</td>
<td>returns term heap free or total space</td>
</tr>
<tr>
<td>text</td>
<td>returns text heap free or total space</td>
</tr>
<tr>
<td>program</td>
<td>returns program heap free or total space</td>
</tr>
<tr>
<td>stack</td>
<td>returns system stack free or total space</td>
</tr>
<tr>
<td>input</td>
<td>returns input string free or total space</td>
</tr>
<tr>
<td>output</td>
<td>returns output string free or total space</td>
</tr>
</tbody>
</table>

The remaining flag, "runtime", returns a list containing two integers: the first gives the total number of milliseconds that have elapsed since the current \text{WIN-PROLOG} session started, and the second gives the number of milliseconds that have elapsed since the previous time that "runtime" was reported.

Examples

The following call returns the free text space, in bytes:

\[
? - \text{statistics}( \text{text_free}, \text{V} ).
\]
\[ V = 1929600 \]

In the next call, Flag is unbound, so successive statistics are returned on backtracking (this example only includes the first three of nineteen possible solutions):

```prolog
?- statistics( F, V ).
F = runtime ,
V = [26877,1024] ;

F = backtrack_free ,
V = 65366 ;

F = local_free ,
V = 65076
```

**Notes**

The `statistics/0` predicate provides a convenient method of displaying a comprehensive session report, while `statistics/2` allows individual statistics to be obtained programmatically. The garbage collector is always invoked by the former predicate, and also by the latter when it is being used to obtain memory-related statistics. See `free/9`, `total/9` and `stats/4` for information about how to obtain statistics more directly, without invoking the garbage collector, and also see `gc/1` for information about calling the garbage collector itself.
**stats/4**

*get timer and garbage collection statistics*

```prolog
stats( Total, Idle, Garbage, Count )
```

- `Total` <variable>
- `Idle` <variable>
- `Garbage` <variable>
- `Count` <variable>

**Comments**

This returns the `Total` elapsed time, total `Idle` time and total `Garbage` collection time for the current Prolog session, together with a `Count` of calls to the garbage collector. All three times are returned as "date_time" structures (see below).

**Examples**

The following call was made early in a Prolog session, returning initial statistics shortly after loading:

```prolog
?- stats( T, I, G, C ).
```

```
T = (0,26877) ,
I = (0,5539) ,
G = (0,182) ,
C = 14
```

**Notes**

Together with `free/9` and `total/9`, the `stats/4` predicate provides raw information about the current **WIN-PROLOG** session: these three predicates are used by `statistics/0` and `statistics/2`, both of which also call the garbage collector whenever reporting information about memory (see `gc/1` for further information about calling the garbage collector).

This is one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date_time" data type provides a uniform structure for handling, processing and storing dates and times throughout **WIN-PROLOG**. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, `time/4` and `time/5`, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, `time/2`, is used to return current local or elapsed time.
get or set the existence of window status bars

\[status\_bars(\text{Status})\]

?Status <variable> or <integer> in the range \([0..1]\)

**Comments**

This gets or sets the existence \(\text{Status}\) of window status bars on text windows. When specified as a variable, \(\text{Status}\) is bound to an integer giving the current setting; when specified as an integer, it defines the new \(\text{Status}\), causing status bars to be shown or hidden, as follows:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hide window status bars</td>
</tr>
<tr>
<td>1</td>
<td>show window status bars</td>
</tr>
</tbody>
</table>

**Examples**

The following call returns the current status bar status:

\[?- status\_bars(\text{Status}).\]

\(\text{Status} = 1\)

The next call turns off windows status bars:

\[?- status\_bars(0).\]

yes

**Notes**

"Status Bars" in WIN-PROLOG provide immediate feedback about the status of a program window, including its window type, saved/compiled/optimised status, and row, column, length and selection size settings. Although windows status bars take up just one line of space, and have a minimal performance overhead, it might be desirable to disable them at times.

The \(status\_bars/1\) predicate provides the required control, and can be used to show or hide windows status bars as often as necessary.
stir/2

stir a list into random order

\texttt{stir( List1, List2 )}

+List1 \hspace{1cm} \texttt{<list>}
-List2 \hspace{1cm} \texttt{<variable>}

Comments

This shuffles the list of terms in \texttt{List1} into a randomised order, returning the result in \texttt{List2}.

Examples

The following command sorts the given list of terms into a randomised order:

\texttt{?- stir( [0,1,2,3,4,5,6,7,8,9], R ).}

\texttt{R = [8,1,6,0,2,3,7,4,5,9]}

Notes

This predicate lets you thoroughly randomise any list of terms, using the Fisher/Yates Shuffle Algorithm, powered by the 1185-bit M255 PRANG (Marsaglia Zaman Pseudo Random Number Generator, with Steel Comb Filtering). This is the same number generator as used in the \texttt{decode/2} and \texttt{encode/3} encryption predicates, and also the "@" function in the \texttt{is/2}, \texttt{fpn/2} and \texttt{rpn/2} predicates.

For lists of up to 192 elements, every possible permutation is equally likely to be generated. For larger lists, only a subset of possible permutations can be generated. This is not a software defect; rather a result of the properties of the Factorial function: \texttt{fact(192)} is smaller than \texttt{2^1185}, but \texttt{fact(193)} and beyond is bigger, meaning there are more potential permutations of the list than there are numbers in the pseudo random sequence!

The \texttt{stir/2} predicate is effectively the opposite of \texttt{sort/3}, which sorts lists of arbitrary terms into ascending order, and can be very useful in a variety of situations, such as shuffling cards in a game program or randomising a sequence of tests. See the entry for \texttt{sort/3} for further information about sorting, duplicate checking, and other features.
**stream_position/2**

*get the current position of the specified stream*

```prolog
stream_position( Stream, Position )
```

**+Stream**  
<file_spec>  

**-Position**  
<variable>

**Comments**  
This is used to return the current *Position* in the given *Stream*.

**Examples**  
The following command opens the examples file, "hanoi.pl", for read access:

```prolog
?- open( examples('hanoi.pl'), read ).
```

```
yes
```

The next command reads the first term from this file:

```prolog
?- see( examples('hanoi.pl') ), read( T ).
```

```
T = (hanoi(_1) :- hanoi(_1,'LEFT','RIGHT','MIDDLE'))
```

This command returns the resulting file position:

```prolog
?- stream_position( examples('hanoi.pl'), P ).
```

```
P = 1841
```

Finally, for tidiness, the file should be closed:

```prolog
?- seen.
```

```
yes
```

**Notes**  
The *stream_position/2* and *stream_position/3* predicates are quirky, being the only input/output predicates that require the stream to be specified, rather than working on current input or output streams. As a result, they are unable to perform pointer manipulation in *WIN-PROLOG* memory files or strings: see *inpos/1* and *outpos/1* for information about how to adjust the position in the current input or output stream respectively.
This is one of a series of file handling predicates which make use of "logical" file names: Stream can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
**stream_position/3**

*set and get the current position of the specified stream*

```
stream_position( Stream, Position, New )
```

+Stream <file_spec>
-Position <variable>
+New <integer>

**Comments**

This is used to return the current *Position* in the given *Stream*, while setting a *New* position.

**Examples**

The following command opens the examples file, "hanoi.pl", for read access:

```
?- open( examples('hanoi.pl'), read ).
yes
```

The next command reads the first term from this file:

```
?- see( examples('hanoi.pl') ), read( T ).
T = (hanoi(_1) :- hanoi(_1,'LEFT','RIGHT','MIDDLE'))
```

This command returns the resulting file position, while resetting it to zero ("0"):

```
?- stream_position( examples('hanoi.pl'), P, 0 ).
P = 1841
```

We can see the file position has been rewound to zero by reading from it once more, before closing it for tidiness:

```
?- read( T ), seen.
T = (hanoi(_2) :- hanoi(_2,'LEFT','RIGHT','MIDDLE'))
```

**Notes**

The `stream_position/2` and `stream_position/3` predicates are quirky, being the only input/output predicates that require the stream to be specified, rather than working on current input or output streams. As a result, they are unable to perform pointer manipulation in WIN-PROLOG memory files or strings: see `inpos/1` and `outpos/1` for information about how to adjust the position in the current input or output stream respectively.
This is one of a series of file handling predicates which make use of "logical" file names: see \texttt{absolute_file_name/3} for further information.
string/1

test whether a term is a string

    string( Term )

    ?Term <term>

Comments This succeeds if Term is bound to a string.

Examples The following calls test various cases:

?- string( 123 ).
    no

?- string( [1,2,3] ).
    no

?- string( foo ).
    no

?- string( `bar` ).
    yes

?- string( X ).
    no

Notes This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments.

The "string" is a special WIN-PROLOG data type which can contain an effectively unlimited quantity of text in a single Prolog term, and can even be used as a virtual input and output device. The text is stored in a compact form, as with atoms, but unlike the latter data type, each instance of a string contains a fresh copy of the text.
**string_chars/2**

*convert between a string and a character list*

```prolog
string_chars( String, Chars )
```

?-string_chars(`foo`, L).
L = [102,111,111]

This call fills in the missing characters in a partially instantiated character list:

?-string_chars(`brian`, [98,X,105|Y]).
X = 114,
Y = [97,110]

A string can be created from a fully instantiated list:

?-string_chars(S, [104,101,108,108,111]).
S = `hello`

Even if the list contains the characters of some other term, such as a number, it is a string which is returned:

?-string_chars(S, "123.45").
S = `123.45`

**Comments**

If *String* is instantiated to a string, then *Chars* is unified with a list consisting of integers representing the character codes that make up the string; otherwise, if *String* is an unbound variable and *Chars* is a list of character codes, a new string is created and bound to *String*.

**Examples**

The following converts a string into a list of character codes:

?-string_chars(`foo`, L).
L = [102,111,111]

This call fills in the missing characters in a partially instantiated character list:

?-string_chars(`brian`, [98,X,105|Y]).
X = 114,
Y = [97,110]

A string can be created from a fully instantiated list:

?-string_chars(S, [104,101,108,108,111]).
S = `hello`

Even if the list contains the characters of some other term, such as a number, it is a string which is returned:

?-string_chars(S, "123.45").
S = `123.45`

**Notes**

In all cases where the *String* is given, the *Chars* list may be as complete or incomplete as required; if *String* is a variable, then the *Chars* list must be fully instantiated: this predicate cannot generate alternative solutions based on partial information.
The "string" is a special WIN-PROLOG data type which can contain an effectively unlimited quantity of text in a single Prolog term, and can even be used as a virtual input and output device (see input/1, output/1, <~/2 and ~>2).
strutf/3

*convert between binary and unicode strings*

\[\text{strutf}( \text{Binary, Unicode, Code} )\]

?Binary \quad <\text{string}> \text{ or } <\text{variable}>

?Unicode \quad <\text{string}> \text{ or } <\text{variable}>

+Code \quad <\text{integer}> \text{ in the range } [-5..5]

**Comments**

This converts a \textit{Binary} string into its \textit{Unicode} equivalent, or vice versa; when \textit{Binary} is bound to a string, and \textit{Unicode} is unbound, the conversion is from the \textit{WIN-PROLOG} binary format into the appropriate \textit{Unicode} encoding; when \textit{Binary} is unbound, and \textit{Unicode} is bound to a string, the conversion is from the appropriate \textit{Unicode} encoding into the \textit{WIN-PROLOG} binary format. The \textit{Unicode} encoding is specified by \textit{Code}, which determines which of eleven character encodings to use for the conversion, as defined in the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>32-bit integer raw-32le format</td>
</tr>
<tr>
<td>-4</td>
<td>16-bit integer raw-16le format</td>
</tr>
<tr>
<td>-3</td>
<td>unicode utf-32le format</td>
</tr>
<tr>
<td>-2</td>
<td>unicode utf-16le format</td>
</tr>
<tr>
<td>-1</td>
<td>7-bit ascii utf-8 format</td>
</tr>
<tr>
<td>0</td>
<td>8-bit iso/iec 8859-1 format</td>
</tr>
<tr>
<td>1</td>
<td>unicode utf-8 format</td>
</tr>
<tr>
<td>2</td>
<td>unicode utf-16be format</td>
</tr>
<tr>
<td>3</td>
<td>unicode utf-32be format</td>
</tr>
<tr>
<td>4</td>
<td>16-bit integer raw-16be format</td>
</tr>
<tr>
<td>5</td>
<td>32-bit integer raw-32be format</td>
</tr>
</tbody>
</table>

See Appendix L for more information about character sets, encodings and formats.

**Examples**

The following call converts a binary string containing a "™" (Trademark) symbol into the UTF-8 encoding:

\[\text{?- strutf}( \text{'This is a TM: ™'}, \text{UTF8, 1} ).\]
UTF8 = 'This is a TM: â?¢'

Here is a similar call, which this time returns the UTF-16LE encoding for the same string:

?- strutf('This is a TM: ™', UTF16LE, -2).
UTF16LE = 'T~@h~@i~@s~@ ~@i~@s~@ ~@a~@ ~@T~@M~@:~@ ~@''

Notes

While WIN-PROLOG handles arbitrary Unicode data sources automatically and transparently, through its fcreate/5 and open/3 predicates, using any of the encodings listed above, there are still times when it may be necessary to perform explicit conversions. One such example might be when implementing Windows Sockets (Winsock) code, such as a web server or agent, where data may be transmitted in UTF-8 or any other of the standard Unicode formats. Without strutf/3, the only way to interpret such data correctly would be to write it to a file that had been opened in 8-bit ISO/IEC 8859-1 format, and then reopen the file in the appropriate Unicode encoding, before reading the data back; the reverse process would have to be performed when preparing data to send back across the network. The strutf/3 predicate obviates the need for such time- and resource-hungry procedures, working as it does entirely within the WIN-PROLOG text heap.
stuff/4

compress data from current input to current output

stuff( Wanted, Size, Raw, Comp )

+Wanted <integer>
+Size <integer> in the range [0..4]
-Raw <variable>
-Comp <variable>

Comments
This copies up to the number of characters Wanted, from the current input stream to the current output stream, compressing them along the way, using the given buffer Size. It terminates when the given number of characters has been processed, or end of file is encountered in the input stream. The raw and compressed character counts are returned in Raw and Comp respectively.

Examples
The following simple programs can be used to create or unpack compressed archives of multiple files:

stuff_archive( Archive, Files ) :-
    open( Archive, write ),
    tell( Archive ),
    forall( member( File, Files ),
        { writeq( File ),
          write( ' . ' ),
          see( File ),
          stuff( -1, 2, _, _ ),
          seen
        } ),
    told.

fluff_archive( Archive ) :-
    see( Archive ),
    repeat,
    read( File ),
    tell( File ),
    fluff( _, _, _ ),
Once compiled, this program can be used to create a stuffed (compressed) archive of multiple files; the following call, for example, will create an archive called "foo" which includes compressed copies of two other files, "bar" and "sux":

?- stuff_archive( foo, [bar,sux] ).
yes

These files can be recovered from the archive on a subsequent occasion with the call:

?- fluff_archive( foo ).
yes

Notes

The LZSS algorithm, as used to compress and decompress data in the stuff/4 and fluff/3 predicates respectively, uses a "sliding window" to maintain a memory of recently processed data, while a "look-ahead" 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 optimum setting for whichever type of data you want to compress. The possible values of Size and their meanings are listed in the following table:

<table>
<thead>
<tr>
<th>Size</th>
<th>Bit Settings</th>
<th>Window</th>
<th>Look-ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9/7</td>
<td>512</td>
<td>128</td>
</tr>
<tr>
<td>1</td>
<td>10/6</td>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>11/5</td>
<td>2048</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>12/4</td>
<td>4096</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>13/3</td>
<td>8192</td>
<td>8</td>
</tr>
</tbody>
</table>

For highly repetitive files, such as Windows NT logfiles, the smallest window (Size="0") will probably give the best compression, with a theoretical maximum ratio of 64:1; this relies on repeating elements in the file being within 512 bytes of each other. For sparsely repetitive files, lists of names and addresses, a larger window will help; with the largest window (Size="4"), repeating elements can be as far as 8192 bytes apart, although the maximum compression ratio is reduced to 4:1. For most data streams, a Size of 2 or 3 will give the best compression.
**style_check/1**

*enable the specified compile-time style checking*

```
style_check( Type )
```

+Type <atom>

**Comments**

This enables the specified Type of compile-time style checking, as shown in the following table:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>single_var</td>
<td>named variable occurs only once in a given clause</td>
</tr>
<tr>
<td>discontinuous</td>
<td>clauses for a given relation are not contiguous</td>
</tr>
<tr>
<td>multiple</td>
<td>predicate is defined in more than one file</td>
</tr>
<tr>
<td>all</td>
<td>all of the above</td>
</tr>
</tbody>
</table>

**Examples**

The following call turns on all style checking:

```
?- style_check( all ).
yes
```

**Notes**

In general, it is recommended to use style checking when compiling programs: the most common Prolog programming errors are the result of mis-spelt variable names, and when set, the `single_var` style check can usually pick this up at compile time.
**sublist/2**

get or check an ordered sublist of a list

```
sublist( Term, List )
```

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

**Comments**
This predicate succeeds when its first argument is bound to a `Term` which is an ordered sublist the `List` bound to the second argument. Either argument latter may be a fully or partially instantiated list, or simply a variable; `sublist/2` can backtrack to generate alternative solutions where appropriate.

**Examples**
The following command extracts each of the elements of a given list in turn:

```
?- sublist( [One,Two], [black,and,white] ).
```

One = black ,
Two = and ;

One = black ,
Two = white ;

One = and ,
Two = white ;

no

**Notes**
The `sublist/2` predicate is similar to `member/2`, which is a classic Prolog program, and is widely used for extracting elements from lists; however, `sublist/2` is more powerful, enabling single, paired or greater collections of elements to be extracted from a list, or simply checking for their ordered presence.
**subsumes_chk/2**

*test whether a term subsumes another*

```
subsumes_chk( Term, Instance )
```

**Comments**

This is used to check whether one term is a specific *Instance* of another given *Term*.

**Examples**

The following call succeeds, because the second term, "foo(123)" is a specific instance of the first term, "foo(A)":

```
?- subsumes_chk( foo(A), foo(123) ).
A = _
```

Conversely, the next call fails, because the same is not true when the two terms are reversed:

```
?- subsumes_chk( foo(123), foo(A) ).
no
```

**Notes**

The `subsumes_chk/2` predicate is slightly unusual, in that unlike most other Prolog predicates, it is not designed to find matches for a given variable; neither does it bind variables in either term. Contrast this behaviour with that of standard unification, which can match terms in both directions:

```
?- foo(A) = foo(123).
A = 123
```

```
?- foo(123) = foo(A).
A = 123
```

The process of "subsumes checking" is useful in theorem provers, and in advanced unification routines where it is desired to handle the "most general" cases of terms, without the risk of inadvertently unifying with other terms.
switch/2

set or get the value of a command line switch

\[
\text{switch}( \text{Switch, Value} )
\]

+Switch <atom>

?Value <integer> or <variable>

Comments
This sets or gets the Value of the given command line Switch; where Value is an unbound variable, the existing switch setting is returned; otherwise, where Value is an integer, this is stored in the appropriate switch register.

Examples
Supposing that \text{WIN-PROLOG} had been started with the switch, "/H1234", the following call would return this value:

\[
\text{?- switch( h, S ).}
\]

\[
S = 1234
\]

The following call stores a new value in the same switch:

\[
\text{?- switch( h, 81 ).}
\]

\[
\text{yes}
\]

Notes
There are 26 command line switch registers, named "a" through "z", or "A" through "Z": switches are not case sensitive. Nine of these are used to set the sizes of \text{WIN-PROLOG}'s memory areas at startup, and although they can subsequently be set to new values (see the second example above), these have no affect on memory allocation: these switches are only read once by the system, very early in its boot process.

With the exception of the "D" switch, which modifies the behaviour of dynamic code, any of the switches can be used for the efficient run-time storage of named global integer variables. The 15 or so switches not used at startup for setting memory buffer sizes can also be used by applications for their own purposes. Any switch not present either on the command line or in the \text{.INI file} has a default setting of "-1", and any switch present, but without a numerical parameter, has a setting of "0"; otherwise, switches are set to whatever unsigned integer parameter follows the switch on the command line or in the .INI file. See Appendix B for more information about \text{WIN-PROLOG}'s command line switches.
swrite/1

write an unquoted term to the current output stream in standard syntax

swrite( Term )

?Term <term>

Comments
This writes the given Term to the current output stream, in unquoted Standard syntax.

Examples
The following command outputs the given term without quotes, so that atoms and strings appear the same:

?- swrite( 'String'+Atom=Var ), nl.
(= (+ String Atom) _1)
Var = _

Notes
This predicate outputs terms using "unquoted" Standard syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally:

?- swrite( 'hello~M~Jworld' ), nl.
hello
world
yes

Terms output to a file in unquoted syntax are not generally readable, even if they are followed by at least one space or control character, because unquoted strings and atoms can be confused with each other and even with variables, or may contain characters which give rise to syntax errors.

The swrite/1 predicate is analogous to the Edinburgh predicate, write/1. Unlike the latter, however, swrite/1 forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output in the highly canonical, easily parsable and operator-free "Standard" syntax (see swrite/2, sprint/2 and sprint/3 respectively).

For more information about Standard syntax, and how it related to Edinburgh syntax, see Appendix S.
**swrite/2**

write an unquoted term with named variables to output in standard syntax

```
swrite( Term, Vars )
```

```
?Term <term>
+Vars <varlist>
```

**Comments**

This writes the given `Term` to the current output stream, in unquoted Standard syntax, naming any variables listed in `Vars` with their corresponding names.

**Examples**

The following command outputs the given term without quotes, so that atoms and strings appear the same, using the given named variable:

```
?- swrite( `String`+'Atom'=Var, [('X',Var)] ), nl.
(= (+ String Atom) X)
Var = _
```

**Notes**

This predicate outputs terms using "unquoted" Standard syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally (see `swrite/1`).

The ability to preserve or create variable names during input and output is a key feature of WIN-PROLOG: predicates such as `sread/2` return the list of variable names used in a term, and these can then be used in predicates like `swrite/2`:

```
?- sread( T, V ), sprint( T, V ), nl.
|: (foo VAR1 VAR2)
(foo VAR1 VAR2)
T = foo(_1,_2) ,
V = [('VAR1',_1),('VAR2',_2)]
```

It is this feature that allows WIN-PROLOG to display program listings with the variable names, and even to display the results of queries to the console! A special predicate, `vars/2`, can be used to identify and name all variables in a term using the sequence, ['A..Z'], ['A1..21'], ['A2..22'], and so forth, with all single-use variables being given the "anonymous variable" name of "_" (underscore). This list can be used to name variables neatly during output.
sysops/0

reset the definitions of all system-declared operators

sysops

Comments

This simply resets the definitions of all system-declared operators, ensuring that standard Prolog source files can be read correctly, and without syntax errors.

Examples

The following call to op/3 removes the definition of "*" as an infix operator:

?- op( 0, yfx, * ).
yes

Once this has been done, the following command can no longer be read, and will generate a syntax error:

?- X is 3 * 27 .
! ----------------------------------------
! Error 42 : Syntax Error
! Goal     : ered(_1,_2)

This (and any other) system operators can be restored easily:

?- sysops.
yes

Now the command that previously gave an error will work:

?- X is 3 * 27 .
X = 81

Notes

Operators are critical components of Prolog syntax, and define how terms are assembled internally during calls to read/1, and how they are output by write/1, writeq/1 and so on. Because operators can be redefined (see op/3), it is all too easy to get Prolog to a point where it can no longer read standard Prolog source files: sysops/0 reinstates the system operators, restoring the functionality of term input/output.
**system_menu/3**

**invoke a programming environment menu function**

```
system_menu( Window, Menu, Function )
```

?Window     <window_handle> or <variable>
+Menu       <atom>
+Function   <atom> or <tuple>

**Comments**
This gives direct programmable access to the functions of the **WIN-PROLOG** development environment, calling a named **Menu** and **Function** with respect to a given **Window**. Certain functions work irrespective of any particular window, and on these occasions **Window** can be specified as any term, although an unbound variable is preferable. The **Function** may simply be a menu item name, or a compound term including the name and some or all of its parameters: providing such information (for example, the name of a file to open) bypasses dialogs that would otherwise be shown.

**Examples**
The following command invokes the "File/Open" dialog, allowing the user to choose one or more files to open; its success is dependent upon whether the user confirms or cancels the dialog:

```
?- system_menu( X, file, open ).
X = _
```

The next command directly opens a program window for the example file, "lunar.pl", without creating any dialogs:

```
?- system_menu( X, file, open(examples(hanoi)) ).
X = _
```

**Notes**
The **system_menu/3** predicate provides powerful, direct programmable access to the **WIN-PROLOG** development environment, and allows routine operations, for example, the loading and/or compiling of a particular set of program files, to be automated; in fact, "project files" are little more than an automatically generated set of calls to this predicate. See Appendix F for the full list of system menu functions that are accessible from this predicate.
**tab/1**

*evaluate and write a number of spaces to current output*

```prolog
tab( Expression )
+Expression <expr>
```

**Comments**
This evaluates the given *Expression* by calling *is/2*, and writes the given number of spaces (character code 20h) to the current output stream; an error is generated if the *Expression* does not evaluate to an integer in the range [0..8192].

**Examples**
In most cases, this predicate is used to output a number of spaces that has already been computed, for example:

```prolog
?- tab( 8 ), write( hello ), nl.
   hello yes
```

The following call illustrates the fact that *tab/1* performs arithmetic, and computes an expression before outputting the resulting integer number of spaces:

```prolog
?- tab( 2 * 4 ), write( hello ), nl.
   hello yes
```

**Notes**
The *tab/1* predicate is useful for laying out tables and other columnar text files, although this is most useful when fixed-width fonts (such as the "Prolog fixed font") are in use. This predicate works by computing the given expression and then calling the formatted write predicate, *fwrite/4*, to output an empty string (```) with a field width corresponding to the result. Although this approach makes *tab/1* very fast, it limits the number of space characters output in a single call to 8192.
tell/1

set the current output stream

\[ \text{tell( Stream )} \]

+Stream \quad <\text{file_spec}>

Comments

This sets the current output Stream, which may be a disk file or the special atom, "user", the latter which refers to the console output device. If Stream refers to a disk file which is not yet open, it is automatically opened in "write" mode (see open/2); if the file was already open, output continues from the point immediately after the previously-written character.

Examples

The following call will create (if not already present) and open (if not already open) a file called "foo" in the Prolog home directory, write the term, "hello", to the file, and finally close it and reset the output stream to the console (user) device:

\[ ?- \text{tell( prolog(foo) ), write( `hello.` ), nl, told.} \]

Notes

Three related predicates, \text{tell/1, telling/1 and told/0}, support high-level control over the output stream, and provide for the easy handling of sequential access files: however, it does not have direct access to special numerically named devices or to memory files. Another predicate, \text{input/1}, provides low-level control over all types of WIN-PROLOG input stream.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
telling/1

get the current output stream

    telling( Stream )

?Stream <file_spec> or <variable>

Comments

This gets the current output Stream, which may be a disk file or the special atom, "user", the latter which refers to the console output device.

Examples

The following call will pick up the current output stream, create (if not already present) and open (if not already open) a file called "foo" in the current working directory, write the characters, "abc", to the file, and finally reset output to the original stream, without closing the file:

    ?- telling( C ), tell( foo ), write( `abc` ), tell( C ).
    C = user

Notes

Three related predicates, tell/1, telling/1 and told/0, support high-level control over the output stream, and provide for the easy handling of sequential access files: however, it does not have direct access to special numerically named devices or to memory files. Another predicate, input/1, provides low-level control over all types of WIN-PROLOG input stream.

This is one of a series of file handling predicates which make use of "logical" file names: FileSpec can therefore be anything from a simple atom to a nested compound term of the general form: "PathAlias(FileSpec)". Path aliases are declared by asserting clauses to file_search_path/2, and are expanded into atom file names by the absolute_file_name/3 predicate before being used.
**term_expansion/2**

*user-definable predicate for extending term expansion*

```
term_expansion( Term1, Term2 )
```

### Comments

If defined by the user, this is called by the `expand_term/2` predicate during term expansion: if this call succeeds, term expansion is considered complete; if it fails, `expand_term/2` uses its default term expansion rules.

### Examples

Consider the following program, which includes a definition of `term_expansion/2`, together with two "clauses" for `foo/1`:

```
term_expansion( foo(Integer), foo(Integer,String) ) :-
    integer( Integer ),
    ( write( `16` ),
      fwrite( r, 8, 16, Integer )
    ) ~> String.
```

`foo( 123 ).`
`foo( abc ).`

Once compiled, the first clause for `foo/1` will have been expanded into a clause for `foo/2`, as shown here:

```
?- listing( foo ).
/* foo/1 */
foo( abc ).
/* foo/2 */
foo( 123, `16'0000007B` ).
```
Notes
Together with `expand_term/2`, the `term_expansion/2` hook allows special grammar rules to be added to the default grammar rule processing: because `expand_term/2` is called during compilation, this feature allows the Prolog syntax to be extended in a completely arbitrary manner.
**throw/2**

generate an error condition with the given goal

```
throw( Error, Goal )
```

+Error               <integer>
+Goal               <goal>

**Comments**

This generates an error condition that is reported as having occurred in the given Goal: the Goal is not called, and variables are not bound, even in the special "success" case which is shown here, together with the "fail" and "error" cases:

<table>
<thead>
<tr>
<th>Error</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>throw/2 succeeds without generating an error</td>
</tr>
<tr>
<td>-1</td>
<td>throw/2 fails without generating an error</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>throw/2 simulates the effect of the goal generating error number &quot;integer&quot;</td>
</tr>
</tbody>
</table>

**Examples**

The following call simulates the "Arithmetic Overflow Error" in the `is/2` predicate, even though this particular goal would normally work:

```
?- throw( 52, X is 9 * 9 ).
! ----------------------------------------
! Error 52 : Arithmetic Overflow
! Goal     : _1 is 9 * 9
```

When called with an error number of zero ("0"), this predicate succeeds directly; the given goal is not called, and so does not return any variable bindings:

```
?- throw( 0, X is 9 * 9 ).
X = _
```

**Notes**

The `throw/2` predicate is normally used together with `catch/2`, providing complete error management; consider the program:

```
square_root( Arg1, Arg2 ) :-
    catch( Error, Arg2 is sqrt(Arg1) ),
    !,
```
throw( Error, square_root(Arg1,Arg2) ).

Here, any error that occurs during the call to "Arg2 is sqrt(Arg1)" is "caught" by catch/2, and then "thrown" by throw/2, making it look as if any errors occured directly inside square_root/2, rather than in is/2:

?- square_root( 123, X ).
X = 11.0905365064094

?- square_root( -123, X ).<enter>
! ----------------------------------------
! Error 53 : Arithmetic Error
! Goal     : square_root(-123,_1)

Note the cut (/0) in this program, which removes the "extra" choicepoint left after a successful call to is/2: see the entry for catch/2 for further information.
**time/2**

*get elapsed running time or local computer time*

```
?time( Flag, Time )
```

*Flag* 
<integer>, <conjunction> or <variable>

*Time* 
<variable>

**Comments**

This binds *Time* to a "date_time" structure, returning either the elapsed running time of the current **WIN-PROLOG** session or the "local" computer time according to the operating system, depending upon the value given in *Flag*, as shown in the following table:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>return elapsed time in days and milliseconds since the start of the current Prolog session</td>
</tr>
<tr>
<td>1</td>
<td>return local time in days and milliseconds since 00:00:00.000 on 01 Jan 1600</td>
</tr>
</tbody>
</table>

If both *Flag* and *Time* are given as variables, they return, respectively, 64-bit time reference and an integer specifying the resolution (see **Notes**, below). If time reference is passed as *Flag* to a subsequent call to *time/2*, the elapsed tick count is returned as *Time*: by dividing this latter value with the resolution returned in the first call, a very accurate timing, of at least the order of 1µs, can be obtained.

**Examples**

The following call picks up the current date and time, and uses it to set the timestamp of the file, "foo" (this example assumes that the date is 01-APR-1999, and the time is 10:08:42):

```
?- time( 1, T ), stamp( foo, T ).
T = (145822,36522000)
```

The next call picks up the number of days and milliseconds for which the current **WIN-PROLOG** session has been running (in this example, this time is just over one second):

```
?- time( 0, T ).
T = (0,1024)
```

The following query calls *time/2* twice, once to pick up a time reference, and a second time, to pick up an elapsed time relative to that reference, in this case, to time a call to true/0:
?- time( Top, Res ), true, time( Top, End ), Time is End / Res.
Top = 1,838741106 ,
Res = 1193180 ,
End = 7 ,
Time = 5.866756066362829e-6

The values returned the first call to time/2 show a 64-bit time reference and its resolution in Top and Res respectively; when the latter is passed to a second call to time/2, the elapsed tick count is returned, in End. Dividing End by Res gives a timing, in seconds.

Notes

Starting with WIN-PROLOG 4.7, the time/2 predicate can be used to access the highest-resolution clock available on any given computer platform. By bracketing any Prolog goal between two calls to time/2, as shown here, very accurate timings, of a known resolution, can be obtained:

time( Top, Res ),
<GOAL>
time( Top, End ),
Time is End / Res,

The values returned by the first call comprise a 64-bit reference time, returned as a conjunction of two 32-bit integers, and count of ticks per second, which will usually be 1,193,180 or greater (the bigger the number, the finer the clock resolution); when the reference time is passed into the second call, the number of ticks that have elapsed is returned: simple division can then be used to compute an accurate timing. The ms/2 predicate has been rewritten to use this new feature of time/2, which explains why it now has a resolution of around 1µs (one microsecond) or better.

This is one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date_time" data type provides a uniform structure for handling, processing and storing dates and times throughout WIN-PROLOG. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, time/4 and time/5, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, time/2, is used to return current local or elapsed time.
time/3

convert from date/time structure to formatted date and time

\[
\text{time(\ DateTime, Date, Time )}
\]

+\DateTime \ <\text{date\_time}>
-Date \ <\text{variable}>
-Time \ <\text{variable}>

Comments
This takes a "date\_time" structure, \DateTime\, and formats its date and time components into a pair of strings which are bound to \Date\ and \Time\ respectively.

Examples
The following call picks up the current date and time, and converts the day number component into a year, month and day (this example assumes that the date is 01-APR-1999, and the time is 10:08:42):

\[
\text{?- time(1, T ), time( T, Date, Time )}. \quad <\text{enter}>
\]
\[
T = (145822,36522000) ,
Date = `Thu 01 Apr 1999` ,
Time = `10:08:42`
\]

Notes
This is one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date\_time" data type provides a uniform structure for handling, processing and storing dates and times throughout \textbf{WIN-PROLOG}. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, \textit{time/4} and \textit{time/5}, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, \textit{time/2}, is used to return current local or elapsed time.

The present predicate, \textit{time/3}, is useful both when displaying the current time, as in the example above, and also when displaying time stamps for files, or the results of date and time computations, when a standardised format is required.
**time/4**

convert between day number and date

```prolog
time( Number, Year, Month, Day )
```

?Number <integer> or <variable>
?Year <integer> or <variable>
?Month <integer> or <variable>
?Day <integer> or <variable>

**Comments**
This converts between a `Number` of days since 01-JAN-1600 and the corresponding `Year`, `Month` and `Day`. If `Number` is an integer, then `Year`, `Month` and `Day` must be unbound variables which return the computed date; if `Year`, `Month` and `Day` are integers, `Number` must be an unbound variable which returns the computed day number.

**Examples**
The following call picks up the current date and time, and converts the day number component into a year, month and day (this example assumes that the date is 01-APR-1999, and the time is 10:08:42):

```prolog
?- time( 1, T ), T = (N,_), time( N, Y, M, D ).
```

```
T = (145822,36522000),
N = 145822,
Y = 1999,
M = 4,
D = 1
```

The next call uses the day number returned above to compute the number of days until the following Christmas:

```prolog
?- time( X, 1999, 12, 25 ), D is X - 145822.
```

```
X = 146090,
D = 268
```

**Notes**
This is one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date_time" data type provides a uniform structure for handling, processing and storing dates and times throughout **WIN-PROLOG**. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, `time/4` and `time/5`, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, `time/2`, is used to return current local or elapsed time.
**time/5**

*convert between tick count and time*

```
\text{time}( \text{Number}, \text{Hour}, \text{Minute}, \text{Second}, \text{Millisecond} )
```

- ?Number: <integer> or <variable>
- ?Hour: <integer> or <variable>
- ?Minute: <integer> or <variable>
- ?Second: <integer> or <variable>
- ?Millisecond: <integer> or <variable>

**Comments**

This converts between a `Number` of milliseconds and the corresponding `Hour`, `Minute`, `Second` and `Millisecond`. If `Number` is an integer, then `Hour`, `Minute`, `Second` and `Millisecond` must be unbound variables which return the computed time; if `Hour`, `Minute`, `Second` and `Millisecond` are integers, `Number` must be an unbound variable which returns the computed day number.

**Examples**

The following call picks up the current date and time, and converts the tick count component into an hour, minute, second and millisecond (this example assumes that the date is 01-APR-1999, and the time is 10:08:42):

```
?- \text{time}( 1, T ), T = (_,N), \text{time}( N, H, M, S, F ).
```

```
T = (145822,36522000)
N = 36522000
H = 10
M = 8
S = 42
F = 0
```

**Notes**

This is one of a family of date and time handling predicates which store time as a conjunction of two integers, representing days and milliseconds respectively: this "date_time" data type provides a uniform structure for handling, processing and storing dates and times throughout WIN-PROLOG. Dates are counted as days since 01-JAN-1600, and times as milliseconds since midnight. Two predicates, `time/4` and `time/5`, convert between day numbers and (Y/M/D) dates, and between millisecond counts and (H:M:S:F) times respectively; a third, `time/2`, is used to return current local or elapsed time.
time/7

return the local machine date and time

time( Year, Month, Day, Hour, Minute, Second, Millisecond )

- Year <variable>
- Month <variable>
- Day <variable>
- Hour <variable>
- Minute <variable>
- Second <variable>
- Millisecond <variable>

Comments This returns the Year, Month, Day, Hour, Minute, Second and Millisecond of the "local" computer time according to the operating system.

Examples The following call picks up the current date and time (this example assumes that the date is 01-APR-1999, and the time is 10:08:42):

?- time( Y, N, D, H, M, S, F ).

Y = 1999 ,
N = 4 ,
D = 1 ,
H = 10 ,
M = 8 ,
S = 42 ,
F = 0

Notes The time/7 predicate is provided for convenience, and is the only system-level time predicate not to use the "date_time" data type (see time/2 for further information). The former predicate is written in terms of the latter, together with the conversion predicates, time/4 and time/5, as follows:

time( Year, Month, Day, Hour, Minute, Second, Millisecond ) :-
    time( 1, Real ),
    Real = (Date,Time),
    time( Date, Year, Month, Day ),
time( Time, Hour, Minute, Second, Millisecond ).

Because date and time are read simultaneously, time/7 is immune from the "midnight rollover bug" which occurs when date and time are read independently, either side of midnight, resulting in an time error of about 24 hours.
**timer_close/1**

*close a named timer*

\[
\text{timer_close( Name )}
\]

+Name \text{<atom>}

**Comments**

This closes the named timer with the given *Name*, returning its resources to the timer pool; if the timer does not exist, this predicate generates an error.

**Examples**

The following call will close the named timer called "foo", assuming that such a timer has previously been created:

\[
?\text{- timer_close( foo ).}
\]

yes

**Notes**

The WIN-PROLOG system supports 64 independent timers, numbered 0..63, which can be set in any combination to perform elapsed-time interrupts over periods ranging from 1 millisecond to \(2^{31}\) milliseconds (about 24.8 days). Timers are created with `timer_create/2`, set through `timer_set/2`, queried using `timer_data/2`, `timer_dict/2` and `timer_get/2`, and closed with `timer_close/1`. See the "notes" section of the entry for `timer_create/2` for further information about named timers.
**timer_create/2**

*create a named timer*

\[
\text{timer_create}( \text{Name}, \text{Pred} )
\]

+Name <atom>
+Pred <atom>

**Comments**

This creates a named timer with the given \text{Name}, and declaring that its events are handled by the predicate, \text{Pred}/2. The timer is not actually set by this call, but is simply declared. If a previous timer of this name exists, it is automatically stopped and closed before the new version is created, returning its resources to the pool of 64 timers.

**Examples**

Consider the following timer handler program:

\[
\text{foo_handler}( \text{Name}, \text{Status} ) :-
\]

\[
( \text{write( Name - Status ), nl } ) \leadsto \text{user.}
\]

Once compiled, the following call will create a timer that will use this handler:

\[
?- \text{timer_create}( \text{foo}, \text{foo_handler} ).
\]

yes

The next call sets the timer to "fire" after 1000ms:

\[
?- \text{timer_set}( \text{foo}, 1000 ).
\]

yes

After a period of 1000ms, the timer will fire, calling the \text{foo_handler/2} predicate defined earlier, displaying the following output, where "12345" will be the expiry time (this is the sum of the interval and the original setting time):

\[
\text{foo} - (1000,12345)
\]

This command can be used to cancel a timer, while picking up its status:
?- timer_get( foo, Status ).
Status = 0

If the timer was active, its status is reported as a pair of integers that can be used to restart the timer in a subsequent call to \texttt{timer_set/2}; if the timer was inactive (either it had not been set, or had been set but subsequently expired), the returned status is simply "0", as in the above example.

Notes

The \texttt{WIN-PROLOG} system supports 64 independent timers, numbered 0..63, which can be set in any combination to perform elapsed-time interrupts over periods ranging from 1 millisecond to $2^{31}$ milliseconds (about 24.8 days). Timers are created with \texttt{timer_create/2}, set through \texttt{timer_set/2}, queried using \texttt{timer_data/2}, \texttt{timer_dict/2} and \texttt{timer_get/2}, and closed with \texttt{timer_close/1}.

When a timer expires, it interrupts the Prolog execution stream, replacing a goal with a call to the handler specified when the timer was created. This handler receives the name of the timer that has just expired, together with its status, which contains the originally specified interval and the time at which it was set to expire, both measured in milliseconds, and comprises the least-significant 32 bits of \texttt{WIN-PROLOG}'s 64-bit elapsed time counter.

Timers are intrinsically one-shot events: if it is desired to invoke the timer periodically, \texttt{timer_set/2} should be called as the last operation in the timer handler predicate, passing in the name and status arguments. Because the latter parameter includes both an interval value and the original setting time, the next timer interrupt is automatically set relative to the previous expiry time, rather than the current time: this feature enables \texttt{WIN-PROLOG} timers to retain synchronisation with real time, provided the overall time required to process the interrupt does not exceed the specified interval period.
timer_data/2

return data for a named timer

\[ \text{timer_data}( \text{Name}, \text{Pred} ) \]

+Name <atom>
-Pred <variable>

Comments This returns the name of the predicate, \text{Pred}/2, that handles events for the timer with the given \text{Name}; if the timer does not exist, this predicate generates an error.

Examples The following call will create a timer called "foo" that uses a handler called "foo_handler/2":

\[ \text{?- timer_create}( \text{foo}, \text{foo_handler} ). \]

\[ \text{yes} \]

The next call returns the name of the timer's handler predicate:

\[ \text{?- timer_data}( \text{foo}, \text{Handler} ). \]

\[ \text{Handler} = \text{foo_handler} \]

Notes The \text{WIN-PROLOG} system supports 64 independent timers, numbered 0..63, which can be set in any combination to perform elapsed-time interrupts over periods ranging from 1 millisecond to \(2^{31}\) milliseconds (about 24.8 days). Timers are created with \text{timer_create/2}, set through \text{timer_set/2}, queried using \text{timer_data/2}, \text{timer_dict/2} and \text{timer_get/2}, and closed with \text{timer_close/1}. See the "notes" section of the entry for \text{timer_create/2} for further information about named timers.
timer_dict/2

 return a dictionary of named timers

    timer_dict( Flag, Dict )

+Flag   <integer> in the domain {-1,0,1}
-Dict   <variable>

Comments  This returns a list of all the currently defined named timers, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples  The following call will return a list of currently defined named timers, assuming that one called "foo" has previously been created:

    ?- timer_dict( 0, D ).
    D = [foo]

Notes  The WIN-PROLOG system supports 64 independent timers, numbered 0..63, which can be set in any combination to perform elapsed-time interrupts over periods ranging from 1 millisecond to $2^{31}$ milliseconds (about 24.8 days). Timers are created with timer_create/2, set through timer_set/2, queried using timer_data/2, timer_dict/2 and timer_get/2, and closed with timer_close/1. See the "notes" section of the entry for timer_create/2 for further information about named timers.
**timer_get/2**

reset and get the status of a named timer

\[
\text{timer_get}( \text{Name}, \text{Status} )
\]

+Name <atom>
-Status <variable>

Comments
This resets (cancels) a named timer with the given Name, and returns its previous Status. If the timer was active, its Status is reported as pair of integers that can be used to restart the timer in a subsequent call to \text{timer_set/2}; if the timer was inactive (either it had not been set, or had been set but subsequently expired), the returned status is simply "0".

Examples
The following call will create a timer called "foo" that uses a handler called "foo_handler/2":

\[
?\text{- timer_create}( \text{foo}, \text{foo_handler} ).
\]

yes

The next picks up the timer status, resetting it if it was active:

\[
?\text{- timer_get}( \text{foo}, \text{Status} ).
\]

Status = 0

Notes
The WIN-PROLOG system supports 64 independent timers, numbered 0..63, which can be set in any combination to perform elapsed-time interrupts over periods ranging from 1 millisecond to \(2^{31}\) milliseconds (about 24.8 days). Timers are created with \text{timer_create/2}, set through \text{timer_set/2}, queried using \text{timer_data/2}, \text{timer_dict/2} and \text{timer_get/2}, and closed with \text{timer_close/1}. See the "notes" section of the entry for \text{timer_create/2} for further information about named timers.
**timer_set/2**

set the status of a named timer

```
timer_set( Name, Status )
```

+Name <atom>
+Status <integer> or (<integer>, <integer>)

**Comments**

This sets (arms) a named timer with the given Name, using the given Status to define its expiry time. If Status is an integer, the current elapsed time is taken as a timebase, and the timer will fire the given number of milliseconds after this time; if Status is a pair of integers, the first is interpreted as the millisecond interval, and the second is taken as the timebase.

**Examples**

Consider the following timer handler program:

```
foo_handler( Name, Status ) :-
    (  write( Name - Status ),
    nl
    ) -> user.
```

Once compiled, the following call will create a timer that will use this handler:

```
?- timer_create( foo, foo_handler ).
yes
```

The next call sets the timer to "fire" after 1000ms:

```
?- timer_set( foo, 1000 ).
yes
```

After a period of 1000ms, the timer will fire, calling the foo_handler/2 predicate defined earlier, displaying the following output, where "12345" will be the expiry time (this is the sum of the interval and the original setting time):

```
foo - (1000,12345)
```

This command will also set the timer, with an interval of 1 hour (3,600,000 ms), using the given value, "0", as the timebase rather than
current elapsed time, so that the timer will fire one hour after the start of the current WIN-PROLOG session:

?- timer_set( foo, (36000000,0) ).
yes

Exactly one hour into the WIN-PROLOG session, the following output will be displayed:

foo - (3600000,3600000)

If this value was passed back into timer_set/2, the timer would fire again exactly one hour later.

Notes

The WIN-PROLOG system supports 64 independent timers, numbered 0..63, which can be set in any combination to perform elapsed-time interrupts over periods ranging from 1 millisecond to $2^{31}$ milliseconds (about 24.8 days). Timers are created with timer_create/2, set through timer_set/2, queried using timer_data/2, timer_dict/2 and timer_get/2, and closed with timer_close/1. See the "notes" section of the entry for timer_create/2 for further information about named timers.
**tipbox/3**

display or hide the modeless system tooltip

```prolog
?Message <string> or <variable>
?X <integer> or <variable>
?Y <integer> or <variable>
```

**Comments**
This shows, hides or retrieves information about the system "Tooltip" window, which can be used to display helpful text prompts at any point on the screen. If `Message` is bound to a non-empty string, the tooltip is shown at the screen location specified in the given `X` and `Y` coordinates; if `Message` is bound to an empty string, the tooltip is hidden. If all three arguments are variables, they become bound to the existing `Message` string and `(X,Y)` coordinates.

**Examples**
The following call displays the tooltip near the top left of the screen, before returning control to the console, leaving the tooltip visible:

```prolog
?- tipbox( `The quick brown fox`, 50, 50 ).
yes
```

The text can be updated simply by making another call:

```prolog
?- tipbox( `jumps over a lazy dog`, 50, 50 ).
yes
```

The existing text and screen location of the tooltip is returned by the following call:

```prolog
?- tipbox( Message, X, Y ).
Message = 'jumps over a lazy dog',
X = Y = 50
```

To make the tooltip disappear, an empty string should be passed into the call (the location coordinates are ignored in this case, but must still be integers):

```prolog
?- tipbox( ``, 0, 0 ).
yes
```
The `tipbox/3` predicate provides direct access to a specially defined "system" tooltip control, to simplify the process of displaying short help prompts, typically in the vicinity of the mouse pointer in response to a "msg_mousehover" message; it is normal to hide the tooltip on some other mouse event, such as "msg_mousemove" or "msg_mouseleave".

The tooltip is created by WIN-PROLOG at startup time, and is shared by all dialogs and toolbars within its environment. It uses the "ws_ex_topmost" extended window style, which means that it will always appear above all other non-topmost windows, not just those created in the current WIN-PROLOG session, but also those of other applications.

Because the system tooltip is thought of as not belonging to any window in particular, its (X,Y) coordinates are specified relative to the top left of the screen, and not any given window's client area. This means that in order to display the tooltip near the mouse pointer, especially in response to one of the "msg_mouse" messages, it is necessary to convert between "client" and "screen" coordinates, using the `warea/5` predicate to provide the conversion data.

The following code fragment shows a window handler which displays a tooltip in response to "msg_mousehover", at a position just below the mouse cursor. One of two adjustments is used: for "Edit" windows, the tooltip is shown 10 pixels below the default i-bar cursor, while for other windows, it is shown 20 pixels below the default arrow cursor:

```prolog
my_handler( Window, msg_mousehover, (X0,Y0), _ ) :-
    wclass( Window, Class ),
    warea( Window, X1, Y1, _, _ ),
    ( Class = 'Edit'
     -> Adjust = 10
      ; Adjust = 20
    ),
    X is X0 + X1,
    Y is Y0 + Y1 + Adjust,
    writeq( Window = Class - (X0,Y0) ) ~> String,
    tipbox( String, X, Y ).
```

In common with other predefined windows in WIN-PROLOG, the system tooltip can be referred to directly using a special numerical window handle, "3": in general, however, it should be programmed only through the `tipbox/3` predicate.
told/0

close the current output stream and revert to user output

told

Comments
This closes any file associated with the current output stream, and resets output to the "user" (console output) device.

Examples
The following call will create (if not already present) and open (if not already open) a file called "foo" in the Prolog home directory, write the term, "hello", to the file, and finally close it and reset the output stream to the console (user) device:

?- tell(prolog(foo)), write(`hello.`), nl, told.  
yes

Notes
Three related predicates, tell/1, telling/1 and told/0, support high-level control over the output stream, and provide for the easy handling of sequential access files: however, it does not have direct access to special numerically named devices or to memory files. Another predicate, input/1, provides low-level control over all types of WIN-PROLOG input stream.
**total/9**

return the total amounts of space

\[ \text{total}( \text{B, L, R, H, T, P, S, I, O}) \]

-B <variable>
-L <variable>
-R <variable>
-H <variable>
-T <variable>
-P <variable>
-S <variable>
-I <variable>
-O <variable>

**Comments**
This returns the total amount of space, in bytes, in each of these memory areas: backtrack stack (B), local stack (L), reset stack (R), term heap (H), text heap (T), program heap (P), system stack (S), string input buffer (I) and string output buffer (O).

**Examples**
The following call returns the total amount of program space:

?-_total( _, _, _, _, _, P, _, _, _).

P = 8388480

**Notes**
The total/9 predicate returns the total size of each of the nine configurable memory areas, but does not indicate how much of this space is used and how much is free. Another predicate, free/9, returns the amounts of memory currently free in the same nine areas: by comparing the results of calling free/9 and total/9, it is possible to compute additional statistics, such as the amount of space being used, or the percentage of space remaining free. See the entries for free/9 and gc/1 for additional information regarding garbage collection.

A further related predicate, xinit/9, allows changes to be made to the memory allocation at runtime. Using this, it is possible to adjust memory usage in response to error conditions, or simply to optimise resource allocation within an application.
toteall/3

return numerical totals from solutions of a given goal

toteall( Term, Goal, Total )

?Term <term>
+Goal <goal>
?Total <variable> or <term>

Comments  This predicate returns a duplicate of the input Term, where every variable in that term which is shared by the Goal, is replaced by a numerical Total of all the solutions. It succeeds if Total can be unified to a duplicate of Term such that Goal is true, and the variables shared between Term and Goal are replaced by their numerical totals. The Term may be any Prolog term, and Goal may be any Prolog goal, with or without calls to the existential quantifier predicate, /2, which is ignored in the present predicate (see "notes" below).

Examples  Consider the database relation:

    foo( 1, 1 ).
    foo( 2, 2 ).
    foo( 1, 3 ).
    foo( 2, 4 ).

The following call returns a list of all solutions of "X" in the goal "foo(Y,X)" without partitioning on different values of "Y":

    ?- toteall( (X,Y), foo(X,Y), Z ).  <enter>

    X = _ ,
    Y = _ ,
    Z = (6,10)

This predicate can be very useful in the preparation of certain types of statistics, such as the mean and standard deviation, which require the counting of solutions, as well as the toting up of their values and the squares of values. Consider another database relation:

    foo( 9 ).
    foo( 2 ).
    foo( 27 ).
    foo( 81 ).
The following call counts up the total number of solutions and the total of those solutions in order to calculate the arithmetic mean:

?- toteall( (One,Val), (One=1,foo(Val)), (Count,Total) ), Mean is Total/Count. <enter>
One = _ ,
Val = _ ,
Count = 4 ,
Total = 119 ,
Mean = 29.75

**Notes**

Unlike bagof/3, setof/3 and findall/3, toteall/3 does return lists of solutions; rather it can be used to total up rules for assorted statistical purposes, such as the computation of solution counts, arithmetic means and standard deviations.

Like findall/3, toteall/3 does not perform any partitioning of results. It therefore has no concept of "existential quantification", and as such treats any calls to ^/2 just as if the appropriate goal were being called directly. For further information about existential quantification, see the entries for bagof/3 and setof/3.
touch/0

check whether touch is supported

touch

Comments
This predicate tests whether the current version of the operating system supports touch input.

Examples
The following call checks to see whether touch is available on the current operating system:

?- touch.

yes

Notes
Touch, or more specifically, multi-touch input, is available in Windows 7 and later, and WIN-PROLOG version 5.000 onwards supports tracking of up to 16 simultaneous touch points on any windows which have been "touch-enabled".

Please note that touch/0 only tests for the existence of the Windows Touch APIs, and not for the physical presence of a touch screen or other touch input device. Its primary purpose is to test whether subsequent calls to touch/3 will raise errors due to the absence of the Touch APIs, as would be the case in any versions of Windows prior to Windows 7.

See touch/3 for more detailed information about multi-touch input.
**touch/3**

*get, set or test the touch status of a window, or return most recent touch point data*

```prolog
touch( Window, Status, Flags )
```

- **Window**  
  `<window_handle>` or `<variable>`

- **Status**  
  `<integer>` in the domain `{0,1}` or `<variable>`

- **Flags**  
  `<integer>` in the domain `{0,1,2,3}` or `<variable>`

**Comments**

Where `Window` specifies a window handle, this predicate gets (`Status` and `Flags` are both variables) or sets (`Status` and `Flags` are both integers) the touch-enabled status of the given window. Where all three arguments are variables, this predicate returns information about the most recent touch event, returning the cancelled touch point handle in `Window`, its time stamp in `Status` and the list of raw touch data in `Flags`.

**Examples**

The following call will set window "(foo,123)" to be touch-enabled, with the "TWF_WANTPALM" (16'2) flag specified:

```prolog
?- touch( (foo,123), 1, 2 ).
      yes
```

Meanwhile, the following call will check the existing touch-enabled state of the same window:

```prolog
?- touch( (foo,123), Status, Flags ).
      Status = 1 ,
      Flags = 2
```

Assuming one or more touch events have occurred, the following call will retrieve information about the most recent event:

```prolog
?- touch( Handle, Time, Points ).
      Handle = 12345 ,
      Time = 67890 ,
      Points = [0,1,23,45,1,1,67,89]
```

**Notes**

Touch, or more specifically, multi-touch input, is available in Windows 7 and later, subject to the presence of an appropriate touch input device (for example, tablet screen, or digitiser pad), and **WIN-PROLOG** version 5.000 onwards supports tracking of up to 16 simultaneous touch points on any windows which have been "touch-enabled".
When a window is "touch-enabled", the **WIN-PROLOG** kernel picks up touch messages, manages the touch handles, and reports "msg_touch" messages to applications. When picked up through a window handler, *msg_touch* bypasses the need to call *touch/3*, because it does so behind the scenes, massaging the raw data into a list of points, where each point is of the form:

$$ \text{ID(State, (X,Y))} $$

where:

<table>
<thead>
<tr>
<th>ID</th>
<th>State</th>
<th>(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>is a unique integer within a given series of touch events</td>
<td>is an atom in the domain {down, move, up}</td>
<td>are the X and Y client coordinates of the touch point</td>
</tr>
</tbody>
</table>

Coordinates are expressed in pixels relative to the top left of the window that receives the *msg_touch*. An ID of 0 is given to the "primary" touch point in any given sequence, which is effectively the mouse location, and subsequent touch points are numbered sequentially.

Touch events form "touch sequences", which begin when the first contact is made between a finger (or pen) and the touch surface within the confines of a touch-enabled window. The first touch event in a sequence will by definition include the primary touch point (with an ID of 0), in conjunction with a "down" state. From that point on, a stream of messages will follow, including introduction of further touch points, moving of existing touch points ("move" state), and finishing of touch points ("up" state). Only when all fingers (or pen) have been removed from the screen, will the touch sequence be deemed complete.

Note that if when second or subsequent touch point is active, the primary touch point can be released, without terminating the touch sequence. No new primary touch point can be established until all touch points have been removed, and a new one created.

The *touch/3* predicate is used internally by **WIN-PROLOG**'s message processor, to retrieve touch information in direct response to a *msg_touch* message, before calling a user-defined handler. The only time applications should call *touch/3* with three variables, is where a message hook (for example, *?MESSAGE?/4*) is being defined for debugging purposes. In this case, the list returned consists of a series of groups of 4 numbers, each group representing the ID, State, X and Y coordinate of a single point. The state will be an integer combining three mutually exclusive flags:

<table>
<thead>
<tr>
<th>16'0001</th>
<th>16'0002</th>
<th>16'0004</th>
</tr>
</thead>
<tbody>
<tr>
<td>the &quot;move&quot; state (ie, existing point update)</td>
<td>the &quot;down&quot; state (ie, new point established)</td>
<td>the &quot;up&quot; state (ie, existing point finished)</td>
</tr>
</tbody>
</table>

Note also that *touch/3*, used to retrieve touch information, is a one-shot predicate. Calling it a second time will return a handle of zero, a timestamp which should normally be zero (occasionally it will be a small positive integer), and an empty list.
Touch must be explicitly enabled on every window for which it is desired to process touch messages. In the absence of touch enabling, Windows may still react to touch, using mouse and scroll messages that are automatically generated by the Windows operating system in response to certain touch events.

Please note that touch/0 only tests for the existence of the Windows Touch APIs, and not for the physical presence of a touch screen or other touch input device. Its primary purpose is to test whether subsequent calls to touch/3 will raise errors due to the absence of the Touch APIs, as would be the case in any versions of Windows prior to Windows 7.
trace/0

set the debugging mode to "trace"

trace

Comments
This predicate sets the debugging mode to "trace", which will result in the debugger being invoked whenever a goal is executed from the command prompt.

Examples
The following call sets the debugging mode to "trace" so that the debugger will take over at the next command to be entered:

?- trace.
   # Debugging mode switched to trace
   yes

Notes
The debugging mode has three settings: "off", "debug", and "trace". In the first case, the debugger is never invoked, while in the second, it is invoked if a spied predicate is called. In the third case, the debugger is invoked immediately a query is entered at the console prompt.

The three debugging modes are mutually exclusive, but are somewhat confusingly set by four predicates, as shown in the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>nodebug/0 or notrace/0</td>
</tr>
<tr>
<td>debug</td>
<td>debug/0</td>
</tr>
<tr>
<td>trace</td>
<td>trace/0</td>
</tr>
</tbody>
</table>

The nodebug/0 and notrace/0 predicates are effectively identical, and either can be used to cancel the effects of either debug/0 or trace/0. Please note that WIN-PROLOG's default debugging mode is "debug".
true/0

succeed

true

Comments  This predicate simply succeeds, and is used as a "no-op" or null goal where it is desired to do nothing further; it is mainly used in the context of "if:then" or "if:then:else" structures, or as the final "succeed" branch of an "either:or" disjunction.

Examples  Consider the following program:

    foo( 1 ).
    foo( 2 ).
    foo( 3 ).

By forcing failure with fail/0, it is possible to display all solutions; the call to true/0 enables the disjunctive query finally to succeed:

    ?- (foo( X ), write( X ), nl, fail) ; true.  

1
2
3
X = _

Notes  The true/0 predicate does absolutely nothing, but is useful where a null, placeholder goal has to be provided, for example as an argument to a metaprogramming predicate, or as a null clause body (see clause/2). Notice, for example, that the "if:then" predicate, ->/2, fails if its "if" test goal fails:

    ?- (integer( 3.14 ) -> write( `Integer` )).  

no

In order to get an "if:then" construct which succeeds even if the "if" goal fails, the "if:then:else" case of ;/2 can be used, with its "else" branch simply comprising a call to true/0:

    ?- (integer( 3.14 ) -> write( `Integer` ) ; true).  

yes
Please note that another predicate, *otherwise/0*, is in fact identical to *true/0*; the former name is provided as a synonym for compatibility with programs written for some earlier Prolog systems; new applications should use the latter predicate.
ttyflush/0

Flush the user output stream

ttyflush

Comments
This causes any data written to the console device (user output stream), but not yet displayed, to be flushed to this device.

Examples
Consider the following program, which provides a simple delay of one second (1000ms) by calling repeat/0 within ms/2, and testing the resulting time is greater than 999:

```prolog
pause :-
    ms( repeat, T ),
    T > 999,
    !.
```

Once this is compiled, the following call will write a word, "hello", to the console (assuming that "user" is the current output stream) and then delay for one second. The output will only be finally seen after the time has elapsed, simultaneously with the success flag, "yes":

```
?- write( hello ), nl, pause.
```

```
hello
yes
```

By adding a call to ttyflush/0, the output can be displayed immediately, one whole second before "yes" is echoed:

```
?- write( hello ), nl, ttyflush, pause.
```

```
hello
yes
```

Notes
The WIN-PROLOG console (user) output device buffers up to 4096 characters of data before physically displaying the text in the console window; this is done to overcome the inefficiency of text display in the Windows environment. Normally, buffered text is flushed to the console either when the 1025th character is written, or when console (user) input is attempted: the latter case ensures that all output and prompts are complete prior to reading input from the user. Programs which perform prolonged processing, writing occasional data (such as status reports) to the console can use ttyflush/0 to ensure that such output appears immediately.
ttyget/1

read a printable character from the user input stream

\[ \text{ttyget( Char )} \]

?Char \quad <\text{variable}> \text{ or } <\text{char}>

Comments
This reads the next "printable" (non-white-space) character from the console (user input stream), and unifies Char with its character code. A printable character is defined to be one whose character code is greater than that of <space> (20h).

Examples
The following call reads the first printable character from the next line of input at the console, which begins with a <tab> character (this is skipped). Note that a call to the flush/0 predicate has been added: this is to dispose of any excess characters ("ello" in this example):

```
?- ttyget( C ), flush.
|: hello
C = 104
```

Notes
The ttyget/1 predicate is closely related to ttyget0/1: the only difference is that the former scans input for printable characters (those with character codes of 33 or greater), while the latter returns all characters. Both these predicates return the special value of "-1" if they encounter the end of file at the console (this is signalled if the user presses <escape>).

This is one of a set of related predicates, including ttyget/1, ttyget0/1, ttyskip/1, ttyput/1, ttytab/1 and ttynl/0, which are provided mainly for compatibility with older Prologs. They each perform identical functions to their "non-tty" equivalents, except that they directly read from or write to the console (user input or output device), ignoring the current input or output stream setting. For example, ttyget/1 and ttyput/1 are implemented simply as follows:

```
ttyget( Char ) :-
    get( Char ) <= user.

ttyput( Expr ) :-
    put( Expr ) => user.
```

Curiously, the set is incomplete: there are, for example, no equivalent "ttyread/1" or "ttywrite/1" predicates.
ttyget0/1

*read a character from the user input stream*

\[
n\text{ttyget0}( \text{Char} )\n\]

?Char \hspace{1cm} \text{<variable> or <char>}

**Comments**  
This reads the next character from the console (user input stream), and unifies Char with its character code.

**Examples**  
The following call reads the first character from the next line of input at the console, which begins with a \(<\text{tab}>\) character (this is returned). Note that a call to the \text{flush/0} predicate has been added: this is to dispose of any excess characters ("hello" in this example):

\[
?- \text{ttyget0}( \text{C} ), \text{flush}. \quad \text{<enter>}
\]

|: hello  \quad \text{<enter>}

\[
\text{C} = 9
\]

**Notes**  
The \text{ttyget0/1} predicate is closely related to \text{ttyget/1}: the only difference is that the former returns all characters, while the latter scans input for printable characters (those with character codes of 33 or greater). Both these predicates return the special value of "-1" if they encounter the end of file at the console (this is signalled if the user presses \(<\text{escape}>\)).

This is one of a set of related predicates, including \text{ttyget/1}, \text{ttyget0/1}, \text{ttyskip/1}, \text{ttyput/1}, \text{ttytab/1} and \text{ttynl/0}, which are provided mainly for compatibility with older Prologs. They each perform identical functions to their "non-tty" equivalents, except that they directly read from or write to the console (user input or output device), ignoring the current input or output stream setting. For example, \text{ttyget/1} and \text{ttyput/1} are implemented simply as follows:

\[
\begin{align*}
\text{ttyget}( \text{Char} ) &: - \\
& \text{get}( \text{Char} ) <\sim \text{user}. \\
\text{ttyput}( \text{Expr} ) &: - \\
& \text{put}( \text{Expr} ) \sim> \text{user}. 
\end{align*}
\]

Curiously, the set is incomplete: there are, for example, no equivalent "ttyread/1" or "ttywrite/1" predicates.
ttynl/0

output a new line to the user output stream

**ttynl**

**Comments**
This predicate simply outputs a carriage return and line feed to the console (user output stream), to complete a line of output.

**Examples**
The following call writes out two lines, each finishing with a new line command:

```prolog
?- write( hello ), ttynl, write( world ), ttynl. <enter>
hello
world
yes
```

**Notes**
The `ttynl/0` predicate provides a portable way to output the end of line sequence: although in Windows, it is directly equivalent to outputting `<cr><lf>`, other operating systems use different character combinations to indicate a new line.

This is one of a set of related predicates, including `ttyget/1`, `ttyget0/1`, `ttyskip/1`, `ttyput/1`, `ttytab/1` and `ttynl/0`, which are provided mainly for compatibility with older Prologs. They each perform identical functions to their "non-tty" equivalents, except that they directly read from or write to the console (user input or output device), ignoring the current input or output stream setting. For example, `ttyget/1` and `ttyput/1` are implemented simply as follows:

```prolog
ttyget( Char ) :-
    get( Char ) <~ user.

ttyput( Expr ) :-
    put( Expr ) ~> user.
```

Curiously, the set is incomplete: there are, for example, no equivalent "ttyread/1" or "ttywrite/1" predicates.
ttyput/1

*evaluate and write a character to the user output stream*

\[
ttyput( \text{Expression} )
\]

+Expression <expr>

**Comments**

This evaluates the given *Expression* by calling *is/2*, and writes the result to the console (user output stream), as if it were a character code; an error is generated if the *Expression* does not evaluate to an integer.

**Examples**

The following call illustrates the fact that *ttyput/1* performs arithmetic, and computes an expression before outputting the resulting integer as a character code:

\[
?- \text{ttyput( } \text{"a" - 32 }, \text{ttynl.}
\]

A
yes

**Notes**

Although similarly named, the *ttyput/1* predicate is not symmetrically opposed to *ttyget/1*: the latter never performs arithmetic, only dealing with variables or integers; furthermore, *ttyget/1* does not process non-printable characters (those with character codes of 20h or less).

This is one of a set of related predicates, including *ttyget/1*, *ttyget0/1*, *ttyskip/1*, *ttyput/1*, *ttytab/1* and *ttynl/0*, which are provided mainly for compatibility with older Prologs. They each perform identical functions to their "non-tty" equivalents, except that they directly read from or write to the console (user input or output device), ignoring the current input or output stream setting. For example, *ttyget/1* and *ttyput/1* are implemented simply as follows:

\[
\begin{align*}
ttyget( \text{Char} ) & : - \\
& \text{get( Char ) }<~ \text{user.}
\end{align*}
\]

\[
\begin{align*}
ttyput( \text{Expr} ) & : - \\
& \text{put( Expr ) }\sim > \text{user.}
\end{align*}
\]

Curiously, the set is incomplete: there are, for example, no equivalent "ttyread/1" or "ttywrite/1" predicates.
**ttyskip/1**

`evaluate and skip past a given character on user input`

```prolog
+Expression
```

**Comments**

This evaluates the given `Expression` by calling `is/2`, interprets the result as a character code, and then searches for this character on the console (user input stream). Searching stops when the character is found, or at end of file (the latter event occurs if the user presses `<escape>`): in either case, this predicate succeeds. An error is generated if the `Expression` does not evaluate to an integer.

**Examples**

In most cases, this predicate is used to search for characters which are already represented as character codes, for example in the following call, where the input, "abc@" is consumed, leaving "write( hello ), nl." on the command line:

```prolog
?- ttyskip( 64 ).
|: abc@write( hello ), nl.
yes
```

```
hello
yes
```

**Notes**

This is one of a set of related predicates, including `ttyget/1`, `ttyget0/1`, `ttyskip/1`, `ttyput/1`, `ttytab/1` and `ttynl/0`, which are provided mainly for compatibility with older Prologs. They each perform identical functions to their "non-tty" equivalents, except that they directly read from or write to the console (user input or output device), ignoring the current input or output stream setting. For example, `ttyget/1` and `ttyput/1` are implemented simply as follows:

```prolog
ttyget( Char ) :-
  get( Char ) <~ user.

ttyput( Expr ) :-
  put( Expr ) ~> user.
```

Curiously, the set is incomplete: there are, for example, no equivalent "ttyread/1" or "ttywrite/1" predicates.
ttytab/1

evaluate and write a number of spaces to user output

    ttytab( Expression )

     +Expression     <expr>

Comments
This evaluates the given Expression by calling is/2, and writes the given number of spaces to the console (user output stream); an error is generated if the Expression does not evaluate to an integer in the range [0..8192].

Examples
The following call illustrates the fact that tab/1 performs arithmetic, and computes an expression before outputting the resulting integer number of spaces:

```
?- ttytab( 2 * 4 ), write( hello ), ttynl.
    hello
    yes
```

Notes
The ttytab/1 predicate is useful for displaying tables and other columnar text, although this is most useful when fixed-width fonts (such as the "Prolog fixed" font) are in use. This predicate works by computing the given expression and then calling the formatted write predicate, fwrite/4, to output an empty string (```) with a field width corresponding to the result. Although this approach makes tab/1 very fast, it limits the number of space characters output in a single call to 8192.

This is one of a set of related predicates, including ttyget/1, ttyget0/1, ttyskip/1, ttput/1, ttytab/1 and ttynl/0, which are provided mainly for compatibility with older Prologs. They each perform identical functions to their "non-tty" equivalents, except that they directly read from or write to the console (user input or output device), ignoring the current input or output stream setting. For example, ttyget/1 and ttput/1 are implemented simply as follows:

```
ttyget( Char ) :-
    get( Char ) <~ user.

ttput( Expr ) :-
    put( Expr ) ~> user.
```

Curiously, the set is incomplete: there are, for example, no equivalent "ttyread/1" or "ttywrite/1" predicates.
**type/2**

get or check the type of a term

\[
\text{type( } \text{Term, Type } \text{)}
\]

?Term  \text{<term>}
?Type  \text{<variable> or <integer>}

Comments  This gets or checks the type of the given term, unifying Type to an integer in the range [0..9] according to the type of the given Term, as shown in the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>variable</td>
<td>Var</td>
</tr>
<tr>
<td>1</td>
<td>integer</td>
<td>81</td>
</tr>
<tr>
<td>2</td>
<td>float</td>
<td>3.14159</td>
</tr>
<tr>
<td>3</td>
<td>atom</td>
<td>hello</td>
</tr>
<tr>
<td>4</td>
<td>string</td>
<td><code>the quick brown fox</code></td>
</tr>
<tr>
<td>5</td>
<td>empty list</td>
<td>[]</td>
</tr>
<tr>
<td>6</td>
<td>list</td>
<td>[the,quick,brown,fox</td>
</tr>
<tr>
<td>7</td>
<td>tuple</td>
<td>the(quick,brown,fox</td>
</tr>
<tr>
<td>8</td>
<td>conjunction</td>
<td>(the,quick,brown,fox,More)</td>
</tr>
<tr>
<td>9</td>
<td>disjunction</td>
<td>[the</td>
</tr>
</tbody>
</table>

Examples  The following calls return or check the types of an assortment of terms:

\[
\begin{align*}
?&- \text{type( 123, T ).} \quad \text{<enter>} \\
&\text{T = 1} \quad \text{<enter>}
\end{align*}
\]

\[
\begin{align*}
?&- \text{type( [1,2,3], T ).} \quad \text{<enter>} \\
&\text{T = 6} \quad \text{<enter>}
\end{align*}
\]

\[
\begin{align*}
?&- \text{type( foo, 3 ).} \quad \text{<enter>}
&\text{yes}
\end{align*}
\]
Notes

There are ten internal data types in **WIN-PROLOG**: apart from the unbound variable, there are five atomic types (integer, float, atom, string and empty list) and four compound ones (list, tuple, conjunction and disjunction). These data types are genuinely distinct, and yield significant efficiency compared with simpler Prolog systems that attempt to build lists and conjunctions from standard tuple terms.

In **WIN-PROLOG**, the list and tuple data types are fundamentally identical: in fact, it is the type of the *pointer* to a list sequence that determines whether the data structure is interpreted as a list or tuple. Thus most cases of "univ" (=../2) could be handled by the following definition:

\[
\text{Head}([\text{Tail}]) =.. [\text{Head}|\text{Tail}].
\]

The actual definition takes other cases into account, such as converting single-element lists to atoms, and handling the special cases of conjunctions and "true" disjunctions.

One special feature of **WIN-PROLOG** is its extremely efficient execution of metacalls, and this is helped by two special data types. The conjunction ("(A,B,...)") is not represented as nested two-tuples of the comma (",") functor, but as a dedicated list-like structure. The low-level metacall dispatcher recognises this data type, and recursively executes first its left and then its right argument. Less widely used is the "true" disjunction ("(A|B|...)"), which is similarly recognised. The more complex Edinburgh disjunction ("(A;B;...)") is not specially optimised, because it has to take "if:then:else" into account.

Please note that the special "empty list" atom, "[]", is not a true atom in **WIN-PROLOG**: for efficiency, this system uses a special data type to mark the ends of lists (and tuples). It is therefore not possible to use "[]" to name predicates or files, and it should be noted that the following call fails:

\[
?- [] = '[]'.
\]

There are a couple of dozen "standard" type testing predicates in **WIN-PROLOG**, such as `var/1`, `integer/1`, `atom/1`, and so on: each of these is implemented as a call to `type/2`, with subsequent checking of the returned value where necessary. The latter predicate can be used to provide efficient term type indexing when writing recursive, deterministic term decomposition programs.
unifiable/2

check whether two terms could potentially be unified

unifiable( Term1, Term2 )
+Term1 <term>
+Term2 <term>

Comments  This tests whether the unification of Term1 and Term2 would succeed, without performing the unification or binding any variables.

Examples  The following calls illustrate this predicate, comparing it to standard unification (=/2):

?- unifiable( foo(X), Y(bar) ).
X = _ ,
Y = _

?- foo(X) = Y(bar).
X = bar ,
Y = foo

?- unifiable( foo(123), Y(bar) ).
no

?- foo(123) = Y(bar).
n0

Notes  Effectively, unifiable/2 is true of any two terms for which =/2 is also true, and false for any other pair of terms. The only difference between these two predicates is that while the latter actually performs the unification, binding variables as necessary, the former tests the unification but leaves variables unbound, through a process of "double negation":

unifiable( Term1, Term2 ) :-
\+ \+ Term1 = Term2.

The double negation allows any Prolog goal to be executed, while leaving all variables unbound upon completion. This is because the inner (right-most) "not" (\+/1) fails if the goal succeeds, undoing any variable bindings, before the outer (left-most) "not" in turn succeeds.
unknown/2

get and set the Prolog unknown predicate flag

unknown( Old, New )

?Old : <atom> or <variable>
+New : <atom> in the domain {error,fail}

Comments
This is used to modify the behaviour of Prolog when calling an undefined predicate. The existing "unknown" flag is retrieved, and an attempt is made to unify it with Old: if the match succeeds, the flag is set to the New value accordingly.

Examples
The following calls set the unknown predicate flag to "fail" before trying to execute a non-existent predicate:

?- unknown( U, fail ).
U = error

?- gobbledegook.
no

The next calls restore the original flag setting, and try again:

?- unknown( U, error ).
U = fail

?- gobbledegook.
! ----------------------------------------
! Error 20 : Predicate Not Defined
! Goal     : gobbledegook

Notes
The unknown/2 predicate works by setting one of the Prolog flags, "unknown" (see prolog_flag/3), and is implemented as follows:

unknown( Old, New ) :-
    prolog_flag( unknown, Old, New ).

When the "unknown" flag is set to "fail", WIN-PROLOG mimics early Edinburgh Prologs which failed if an undefined predicate was called. This
archaic behaviour made debugging difficult, and "error" is the preferred (and default) setting for this flag. See unknown_predicate_handler/2 for additional information.
**unknown_predicate_handler/2**

*user-definable predicate for handling undefined predicates*

```
unknown_predicate_handler( Term1, Term2 )
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term1</td>
<td>&lt;term&gt;</td>
</tr>
<tr>
<td>Term2</td>
<td>&lt;term&gt;</td>
</tr>
</tbody>
</table>

**Comments**

If defined by the user, and the "unknown" Prolog flag is set to "error" (see unknown/2 and prolog_flag/3), this is called by the system whenever an undefined predicate is called. The internal error handler unifies the bad call with Term1, and then attempts to call whatever is returned by Term2.

**Examples**

Consider the following definition of this hook:

```
unknown_predicate_handler( foo(X), (write(X), nl) ).
```

Once compiled, it will intercept calls to the non-existent predicate, foo/1, perform a call to write/1 and nl/0 instead:

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

**Notes**

When unknown_predicate_handler/2 has been defined by the user, and the "unknown" predicate flag is set to "error" (this is **WIN-PROLOG**'s default setting), all calls to undefined predicates are "offered" to the present hook, attempting to unify such calls with Term1. If a matching clause is found, a call is made to whatever is returned in Term2: this may be a simple "action", such as the atom, "fail" (which simply calls fail/0, forcing failure) or (as in the example above) a more complex goal. One possible use would be to use this hook to load program modules automatically the first time one of their constituent predicates was called in a given session.

The **WIN-PROLOG** system includes another hook predicate, 'ERROR?/2, which provides similar control over nearly all run-time errors while the development environment is active. In stand-alone applications, any arity-two predicate can be nominated for the task of recovering after an error.

Please note that **WIN-PROLOG** includes a powerful catch-and-throw error handler in addition to the 'ERROR?/2 hook mechanism (see catch/2, catch/3 and throw/2).
**var/1**

test whether the given term is an unbound variable

\[
\text{var}(\ \text{Term}\ )
\]

?Term <term>

**Comments**

This predicate succeeds if the given Term is an unbound variable; otherwise it fails.

**Examples**

The following calls test various cases:

?\- \text{var}(123)\).  \no\n
?\- \text{var}([1,2,3]).  \no\n
?\- \text{var}(\text{foo}).  \no\n
?\- \text{var}(\text{`bar`}).  \no\n
?\- \text{var}(X).  \yes\n
**Notes**

This is one of a comprehensive set of type testing predicates which are commonly used as "guards" in Prolog programs to check the data types of input arguments. In this case, the predicate has an exact opposite: \textit{nonvar/1} is effectively defined in terms of \textit{var/1}, as follows:

\[
\text{nonvar}(\ \text{Term}\ ) \ :- \\
\ \ + \text{var}(\ \text{Term}\ ) .
\]
name and return a list of variables in a term

\[\text{vars}( \text{Term}, \text{Vars} )\]

\[?\text{Term} \quad \text{<term>}\]
\[-\text{Vars} \quad \text{<variable>}\]

Comments
This binds \text{Vars} to a list containing all the variables in the given \text{Term}, complete with automatically-generated names. Each element of the list is a conjunction of (\text{<atom>}, \text{<variable>}), containing the chosen name and variable respectively.

Examples
Consider the following program, which uses \text{vars/2} to generate a list of names and variables for a given term, before writing the term with these names, via \text{eprint/2}:

\[
\text{pretty_print}( \text{Term} ) :-
\quad \text{vars}( \text{Term}, \text{Vars} ),
\quad \text{eprint}( \text{Term}, \text{Vars} ),
\quad \text{nl}.
\]

Once compiled, this can be used to display terms with alphabetic variable names, rather than the usual "_nnn" addresses:

\[
?- \text{pretty_print}( \text{foo(Var1,Var2,Var1)} ) . \quad <\text{enter}>\]
\[
\text{foo}(\text{A,}_,\text{A})
\quad \text{Var1} = _ ,
\quad \text{Var2} = _
\]

Notes
The \text{vars/2} predicate provides several related functions. Firstly, it can be used to extract a list of variables from a term for linear processing, avoiding the need to recurse over a complex compound structure. Secondly, it can be used to name variables for "pretty" output (see the above example). A third, less obvious use is in the identification of single-use variables in terms, which is helped by this predicate's naming algorithm.

The extraction and naming of variables occurs in two passes: the first simply recurses over the term, collecting each variable into a table, and counting its number of occurrences; the second passes over the list, assigning names. All variables which occur only once in a term are given the name of the "anonymous variable" ("_"), while others are named using the sequence: [\text{A..Z}, \text{AA..AB..ZZ}, \text{AAA..BAA..2ZZ}, ...].
**Comments**

This writes the **WIN-PROLOG** welcome banner to the current output stream, showing version and license information, together with a summary of the amount of space in the nine main memory areas (see `free/9`) after performing a text garbage collection (see `gc/1`). The *Mode* flag simply determines whether or not "tramlines" are printed around the banner (values of "1" and "0" respectively).

**Examples**

The following call displays the welcome banner with tramlines (*Mode = 1*):

```prolog
?- ver(1).
```

```
---------------------------------------------------
BDS WIN-PROLOG 7.000 X86 S/N 0020596320 28 Feb 2019
Copyright 1989-2019 Brian D Steel (www.solanum.org)
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=256 T=2048 P=8192 S=64 I=256 O=256
---------------------------------------------------
yes
```

**Notes**

The main purpose of the `ver/1` predicate is to confirm version and licensing information when submitting technical support queries to LPA: it is extremely useful to our helpdesk to have the complete output of the above call included with any email or fax in which a problem is discussed. This information can also be seen in the standard "about box" (see the "Help/About" menu item), but cannot be copied from this dialog into the Windows clipboard.

Two version predicates are built directly into **WIN-PROLOG**: `ver/1` performs output of the welcome banner, which includes all main version details, while `ver/4` provides programmable access to the name, version, compilation date and serial number of the system. Two further predicates, `version/0` and `version/1`, provide additional version information, including support for user-defined version strings.

With the release of the X64 (64-bit) version of **WIN-PROLOG** 7.000 alongside the existing X86 (32-bit) system, two additional values were added to `ver/1`: if "52" is specified, it will succeed in the X86 version of Prolog, but fail in X64; conversely, if "64" is specified, it fails in X86 but succeeds in X64. This topic is discussed further in **Appendix U**.
\textbf{ver}/4

\textit{return Prolog version information}

\begin{verbatim}
ver( Name, Version, Date, Serial )
\end{verbatim}

\begin{itemize}
\item \texttt{-Name} \hspace{1cm} \texttt{<variable>}
\item \texttt{-Version} \hspace{1cm} \texttt{<variable>}
\item \texttt{-Date} \hspace{1cm} \texttt{<variable>}
\item \texttt{-Serial} \hspace{1cm} \texttt{<variable>}
\end{itemize}

\textbf{Comments} \hspace{1cm} This returns four atoms, respectively representing the \textit{Name}, \textit{Version}, compilation \textit{Date} and \textit{Serial} number of the current instance of Prolog.

\textbf{Examples} \hspace{1cm} The following call returns information about the current instance of Prolog:

\begin{verbatim}
?- ver( N, V, D, S ).
N = 'WIN-PROLOG' ,
V = '6.100' ,
D = '14 Jul 2017' ,
S = '0123456789'
\end{verbatim}

\textbf{Notes} \hspace{1cm} The \textit{ver}/4 predicate is useful in programs which must run on more than one version of \texttt{WIN-PROLOG}, or on other BDS Prologs such as \texttt{DOS-PROLOG} or \texttt{MacProlog32}, allowing runtime decisions to be made regarding system-dependent features.

Two version predicates are built directly into \texttt{WIN-PROLOG}: \textit{ver}/1 performs output of the welcome banner, which includes all main version details, while \textit{ver}/4 provides programmable access to the name, version, compilation date and serial number of the system. Two further predicates, \texttt{version/0} and \texttt{version/1}, provide additional version information, including support for user-defined version strings.
version/0

output an extended version banner

version

Comments  This writes an extended version of the WIN-PROLOG welcome banner to the current output stream, showing version and license information, together with a summary of the amount of space in the nine main memory areas (see free/9) after performing a text garbage collection (see gc/1), suffixed with optional user version strings (see version/1).

Examples  The following call displays the extended welcome banner:

? - version.
BDS WIN-PROLOG 7.000 X64 S/N 0020596320 28 Feb 2019
Prolog Overlay 7.016 Compiled And Built 28 Feb 2019
Copyright 1989-2019 Brian D Steel (www.solanum.org)
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=256 T=2048 P=8192 S=64 I=256 O=256
yes

Notes  Two version predicates are built directly into WIN-PROLOG: ver/1 performs output of the welcome banner, which includes all main version details, while ver/4 provides programmable access to the name, version, compilation date and serial number of the system. Two further predicates, version/0 and version/1, provide additional version information, including support for user-defined version strings.
version/1

add a term to extended version banner

\[ \text{version( Term )} \]

?Term \hspace{1cm} <\text{term}>

Comments This appends the given Term to the set of information displayed in the extended version of the WIN-PROLOG welcome banner (see version/0).

Examples The following call adds a term to the extended version banner:

\[ \text{?- version( `the quick brown fox` ).} \]

yes

The extended banner now includes this new term:

\[ \text{?- version.} \]

------------------------------------------------------------------------------------------------------------------------------
BDS WIN-PROLOG 7.000 X64 S/N 0020596320 28 Feb 2019
Prolog Overlay 7.016 Compiled And Built 28 Feb 2019
Copyright 1989-2019 Brian D Steel (www.solanum.org)
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=256 T=2048 P=8192 S=64 I=256 O=256
------------------------------------------------------------------------------------------------------------------------------

the quick brown fox
yes

Notes Two version predicates are built directly into WIN-PROLOG: ver/1 performs output of the welcome banner, which includes all main version details, while ver/4 provides programmable access to the name, version, compilation date and serial number of the system. Two further predicates, version/0 and version/1, provide additional version information, including support for user-defined version strings.

While version/1 makes it possible to add lines to the extended banner, there is no way of removing them other than by starting a new Prolog session. Programs which use this predicate should therefore take precautions to ensure that they only call it once per session.
volatile/1

declare predicates that should not be saved in applications

volatile( Preds )
+Preds <pred_specs>

Comments
This declares one or more predicates, specified in Preds, as "volatile", which means that they will be omitted from any application overlay generated by the "Run/Application" menu item. Such predicates can, however, be saved in object files by save_predicates/2.

Examples
The following call ensures that any predicate called foo/1 will be omitted from a saved application overlay:

?- volatile( foo/1 ).
   yes

A number of predicates can be declared in a single call, using either a list or a conjunction:

   yes

?- volatile( (bar/2,sux/3,you/4) ).
   yes

Because "volatile" is defined as a prefix operator with a very high precedence, it is also possible to declare predicates using the following syntax:

?- volatile foo/1 .
   yes

?- volatile bar/2, sux/3, you/4.
   yes

Notes
The volatile/1 predicate does not affect WIN-PROLOG object (.PC) files, because these are created with save_predicates/2, whose first argument is a list of predicates to be saved. Its only function is to avoid specific predicates being saved in an application: however, it is usually preferable simply to abolish any such predicates before saving the overlay (see abolish/1 and the "Run/Application" menu item).
**wait/1**

*yield to Windows unless a message is waiting*

```prolog
wait( Flag )
```

*+Flag* 

<integer> in the domain {0,1} or <conjunction>

**Comments**

This sets the Windows message flag to "1" (see `flag/1`), and then checks the WIN-PROLOG Windows "message queue". If there is at least one message in the queue, control is transferred to the system "message hook" (see `message_hook/4`), allowing queued messages to be processed; if not, control is yielded to Windows, for a duration specified by the `Flag`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>wait up to 55ms for a message</td>
</tr>
<tr>
<td>1</td>
<td>wait indefinitely for a message</td>
</tr>
</tbody>
</table>

**Examples**

Many WIN-PROLOG applications use the following "idle loop" to suspend active processing while waiting for messages:

```prolog
wait_for_messages :-
    repeat,
    wait( 0 ),
    fail.
```

**Notes**

When the Windows message flag is set (has the value "1" - see `flag/1`), WIN-PROLOG checks its message queue once every 256 predicate calls, or repeatedly during console (user) input. The `wait/1` predicate is a special case, being the only non-input predicate which can be *interrupted*, as opposed to *preempted*, by a Windows message: provided that idle loops such as the example above contain fewer than 256 predicate calls, messages are guaranteed to interrupt `wait/1`.

As a special extension, `Flag` can be specified as a conjunction of (<integer>,<variable>): the integer is interpreted just as above, and the variable is ignored. While this might seem strange, any message hook that picks up an interrupted call to `wait/1` receives a copy of the entire goal, and can therefore "see" and, if required, bind the variable. It is this exact mechanism that is used to return values from modal dialogs: `call_dialog/2` issues the call "wait( (1,X) )", and waits for "X" to be bound by the system message hook.
wait_dialog/2

call a semi-modal dialog and get or check the result

    wait_dialog(Window, Result)

+Window <window_handle>
?Result <term>

Comments
This calls an already visible dialog Window, and waits for the user to complete the dialog: Result is unified with whatever value has been returned by the dialog’s "window handler".

Examples
The following program creates a very simple dialog with two buttons, labelled "Hello" and "World" respectively:

    create_foo :-
        Dstyle = [ws_popup,ws_caption,dlg_ownedbyprolog],
        Bstyle = [ws_child,ws_visible,bs_pushbutton],
        wdcreate( foo, `foo`, 100, 100, 170, 75, Dstyle ),
        wccreate( (foo,1), button, `Hello`, 10, 10, 70, 30, Bstyle ),
        wccreate( (foo,2), button, `World`, 90, 10, 70, 30, Bstyle ).

When compiled, and the following command is entered, the dialog "foo" is created, but not yet displayed:

    ?- create_foo.
    yes

To show the dialog modelessly, make the following call:

    ?- show_dialog( foo ).
    yes

The dialog will appear, with its two buttons: at this point, clicking the buttons will have no effect. To wait for and retrieve a button press, enter the following command:

    ?- wait_dialog( foo, X ).

If the user presses the "Hello" button, its lowercase text will be returned as follows:

\[ X = \text{hello} \]

Unlike modal dialogs, which handled by `call_dialog/2`, this dialog remains on the screen even after a button click. It is therefore possible to call the dialog again to retrieve further button clicks:

?- `wait_dialog( foo, X ).`
\[ X = \text{world} \]

When it is finally desired to hide the dialog, the following call can be made:

?- `hide_dialog( foo ).`
\[ yes \]

**Notes**

The example shown above uses the default window handler, `window_handler/4`, which simply binds the lowercase text of any clicked button to the `Result`. It is the action of binding this variable which actually "completes" the dialog.

Window handlers are simply arity-4 programs which intercept messages destined for a window (including dialogs and their controls), allowing arbitrary actions to take place. Consider the following example:

\[
\text{foo_handler( (foo,1), msg_button, _, Result ) :-}
\text{ \hspace{1cm} time( _, _, _, H, M, S, _ ),}
\text{ \hspace{1cm} Result = \text{finished_at}(H,M,S).}
\]

\[
\text{foo_handler( Window, Message, Data, Result ) :-}
\text{ \hspace{1cm} window_handler( Window, Message, Data, Result ).}
\]

The first clause detects that a button called "(foo,1)" has been clicked, and completes the dialog by returning the structure "finished_at(H,M,S)" rather than the atom "hello". All other messages are passed to the default window handler. Once compiled, the new window handler is attached to the dialog window with a call to `window_handler/2`, as shown here:

?- `window_handler( foo, foo_handler )`. 
\[ yes \]

If the dialog is invoked once again, and the "(foo,1)" button (labelled "Hello") is pressed, the time will be returned instead of the atom
"hello":

?- show_dialog( foo ).
yes

?- wait_dialog( foo, X ).
X = finished_at(10,8,42)

?- hide_dialog( foo ).
yes

Please note that "modal" and "modeless" dialogs have exactly the same window handlers: in fact, in WIN-PROLOG, these two types of dialog are one and the same thing as each other. All that determines whether a dialog behaves in a modal or modeless fashion at run-time is whether it is invoked by call_dialog/2 (modal) or show_dialog/1 (modeless). A third class of dialog, sometimes referred to as a "wizard", behaves semi-modally: like a modal dialog, it waits for input from the user when called with wait_dialog/2, but like a modeless dialog, it remains visible and in focus between successive calls, until finally hidden by invoking hide_dialog/1.
warea/5

get or check the position and size of a window's client area

\[
\text{warea( Window, Left, Top, Width, Depth )}
\]

+Window <window_handle>
?Left   <integer> or <variable>
?Top    <integer> or <variable>
?Width  <integer> or <variable>
?Depth  <integer> or <variable>

Comments
This gets the position and size of a given Window's client area, unifying its \((X,Y)\) screen coordinates with Left and Top, and its \((W,D)\) dimensions with Width and Height. All numerical values are in pixel units, where \((0,0)\) is the left top of the screen.

Examples
Running on an XGA resolution screen (1024*768), the following call picks up the position and size of the WIN-PROLOG "main window" (window '0') client area:

\[
? \text{- warea( 0, L, T, W, D ).} \quad \text{<enter>}
\]

L = 4 ,
T = 42 ,
W = 1012 ,
D = 650

Notes
The warea/5 predicate can be used to get or check the position and size of a window's client area, but not to alter it: for this purpose it is necessary to use the related predicate, wsize/5.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wbtnsel/2**

get or set the selection state of a button window

\[
\text{wbtnsel( Window, State )}
\]

+Window <window_handle>

?State <integer> or <variable>

**Comments**  
This gets or sets the selection State of a given "button" Window. Where State is an unbound variable, it returns the current button state; where it is an integer, it sets the button's state; in either case, State is defined as follows:

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>button is not selected</td>
</tr>
<tr>
<td>1</td>
<td>button is selected</td>
</tr>
</tbody>
</table>

**Examples**  
If there exists a dialog called "foo" which contains a "checkbox" button with an ID of "1"; the following call will return its current state (this example assumes the button was checked):

\[
?- \text{wbtnsel( (foo,1), S )}.  
S = 1
\]

A button's state can be modified by specifying the state; the next call, for example, ensures the same button is unchecked:

\[
?- \text{wbtnsel( (foo,1), 0 )}.  
yes
\]

**Notes**  
This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful **winapi/4** to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, **winapi/4** and its helper predicate, **wintxt/4**, can be used to access virtually any function in the **Windows API** as well as those implemented in third-party DLLs.
**wccreate/8**

create a control window

```
wccreate( Window, Class, Text, Left, Top, Width, Depth, Style )
```

+Window <window_handle>
+Class <atom>
+Text <string>
+Left <integer>
+Top <integer>
+Width <integer>
+Depth <integer>
+Style <list>

**Comments**  
This creates a "control window" with the given Window handle, Class, Text, (Left,Top) position, (Width,Depth) size and Style list. See wcreate/8 for a list of class types, and Appendix C for more information about style lists.

**Examples**  
The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single "default push" button:

```
foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Bstyle = [ws_visible,ws_child,bs_defpushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen (subsequently clicking the button will hide (but not close) the dialog):

```
?- foo.  <enter>
yes
```

**Notes**  
This is one of a family of WIN-PROLOG window creation predicates, implemented in terms of the lower-level predicate, wcreate/8, which expects an integer style value as its final argument. The wccreate/8 and wdcreate/7 predicates expect Style to be specified as a list of atoms, each denoting a style names, which are converted into the style integer; wtcreate/6 and wucreate/6 have no Style parameter, and simply use a default style integer: all four of these then call wcreate/8 with their computed or default style.
wclass/2

get or check the class name of a window

wclass( Window, Class )

+Window <window_handle>
?Class <variable> or <atom>

Comments This unifies Class with an atom representing the class name of a given Window. The following table lists the standard classes of windows which can be created by WIN-PROLOG:

<table>
<thead>
<tr>
<th>Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>button control</td>
</tr>
<tr>
<td>ComboBox</td>
<td>combobox control</td>
</tr>
<tr>
<td>Edit</td>
<td>edit control</td>
</tr>
<tr>
<td>ListBox</td>
<td>listbox control</td>
</tr>
<tr>
<td>MDIClient</td>
<td>MDI client control</td>
</tr>
<tr>
<td>ScrollBar</td>
<td>scrollbar control</td>
</tr>
<tr>
<td>Static</td>
<td>static control</td>
</tr>
</tbody>
</table>

In addition to the above, the following classes are defined by WIN-PROLOG itself:

<table>
<thead>
<tr>
<th>Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>WIN-PROLOG main window</td>
</tr>
<tr>
<td>Cons</td>
<td>WIN-PROLOG console window</td>
</tr>
<tr>
<td>Text</td>
<td>WIN-PROLOG text window</td>
</tr>
<tr>
<td>User</td>
<td>WIN-PROLOG user window</td>
</tr>
<tr>
<td>Dialog</td>
<td>WIN-PROLOG enhanced dialog window</td>
</tr>
<tr>
<td>Rich</td>
<td>WIN-PROLOG enhanced rich edit control</td>
</tr>
<tr>
<td>Grafix</td>
<td>WIN-PROLOG grafix control</td>
</tr>
<tr>
<td>NudgeBar</td>
<td>WIN-PROLOG enhanced up-down control</td>
</tr>
</tbody>
</table>
### Examples

The following call returns the class name of WIN-PROLOG's main window ("0"):

```
?- wclass( 0, C ).
C = 'Main'
```

Notice that when checking a window's class, the name is not case sensitive:

```
?- wclass( 0, 'mAIn' ).
yes
```

### Notes

The `wclass/2` predicate returns window class names using the exact capitalisation shown in the table above, but it is case insensitive when checking names: this anomaly is due to the design of Microsoft Windows. The "MDIClient" control class cannot be used directly within WIN-PROLOG applications, although the client area of the "main window" ("(0,1)") is in fact such a control window.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wclose/1

close a window

\[ \text{wclose}( \text{Window} ) \]

\(+\text{Window} <\text{window\_handle}>\)

Comments

This closes the named Window, freeing its resources.

Examples

The following call creates a small "user" (MDI child) window:

\[ ?- \text{wcreate}( \text{foo, user, ``, 100, 100, 300, 200, 0 } ). \]

\(<\text{enter}> \text{yes} \)

After clicking the mouse back on the "console window", the newly created window can be closed with the next command:

\[ ?- \text{wclose}( \text{foo} ). \]

\(<\text{enter}> \text{yes} \)

Notes

When a "top level" window (such as a dialog or user window) is closed, all of its constituent control windows are also closed: thus there may be many calls to \text{wcreate/8} for each subsequent call to \text{wclose/1}. The number of windows which may be open at any one time is a function of the spare resources in the current Windows environment itself. There is, however, a fixed ceiling in \text{WIN-PROLOG} that limits the number of simultaneously open dialogs to 64: however, because dialogs can be created and closed "on the fly" any number of times, this limit should never pose a problem.

This is one of a small family of \text{WIN-PROLOG} window handling predicates directly implemented in the assembler-coded kernel, enabling special low-level housekeeping functions to be handled efficiently. Most of the other window handling predicates are implemented as a "thin layer" over the all-powerful \text{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \text{winapi/4} and its helper predicate, \text{wintxt/4}, can be used to access virtually any function in the \text{Windows API} as well as those implemented in third-party DLLs.
wcmbadd/4

*add an item to a combobox control*

\[
\text{wcmbadd}( \text{Window, Position, String, Item })
\]

+Window <window_handle>
+Position <integer>
+String <string>
+Item <integer>

**Comments**
This adds a *String*, together with a user-defined numerical *Item*, to a "combobox" control *Window* at the specified *Position*. The first entry in the control is numbered zero ("0"); if *Position* is set to "-1", the *String* is inserted at its default position, which depends upon the combobox style (see Appendix C).

**Examples**
Consider a dialog called "foo" that includes a combobox control with an ID of 9, the "cbs_sort" style, and the following items:

```
aardvark 314
badger   273
dingo    981
```

The following call will insert the string, "cheetah", with a user-defined item, "123", at its default (sorted) location between "badger" and "dingo", maintaining alphabetical order:

```
?- wcmbadd( (foo, 9 ), -1, `cheetah`, 123 ).
yes
```

The next call inserts the string, "zebra", together with the item, "456", but forces it to become the second entry, immediately after "aardvark", even though this spoils the alphabetical order:

```
?- wcmbadd( (foo, 9 ), 1, `zebra`, 456 ).
yes
```

**Notes**
This is one of a family of *WIN-PROLOG* window handling predicates, implemented as a "thin layer" over the all-powerful *winapi/4* to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, *winapi/4* and its helper predicate,
\textit{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
wcmbdel/2

delete an item from a combobox control

\[
\text{wcmbdel}( \text{Window, Position} )
\]

+Window <window_handle>
+Position <integer>

Comments
This deletes an entry in a "combobox" control Window from the specified Position. The first entry in this type of control is numbered zero ("0").

Examples
Consider a dialog called "foo" that includes a combobox control with an ID of 9 and containing the following items:

aardvark 314
badger    273
dingo     981

The following call will delete the second entry ("badger"):

?- wcmbdel( (foo,9 ), 1 ).
   yes

Notes
This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
find an item in a combobox control

wcmbfnd( Window, Position, String, Match )

+Window <window_handle>
+Position <integer>
+String <string>
-Match <variable?>

Comments
This searches for a String in a "combobox" control Window, starting one place after the specified Position. The first entry in this type of control is numbered zero ("0"), so in order to search from the top of the list, Position should be set to "-1". The position of the first string whose first characters match the String is bound to Match.

Examples
Consider a dialog called "foo" that includes a combobox control with an ID of 9 and containing the following items:

- aardvark 314
- badger 273
- dingo 981

The following call will search from the top of the list for an entry that begins with the letters, "BaD". Note that the search is not case-sensitive, and so will match "badger", at position ("1"):

?- wcmbfnd( foo, 9 ), -1, `BaD`, M ).
M = 1

Notes
This predicate does not perform any side effects; it is used for the gathering of information only; however, the returned Match parameter may be passed directly into wcmbget/4 or wcmbset/2 in order to obtain or select the matched string.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wcmbget/4**

*get an item from a combobox control*

```
wcmbget( Window, Position, String, Item )
```

+Window <window_handle>
+Position <integer>
-String <variable>
-Item <variable>

**Comments**

This gets an item from a "combobox" control Window at the specified Position, and binds its text to String and user-defined item value to Item. The first entry in this type of control is numbered zero ("0").

**Examples**

Consider a dialog called "foo" that includes a combobox control with an ID of 9 and containing the following items:

- aardvark 314
- badger 273
- dingo 981

The following call will pick up the second entry ("badger", with its item value):

```
?- wcmbget( (foo,9 ), 1, S, I ).
S = `badger`,
I = 273
```

**Notes**

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wcmbsel/2**

*get or set the selection in a combobox control*

```
   wcmbsel( Window, Position )
```

+Window        <window_handle>
+Position       <integer>

**Comments**

This gets or sets the selection `Position` in a "combobox" control `Window`. The first entry in this type of control is numbered zero ("0"). If `Position` is an unbound variable, it returns the currently selected entry; if `Position` is an integer, it selects the indicated entry accordingly. A `Position` of "-1" can be used to clear the selection, and if this value is returned, it indicates that no entry is currently selected.

**Examples**

Consider a dialog called "foo" that includes a combobox control with an ID of 9 and containing the following items:

```
n  aardvark  314
   badger    273
   dingo     981
```

The following call will select the second entry ("badger"):

```prolog
?- wcmbsel( (foo,9 ), 1 ).
```

```
yes
```

Suppose the user has clicked on the "dingo" entry; the following call will return this selection:

```prolog
?- wcmbsel( (foo,9 ), P ).
P = 2
```

**Notes**

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
**wcount/4**

get or check char, word and line counts in a given window

```prolog
wcount( Window, Chars, Words, Lines )
```

- **Window**: <window_handle>
- **Chars**: <variable> or <integer>
- **Words**: <variable> or <integer>
- **Lines**: <variable> or <integer>

**Comments**
This counts the number of characters, words and lines of text in the given Window, and unifies the results with Chars, Words and Lines respectively.

**Examples**
Consider a "text" window called "foo" whose edit control contains the following text:

```
'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.
```

The following call will return information about this text:

```
?- wcount( (foo,1), C, W, L ).
C = 134 ,
W = 25 ,
L = 4
```

**Notes**
The word counting algorithm used by wcount/4 considers a white-space delimited block of printable characters to be a "word": beyond this, it does not perform any lexical analysis. In particular, it does not distinguish between alphanumeric characters and punctuation, so, for example, "good/bad" is treated as one word, not two (or three).

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
create a window

wcreate( Window, Class, Text, Left, Top, Width, Depth, Style )

+Window <window_handle>
+Class <atom>
+Text <string>
+Left <integer>
+Top <integer>
+Width <integer>
+Depth <integer>
+Style <integer>

Comments
This creates a window with the given Window handle, Class, Text, (Left,Top) position, (Width,Depth) size and Style integer. The following standard window classes may be created by this predicate:

<table>
<thead>
<tr>
<th>Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>button control</td>
</tr>
<tr>
<td>ComboBox</td>
<td>combobox control</td>
</tr>
<tr>
<td>Edit</td>
<td>edit control</td>
</tr>
<tr>
<td>ListBox</td>
<td>listbox control</td>
</tr>
<tr>
<td>ScrollBar</td>
<td>scrollbar control</td>
</tr>
<tr>
<td>Static</td>
<td>static control</td>
</tr>
</tbody>
</table>

In addition to the above, the following classes, which are defined by WIN-PROLOG itself, can also be created:

<table>
<thead>
<tr>
<th>Class</th>
<th>Base Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cons</td>
<td>WIN-PROLOG</td>
<td>WIN-PROLOG console window</td>
</tr>
<tr>
<td>Dialog</td>
<td>#32770</td>
<td>WIN-PROLOG enhanced dialog window</td>
</tr>
<tr>
<td>Grafix</td>
<td>WIN-PROLOG</td>
<td>WIN-PROLOG grafix control</td>
</tr>
<tr>
<td>Main</td>
<td>WIN-PROLOG</td>
<td>WIN-PROLOG main window</td>
</tr>
</tbody>
</table>
Examples

The following call creates a small "user" (MDI child) window:

```prolog
?- wcreate( foo, user, ``, 100, 100, 300, 200, 0 ).
yes
```

After clicking the mouse back on the "console window", the newly created window can be closed with the next command:

```prolog
?- wclose( foo ).
yes
```

Notes

The `wcreate/8` predicate provides low-level support for four other window creators: the `wccreate/8` and `wdcreate/7` predicates treat `Style` as a list of style names, converting them into an integer, while `wtcreate/6` and `wucreate/6` have no style parameter, and use a default integer. See Appendix C for information about Window styles.

This is one of a small family of WIN-PROLOG window handling predicates directly implemented in the assembler-coded kernel, enabling special low-level housekeeping functions to be handled efficiently. Most of the other window handling predicates are implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wdcreate/7

create a dialog window

\[
\text{wdcreate}( \text{Window, Text, Left, Top, Width, Depth, Style } )
\]

+Window <window_handle>
+Text <string>
+Left <integer>
+Top <integer>
+Width <integer>
+Depth <integer>
+Style <list>

Comments This creates a "dialog window" with the given Window handle, Text (title), (Left, Top) position, (Width, Depth) size and Style list. See Appendix C for more information about style lists.

Examples The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single "default push" button:

```
foo :-
    Dstyle = [ws_visible, ws_popup, ws_caption],
    Bstyle = [ws_visible, ws_child, bs_defpushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen (subsequently clicking the button will hide (but not close) the dialog):

```
?- foo.
yes
```

Notes This is one of a family of WIN-PROLOG window creation predicates, implemented in terms of the lower-level predicate, wcreate/8, which expects an integer style value as its final argument. The wccreate/8 and wdcreate/7 predicates expect Style to be specified as a list of atoms, each denoting a style names, which are converted into the style integer; wcreate/6 and wucreate/6 have no Style parameter, and simply use a default style integer: all four of these then call wcreate/8 with their computed or default style.
**wdict/2**

return a list of currently defined top-level windows

```prolog
wdict( Flag, Dict )
```

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

**Comments**
This returns a list of all the currently defined "top-level windows" (dialog, text and user windows), as specified in the given Flag (see `dict/2`), and binds it to the variable Dict.

**Examples**
The following call creates a small "user" (MDI child) window:

```prolog
?- wcreate( foo, user, ``, 100, 100, 300, 200, 0 ).
yes
```

After clicking the mouse back on the "console window", the newly created window will be included along with any other top-level windows in the list returned by the next command (in this example, "foo" is the only such window):

```prolog
?- wdict( 0, D ).
D = [foo]
```

The following call can be used to close all top level windows:

```prolog
?- wdict( 0, D ), forall( member(W,D), wclose(W) ).
```

**Notes**
This is one of a small family of WIN-PROLOG window handling predicates directly implemented in the assembler-coded kernel, enabling special low-level housekeeping functions to be handled efficiently. Most of the other window handling predicates are implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
perform or check a clipboard function in an edit control window

`wedtclp( Window, Function )`

+Window <window_handle>
+Function <integer> in the range {+/- 1..5}

Comments

This performs or checks for the given standard clipboard `Function` with respect to the given edit `Window`. When `Function` is a positive integer, it performs one of the following clipboard functions:

<table>
<thead>
<tr>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cut</td>
</tr>
<tr>
<td>2</td>
<td>Copy</td>
</tr>
<tr>
<td>3</td>
<td>Paste</td>
</tr>
<tr>
<td>4</td>
<td>Clear</td>
</tr>
<tr>
<td>5</td>
<td>Undo</td>
</tr>
</tbody>
</table>

When `Function` is a negative integer, this predicate succeeds or fails depending upon whether the corresponding (positive) function would succeed if performed.

Examples

Consider a dialog called "foo", containing an edit control with an ID of 10, which in turn contains the following text:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

The following call will select the whole of its first line:

```prolog
?- wedtsel( (foo,10), 0, 35 ).  
yes
```

Once selected, this text can be copied from "foo" into the Windows clipboard, and then pasted from this into the "console window" with
the next command:

```
?- wedtclp((foo,10), 2), wrchclp((1,1), 3), nl.
'Twas brillig, and the slithy toves
yes
```

Notes

The `wedtclp/2` predicate can only perform clipboard functions using the standard "text" clip type, and only works with "edit" controls. Other predicates, such as `wtext/2`, can be used to process the text in other classes of window.

Note the use of the related predicate, `wrchclp/2`, to process the console window in the above example: the console, in common with "text" windows, uses a "rich" control, rather than an "edit" control, and so must be processed by the `wrch*/n` family of predicates.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wedtfnd/6

find a text string in an edit control window

\[ \text{wedtfnd}( \text{Window}, \text{From}, \text{To}, \text{String}, \text{Start}, \text{Finish} ) \]

\begin{itemize}
  \item [+Window] \text{<window_handle>}
  \item [+From] \text{<integer>}
  \item [+To] \text{<integer>}
  \item [+String] \text{<string>}
  \item [?Start] \text{<variable> or <integer>}
  \item [?Finish] \text{<variable> or <integer>}
\end{itemize}

Comments
This searches edit control \text{Window} for the given text \text{String} between the offsets \text{From} and \text{To}, unifying the start and finish offsets of any match with \text{Start} and \text{Finish} respectively. As a special case, the \text{String} may be specified as the empty string (``), in which case the Top and End of the next space-delimited token is returned. This predicate fails if no match is found within the specified limits.

Examples
Consider a dialog called "foo", containing an edit control with an ID of 10, which in turn contains the following text:

\[ \text{'Twas brillig, and the slithy toves}
  \text{ Did gyre and gimble in the wabe;}
  \text{ All mimsy were the borogoves,}
  \text{ And the mome raths outgrabe.} \]

The following call returns the start and finish offsets of the first instance of the word, "gimble":

\[ ?- \text{wedtfnd}( (\text{foo,10}), 0, 9999, `\text{gimble`}, S, F ). \]
\[ S = 51 , \]
\[ F = 57 \]

Notes
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned \text{Start} and \text{Finish} parameters may be passed directly into \text{wedtsel/3} in order to highlight the matched string.

Please note that \text{wedtfnd/6} only performs exact-case (case sensitive) searches. The \text{find/3} predicate, on the other hand, can be used to perform both case sensitive and case insensitive searches, and thanks to \text{WIN-PROLOG}'s string and memory file input/output predicates, the latter can easily be used in conjunction with text searches within windows.
wedtlin/4

`get or check the offsets of a line in an edit control window`

```
wedtlin( Window, Offset, Start, Finish )
```

+Window       <window_handle>
+Offset       <integer>
?Start        <variable> or <integer>
?Finish       <variable> or <integer>

Comments
This picks up the start and finish offsets of the line of text which includes the given Offset within the given edit Window, and unifies these with Start and Finish respectively. The returned values include everything on the given line, excluding the terminating `<cr>` `<lf>` pair.

Examples
Consider a dialog called "foo", containing an edit control with an ID of 10, which in turn contains the following text:

```
'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.
```

The following call returns the start and finish offsets of the line of text which includes character offset "81":

```
?- wedtlin( (foo,10), 81, S, F ).
S = 72 ,
F = 101
```

Notes
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned Start and Finish parameters may be passed directly into wedtsel/3 in order to highlight the matched line.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, winxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
convert between offsets and (x,y) coords in an edit window

\[ \text{wedtpxy( Window, Offset, Xpos, Ypos )} \]

Comments
This converts between linear offsets and (x,y) coordinates in the given edit Window. If Offset is an unbound variable, Xpos and Ypos must be bound to integers which specify the column and row coordinates of a character in the Window: its linear offset is computed and bound to Offset. If Offset is an integer, the corresponding (x,y) coordinates of the indicated character are computed and unified with Xpos and Ypos respectively.

Examples
Consider a dialog called "foo", containing an edit control with an ID of 10, which in turn contains the following text:

'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
   All mimsy were the borogoves,
   And the mome raths outgrabe.

The following call returns the offset of character number "20" in line "2" (this is the "b" in "borogoves"):

? - \text{wedtpxy( (foo,10), P, 20, 2 )}. \hspace{1cm} <\text{enter}> \\
\text{P} = 92 \\

The next call performs the reverse operation:

? - \text{wedtpxy( (foo,10), 92, X, Y )}. \hspace{1cm} <\text{enter}> \\
\text{X} = 20 , \\
\text{Y} = 2 \\

Notes
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned Offset parameter may be passed directly into wedtsel/3 in order to highlight the start or end of a text block. When converting from (X,Y) to linear offset,
both values are automatically limited to their maximum values for the given text line (Xpos) or line number (Ypos). Please note that all offsets are numbered from zero ("0").

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful *winapi/4* to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, *winapi/4* and its helper predicate, *wintxt/4*, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
get or set the selection area in an edit control window

\[ \text{wedtsel}( \text{Window}, \text{Start}, \text{Finish} ) \]

| +Window | <window_handle> |
| ?Start  | <variable> or <integer> |
| ?Finish | <variable> or <integer> |

Comments
This gets or sets the Start and Finish offsets of the text selection in the given edit Window. If either or both of Start and Finish is an unbound variable, it is bound to an integer containing the current value; otherwise, the edit window is side-effected to set the Start and/or End of its selection accordingly.

Examples
Consider a dialog called “foo”, containing an edit control with an ID of 10, which in turn contains the following text:

'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.

The following call will select the whole of the third line:

\[ ?- \text{wedtsel}( \text{foo}, 10 ), 72, 101 ). \]

yes

Notes
The wedtsel/3 predicate is used together with wedtfnd/6, wedtln/4 and wedtpxy/4 to change or query the text selection area; together with wedttxt/2 it can also be used to replace selected portions of text in edit control windows.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wedttxt/2

get or set the selected text in an edit control window

\[ \text{wedttxt}( \text{Window, Text} ) \]

+Window <window_handle>
+Text <variable> or <string>

Comments
This gets or sets the Text contained within current selection area of the given edit Window. If Text is an unbound variable, it is bound to a string containing the current text in the selection area; otherwise, the edit window is side-effected to replace the currently selected text with the given Text string.

Examples
Consider a dialog called "foo", containing an edit control with an ID of 10, which in turn contains the following text:

'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.

The following call will select and return the word "brillig":

?- wedtsel( (foo,10), 6, 13 ), wedttxt( (foo,1), T ).
T = `brillig`

The next call replaces this word with "squirmy":

?- wedttxt( (foo,10), `squirmy` ).
yes

Notes
The wedttxt/2 predicate is used together with wedtfnd/6, wedtlin/4, wedtpxy/4 and wedtsel/3, to locate, select and replace selected text in edit control windows; wtext/2 can be used to set or get the entire text content of any type of window.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4,
can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
**wenable/2**

*get or set the enable status of a window*

\[ \text{wenable( \text{Window}, \text{Status} )} \]

+Window \text{<window_handle>}  
?Status \text{<variable> or <integer> in the domain \{0,1\}}

**Comments**

This gets or sets the "enable status" of the given \text{Window}: if \text{Status} is an unbound variable, it returns the existing enable status; otherwise, it is used to enable or disable the \text{Window}. The following table shows the values and meanings of \text{Status}:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>disable the window</td>
</tr>
<tr>
<td>1</td>
<td>enable the window</td>
</tr>
</tbody>
</table>

**Examples**

The following program uses \text{wdcreate/7} and \text{wccreate/8} to create a small dialog containing a single "default push" button:

\[
\text{foo :-}  
\quad \text{Dstyle} = [\text{ws_visible,ws_popup,ws_caption}],  
\quad \text{Bstyle} = [\text{ws_visible,ws_child,bs_defpushbutton}],  
\quad \text{wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle )},  
\quad \text{wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle )}.  
\]

Once compiled, the following call will display the dialog near the top left corner of the screen, but will also disable it so that it cannot be clicked:

\[
?\text{- foo, wenable( foo, 0 ).} \quad \text{<enter>}
\]

yes

In order to make the dialog usable, it must be reenabled; once this is done, it can be hidden by clicking the button:

\[
?\text{- wenable( foo, 1 ).} \quad \text{<enter>}
\]

yes
Notes

When `wenable/2` is used to disable a window, all of its child windows (controls, etc.) are also disabled (see the first example above); however, it is also possible to disable individual child windows independently of the parent dialog. In this way, individual buttons, edit controls, etc., can be "greyed out" when inappropriate, and reenabled when required. Note that disabling the `WIN-PROLOG` "main window" also disables the "console window" together with any program windows, but not any pop-up dialogs. In fact, console call `call_dialog/2` explicitly uses this feature to ensure modal behaviour. In order to avoid the user being permanently locked out if, say, a program crashed during development while the console was disabled, the primary `WIN-PROLOG` windows are automatically shown and enabled whenever an attempt is made to read from the console ("user") input device.

This is one of a family of `WIN-PROLOG` window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the `Windows API` as well as those implemented in third-party DLLs.
`wfclose/1`  

**close a font**  

`wfclose( Font )`  

+Font <atom>  

**Comments**  
This closes the named `Font`, freeing its resources.  

**Examples**  
The following call will create a font called "foo", specifying its face as "Arial", size as 16 points and style as normal ("0"), and assigns this font to the "console window" ("(1,1)"):  

```
?- wfcreate( foo, arial, 16, 0 ), wfont( (1,1), foo ).
yes
```

The next call restores the console's default "Prolog fixed font" ("-1"), before closing the font created above:  

```
?- wfont( (1,1), -1 ), wfclose( foo ).
yes
```

**Notes**  
When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window (see the above example) or graphics device context. Please note that fonts created with `wfcreate/4` are in fact identical to those created by `gfx_font_create/4`: the `wf*/n` window-oriented predicates are completely interchangeable with the newer, `gfx_font*/n"Grafix"` predicates.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the `Windows API` as well as those implemented in third-party DLLs.
`wfcreate/4`

create a font

```prolog
wfcreate( Font, Face, Size, Style )
```

+Font <atom>
+Face <atom>
+Size <integer>
+Style <integer> in the range [0..3]

Comments  This creates the named `Font`, using the given `Face`, point `Size` and `Style`; any existing font of the same name is closed before the new font is created. Styles are listed in this table:

<table>
<thead>
<tr>
<th>Style</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal</td>
</tr>
<tr>
<td>1</td>
<td>italic</td>
</tr>
<tr>
<td>2</td>
<td>bold</td>
</tr>
<tr>
<td>3</td>
<td>bolditalic</td>
</tr>
</tbody>
</table>

Examples  The following call will create a font called "foo", specifying its face as "Arial", size as 16 points and style as normal ("0"), and assigns this font to the "console window" ("(1,1)"):

```
?- wfcreate( foo, arial, 16, 0 ), wfont( (1,1), foo ).
yes
```

The next call restores the console's default "Prolog fixed font" ("-1"), before closing the font created above:

```
?- wfont( (1,1), -1 ), wfclose( foo ).
yes
```

Notes  When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window (see the above example) or graphics device context. Please note that fonts created with `wfcreate/4` are in fact identical to those created by `gfx_font_create/4`: the `wf*/n` window-oriented predicates are completely interchangeable with the newer, `gfx_font*/n` "Grafix" predicates.
This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
wfdata/5

get or check information about a font

\[
\text{wfdata}( \text{Font, Face, Size, Style, Ascent} )
\]

+Font <atom>, <integer> or <tuple>

?Face <atom> or <variable>

?Size <integer> or <variable>

?Style <integer> in the range \([0..3]\) or <variable>

?Ascent <integer> or <variable>

**Comments**

This gets or checks information about the named \text{Font}, picking up its face name, point size, style and ascent (point size above its baseline), and unifying these with values with Face, Size, Style and Ascent respectively. Styles are listed in this table:

<table>
<thead>
<tr>
<th>Style</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal</td>
</tr>
<tr>
<td>1</td>
<td>italic</td>
</tr>
<tr>
<td>2</td>
<td>bold</td>
</tr>
<tr>
<td>3</td>
<td>bolditalic</td>
</tr>
</tbody>
</table>

**Examples**

The following call will create a font called "foo", specifying its face as "Arial", size as 16 points and style as normal ("0"):

\[
?- \text{wfcreate}( \text{foo, arial, 16, 0} ).
\]

yes

The next call gets the font's data, before closing it:

\[
?- \text{wfdata}( \text{foo, F, S, Y, A} ), \text{wfclose}( \text{foo} ).
\]

\begin{verbatim}
F = 'Arial',
S = 16 ,
Y = 0 ,
A = 13
\end{verbatim}

**Notes**

The Windows system does not always provide an exact match for the font specification given when a font is created (see \text{wfcreate/4}):
perhaps the typeface is not present, so an alternative is substituted, or the typeface concerned is not scaleable or does not support a
given style. The \texttt{wfdata/5} predicate provides a means to check whether a font has been created as desired. Its point size and ascent data
are particularly useful in graphics programming, since these values are used to position text vertically. The \texttt{Size} gives the overall height
of the font design, while \texttt{Ascent} gives the height above the "baseline". When displaying a line of text containing more than one font, care
should be made to position all characters along a single baseline: hence the importance of the latter parameter.

As well as handling named fonts, \texttt{wfdata/5} can get or check data about six special pre-defined, numeric "stock" fonts, as well as any integer
font handle where handle is contained in a tuple of the form "font(integer)". The stock fonts are listed here:

<table>
<thead>
<tr>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Prolog fixed font</td>
</tr>
<tr>
<td>0</td>
<td>OEM fixed font</td>
</tr>
<tr>
<td>1</td>
<td>system font</td>
</tr>
<tr>
<td>2</td>
<td>system fixed font</td>
</tr>
<tr>
<td>3</td>
<td>ANSI var font</td>
</tr>
<tr>
<td>4</td>
<td>ANSI fixed font</td>
</tr>
<tr>
<td>5</td>
<td>device default font</td>
</tr>
</tbody>
</table>

When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in
unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window or graphics device
context. Please note that fonts created with \texttt{wcreate/4} are in fact identical to those created by \texttt{gfx_font_create/4}: the \texttt{wf*/n}
window-oriented predicates are completely interchangeable with the newer, \texttt{gfx_font*/n} "Grafix" predicates.

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify
common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely
to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4},
can be used to access virtually any function in the \texttt{Windows API} as well as those implemented in third-party DLLs.
\texttt{wfdict/2}

\textit{return a dictionary of currently defined fonts}

\begin{verbatim}
\texttt{wfdict( Flag, Dict )}
\end{verbatim}

+\texttt{Flag} \quad \textless integer\textgreater \text{ in the domain \{-1,0,1\}}

-\texttt{Dict} \quad \textless variable\textgreater

\textbf{Comments} \quad This returns a list of all the currently defined fonts, as specified in the given \texttt{Flag} (see \texttt{dict/2}), and binds it to the variable \texttt{Dict}.

\textbf{Examples} \quad The following call will create a font called "foo", specifying its face as "Arial", size as 16 points and style as normal ("0"):

\begin{verbatim}
?- \texttt{wfcreate( foo, arial, 16, 0 ).}
\end{verbatim}

\begin{verbatim}
yes
\end{verbatim}

The newly created font will be included along with any other named fonts in the list returned by the next command:

\begin{verbatim}
?- \texttt{wfdict( 0, D ).}
\end{verbatim}

\begin{verbatim}
D = [foo]
\end{verbatim}

\textbf{Notes} \quad When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window (see the above example) or graphics device context. Please note that fonts created with \texttt{wfcreate/4} are in fact identical to those created by \texttt{gfx\_font\_create/4}: the \texttt{wf\_n} window-oriented predicates are completely interchangeable with the newer, \texttt{gfx\_font\_n} "Grafix" predicates.

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
wfind/3

find a desktop window with the given class name and title

\[ \text{wfind}( \text{Class}, \text{Title}, \text{Window} ) \]

+Class <atom>
+Title <string>
-Window <variable>

Comments
This searches the "desktop" for a window with the given Class and Title, binding Window to its handle if successful. If Class is given as an empty atom (''), windows of all classes are included in the search; similarly, if Title is given as the empty string ('`'), windows with any title are included.

Examples
The following call searches for an instance of WIN-PROLOG, whose "main window" has the class "Main", and returns its current title:

\[ \text{?- wfind}( \text{'main'}, \text{``}, \text{W} ), \text{wtext}( \text{W}, \text{T} ) ). \]

\[ W = 0 , \]
\[ T = \text{'WIN-PROLOG - [Console]' \]

Notice that neither the class nor the title is case sensitive:

\[ \text{?- wfind}( \text{'mAIn'}, \text{`wIn-pRoLoG - [cOnSoLe]`, \text{W} ).} \]

\[ W = 0 \]

Notes
The wfind/3 predicate can be used to search the desktop for windows belonging to any application, including main application windows, dialogs, message boxes, and so on. Once any such window is found, wlink/3 can be used to find sibling or child windows: between these two predicates, it is possible to navigate the desktop to locate any window.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wfocu$1

get or set the window focus

\texttt{wfocu$( Window )}\n
?Window \texttt{<window\_handle>} or \texttt{<variable>}

Comments
This gets or sets the window "focus": if \texttt{Window} is an unbound variable, it returns the handle of the window currently in focus; otherwise, focus is set to the given \texttt{Window}.

Examples
The following program uses \texttt{wdcreate/7} and \texttt{wccreate/8} to create a small dialog containing a single "default push" button:

\begin{verbatim}
foo :-
    Dstyle = [ws\_visible,ws\_popup,ws\_caption],
    Bstyle = [ws\_visible,ws\_child,bs\_defpushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ).
\end{verbatim}

Once compiled, the following call will display the dialog near the top left corner of the screen, before immediately restoring focus to the "main window" ("0"):

\begin{verbatim}
?- foo, wfocu$( 0 ).
\end{verbatim}

yes

The next command restores focus to the dialog; once this is done, it can be hidden by clicking the button:

\begin{verbatim}
?- wfocu$( foo ).
\end{verbatim}

yes

Notes
When \texttt{wfocu$1} is used to set focus to a top-level window, focus is usually passed on to one of its controls. Please note that windows can only receive focus when enabled (see \texttt{wenable/2}).

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4},
can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wfont/2**

*get or set the font used by a control window*

\[ wfont( \text{Window}, \text{Font} ) \]

+Window <window_handle>

?Font <variable>, <atom>, <integer> or <tuple>

**Comments**

This gets or sets the Font currently used to display text by the given control Window: if Font is an unbound variable, it returns the existing window font; otherwise, it is used to set the window's new Font.

**Examples**

The following call will create a font called "foo", specifying its face as "Arial", size as 16 points and style as normal ("0"), and assigns this font to the "console window" "(1,1)"

\[ ?- wfccreate( \text{foo, arial, 16, 0 } ), wfont( (1,1), \text{foo} ). \]

yes

The next call restores the console's default "Prolog fixed font" "(-1)", before closing the font created above:

\[ ?- wfont( (1,1), -1 ), wfclose( \text{foo} ). \]

yes

**Notes**

This predicate can only be used to set the font used by "control windows": it has no effect on top-level window title bars. As well as handling named fonts, wfont/2 can be used to get or set six special pre-defined, numeric "stock" fonts, as well as any integer font handle where handle is contained in a tuple of the form "font(integer)". The stock fonts are listed here:

<table>
<thead>
<tr>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Prolog fixed font</td>
</tr>
<tr>
<td>0</td>
<td>OEM fixed font</td>
</tr>
<tr>
<td>1</td>
<td>system font</td>
</tr>
<tr>
<td>2</td>
<td>system fixed font</td>
</tr>
<tr>
<td>3</td>
<td>ANSI var font</td>
</tr>
<tr>
<td>4</td>
<td>ANSI fixed font</td>
</tr>
</tbody>
</table>
When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window (see the above example) or graphics device context. Please note that fonts created with \texttt{wfcreate/4} are in fact identical to those created by \texttt{gfx\_font\_create/4}: the \texttt{wf\_/n} window-oriented predicates are completely interchangeable with the newer, \texttt{gfx\_font\_/n} "Grafix" predicates.

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
**wfsize/4**

get or check the dimensions of a string in a given font

```
wfsize( Font, String, Width, Depth )
```

+Font <atom>, <integer> or <tuple>
+String <string>
?Width <variable> or <integer>
?Depth <variable> or <integer>

**Comments**

This gets or checks the dimensions of a String rendered in the named Font, picking up its point width and depth, and unifying these with Width and Depth respectively.

**Examples**

The following call will create a font called "foo", specifying its face as "Arial", size as 16 points and style as normal ("0"):

```
?- wfcreate( foo, arial, 16, 0 ).
yes
```

The next call gets the dimensions of the string, "hello world" rendered in this font, before closing it:

```
?- wfsize( foo, `hello world`, W, D ), wfclose( foo ).
W = 61 ,
D = 16
```

**Notes**

The `wfsize/4` predicate is handy when computing the size of small control windows, such as "tooltips", where it is desired to fit the window closely around a piece of text. It is also useful for similar purposes in graphics programming, although it should be noted that dimensions are computed relative to the screen, and may not be completely accurate for other, higher-resolution devices such as printers. The related predicate, `wfdata/5`, provides additional information about a font, including its "ascent", which gives the height above the "baseline". When displaying a line of text containing more than one font, care should be made to position all characters along a single baseline: hence the importance of the latter parameter.

As well as handling named fonts, `wfsize/4` can get or check the dimensions of text rendered in any one of seven special pre-defined, numeric "stock" fonts, as well as any integer font handle where handle is contained in a tuple of the form "font(integer)". The stock fonts are listed here:
<table>
<thead>
<tr>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Prolog fixed font</td>
</tr>
<tr>
<td>0</td>
<td>OEM fixed font</td>
</tr>
<tr>
<td>1</td>
<td>system font</td>
</tr>
<tr>
<td>2</td>
<td>system fixed font</td>
</tr>
<tr>
<td>3</td>
<td>ANSI var font</td>
</tr>
<tr>
<td>4</td>
<td>ANSI fixed font</td>
</tr>
<tr>
<td>5</td>
<td>device default font</td>
</tr>
</tbody>
</table>

When creating and using fonts, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a font should never be closed or redefined while currently selected into a window or graphics device context. Please note that fonts created with `wfcreate/4` are in fact identical to those created by `gfx_font_create/4`: the `wf*/n` window-oriented predicates are completely interchangeable with the newer, `gfx_font*/n "Grafix"` predicates.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
call an external Windows API or DLL function

\[ \text{winapi( Function, Arguments, Code, Result )} \]

+Function \; <\text{conjunction}> \text{ or } <\text{integer}>
+Arguments \; <\text{list}>
+Code \; <\text{integer}> \text{ in the range } [0..3]
-Result \; <\text{variable}>

Comments

This calls the named external Windows API or DLL \textit{Function}, passing it an arbitrary number of \textit{Arguments}, encoding any literal text parameters with the given \textit{Code} and binds the \textit{Result} to the function's return value. If the \textit{Function} is a conjunction of two atoms, the first defines the name of the module that exports the function, while the second defines the name of the function itself; if \textit{Function} is an integer, it is assumed to be the known address of a function, which is executed directly. The \textit{Arguments} list can contain any appropriate mixture of the following data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;atom&gt;</td>
<td>the atom must name a memory file previously created with fcreate/5: its address is passed to the API or DLL function (LPSTR)</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>the 32-bit (X86) or 64-bit (X64) integer is passed to the API or DLL function (UINT or INT as appropriate)</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>either of the above cases can be combined with a linear offset as a conjunction of the form, ((&lt;\text{atom}&gt;,&lt;\text{integer}&gt;)) or ((&lt;\text{integer}&gt;,&lt;\text{integer}&gt;)): in both cases, the first element is processed as before, and the second element is added to it; the result is then passed to the API or DLL function (LPSTR, UINT or INT as appropriate)</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>space is allocated in a special reserved buffer, and the text contained in the string is copied into this space, replacing any (&lt;\text{null}&gt;) (character code zero (&quot;0&quot;)) characters with spaces, and terminating the text with a single (&lt;\text{null}&gt;) character; address of the resulting null-terminated string is passed to the API or DLL function (LPSTR)</td>
</tr>
</tbody>
</table>

The \textit{Code} determines which of four character encodings to use to copy literal text arguments to the function, as defined in the following table:
<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8-bit text is passed directly to a Windows ANSI function without conversion</td>
</tr>
<tr>
<td>1</td>
<td>16-bit text is converted to 8-bit &quot;multibyte&quot; text before passing to a Windows ANSI function</td>
</tr>
<tr>
<td>2</td>
<td>16-bit text is passed directly to a Windows WIDE function without conversion</td>
</tr>
<tr>
<td>3</td>
<td>32-bit text is passed directly to a DLL function which can accept this format</td>
</tr>
</tbody>
</table>

Note that while the Code values bear a passing resemblance to those used to specific character encodings in `fcreate/5`, they in fact different. Windows uses an 8-bit ANSI character set, which (in USA/UK versions of the operating system at least) shares most of its code points with ISO/IEC 8859-1; however, ANSI is not the same as the latter standard: in particular, it uses control codes in the range 80h..9fh for certain printable characters. For example, the Trademark symbol ("™") has a Unicode code point of 2122h, but in ANSI is mapped to the value 99h. Please see Appendix L for more information about character sets, encodings and formats.

Examples

The scope of `winapi/4` is so great, that any given set of examples will only touch on its potential. This section presents just a few to illustrate some of the possibilities. The following program shows a simplified implementation of the `msgbox/4` predicate, renamed so that it will compile without errors:

```
msg_box( Title, Message, Style, Key ) :-
    write( Title ) ~> S1,
    write( Message ) ~> S2,
    winapi( (user32,'MessageBoxA'), [0,S2,S1,Style], 1, Key ).
```

Once this program is compiled, the next call will create a message box with a single "OK" button and exclamation mark icon, together with the given title and message; the result is returned when the user clicks on the button or hits <enter>:

```
?- msg_box( ['Tasty'], value('Pi',3.14159), 48, D ).
D = 1 <enter>
```

Notice the use of a Code of "1" in this example: this assumes that the given text is in Windows WIDE format, which more or less maps to Unicode; where necessary, Unicode characters such as the Trademark symbol, 2122h, are converted to their ANSI equivalents (99h in this case) before passing to the ANSI Windows function (MessageBoxA), allowing it to display correctly even on a non-Unic ode version of Windows.

The program below illustrates a method for transferring the entire text of a large "text" window to a disk file, without making use of `WIN-PROLOG` strings. It makes use of two `winapi/4` calls: the first picks up the length of the text in a window, and the second copies the window's text into the buffer ("$") prepared as a memory file by `fcreate/5`:
window_to_file( Window, File ) :-
    wndhdl( (Window,1), Handle ),
    winapi( (user32,'GetWindowTextLengthA'), [Handle], 0, Length ),
    Size is Length + 1,
    fcreate( File, File, -1, 0, 0 ),
    fcreate( $, [], -2, Size, 0 ),
    winapi( (user32,'GetWindowTextA'), [Handle,$,Size], 0, _ ),
    input( Input ),
    output( Output ),
    input( $ ),
    output( File ),
    copy( Len, _ ),
    input( Input ),
    output( Output ),
    fclose( File ),
    fclose( $ ).

Once compiled, it can be used to copy any "text" window, or even the "console window", to a named disk file; the latter case is performed by this command:

?- window_to_file( 1, foo ).
  yes

The reverse process can be carried out by the following program, which only requires a single winapi/4 call to transfer the contents of the prepared buffer to the given window:

file_to_window( File, Window ) :-
    wndhdl( (Window,1), Handle ),
    file( File, 4, Len ),
    Size is Len + 1,
    fcreate( File, File, 0, 0, 0 ),
    fcreate( $, [], -2, Size, 0 ),
    input( Input ),
    output( Output ),
    input( File ),
    output( $ ),
Once this program has also been compiled, the first of the following calls will create an empty text window called "bar", and the second will fill it with the contents of the file, "foo", that was created in the previous example:

?- wtcreate( bar, `Bar`, 0, 0, 0, 0 ), wfocus( 1 ).
yes

?- file_to_window( foo, bar ).
yes

Many Windows API and DLL functions return text in the form of "LPSTR" pointers; winapi/4 returns these as simple integers, but they can be converted to WIN-PROLOG strings by another predicate, wintxt/4. The following program uses the "lstrcat" function to concatenate two strings:

\[
glue( \text{Head}, \text{Tail}, \text{Text} ) :-
\]
\[
\text{winapi( (kernel32,lstrcat), [Head,Tail], 0, Address ),}
\]
\[
\text{wintxt( Address, 0, 0, Text ).}
\]

Once compiled, this program can be called as follows:

?- glue( `hello`, `world`, T ).
T = `helloworld`

As mentioned in the "Comments" section, winapi/4 can be used to call functions exported by virtually any DLL, and not just Windows API functions. Consider the following "C" function, defined in a DLL called "foo.dll":

\[
\text{UINT WINAPI Double(Data)}
\]
\[
\text{UINT Data ;}
\]
\[
\}
\]
The "Double" function simply takes an unsigned integer argument, and returns a value twice its size. The following program shows how it can be called from inside WIN-PROLOG. In this example, three calls are made to winapi/4: first, "LoadLibraryA" loads "foo.dll" into memory, returning a "handle" which can be used to close ("free") the DLL when it is no longer needed; second, a call is made to the "Double" function itself, and finally, the DLL is freed with a call to "FreeLibrary", using the "handle" that was returned in the original call:

```prolog
double( Number ) :-
    winapi( (kernel32,'LoadLibraryA'), [`foo.dll`], 0, Handle ),
    winapi( (foo,'Double'), [Number], 0, Result ),
    winapi( (kernel32,'FreeLibrary'), [Handle], 0, _ ),
    write( Result ), nl.
```

Assuming "foo.dll" has been prepared and the above program has been compiled, the following call will display a number twice the size of the given integer:

```prolog
?- double( 123 ).
```

246

yes

Note that if a DLL function is to be called repeatedly, it is more efficient to load the DLL once, at the start of a session, and then use the functions it contains as often as required, closing it only when finished; here is an alternative to the previous program:

```prolog
:- dynamic dll/1.

load :-
    \+ dll( _ ),
    winapi( (kernel32,'LoadLibraryA'), [`foo.dll`], 0, Handle ),
    assert( dll(Handle) ).

free :-
    retract( dll(Handle) ),
    winapi( (kernel32,'FreeLibrary'), [Handle], 0, _ ).
```
The three `winapi/4` functions have been split amongst three predicates, and once compiled, permit the DLL to be loaded and freed independently of invocations of its "Double" function, as shown in the following sequence of calls:

```prolog
?- load.
yes

?- double( 123 ).
<enter>
246
yes

?- double( 512 ).
<enter>
1024
yes

?- free.
<enter>
yes
```

**Notes**

The `winapi/4` predicate provides a near-universal gateway between WIN-PROLOG and conventional languages such as C, C++, Visual Basic, Delphi and even Java. With only two specific exceptions, absolutely any function in any DLL or Windows API library can be called directly, and its returned results interpreted. The first exception is a small class of Windows API functions that require a "call-back" function address to be passed in for subsequent enumeration, such as "EnumFonts": currently WIN-PROLOG has no way of defining the call-back itself. The second exception is the even-smaller class of Windows API functions that require 32-bit (single precision) floating point numbers as parameters; such as "SetWorldTransform": the floating pointer numbers used in WIN-PROLOG are in 64-bit, double-precision format, which is not easily convertible to the simpler format.

Curiously, the function name in a Windows DLL or API is case sensitive, even though the module name is not. This can be seen in all the examples above, where the module name is routinely given in lowercase, but the function name is quoted with specific capitalisation. Please note that more than a few functions end in an uppercase "A": this is a suffix applied by Windows to any 8-bit ("ANSI") text handling function. When WIN-PROLOG is running in Windows NT, it has the option of using 16-bit ("WIDE") characters, which represent the Windows version of "Unicode"; in the case the names of functions would end in "W". In general, if a Windows API function takes one or more text arguments, then its name, so far as winapi/4 is concerned, must include the "A" suffix if ANSI text is being used, and "W" if WIDE text is
being used. Please see Appendix L for further information about character sets.

A special memory buffer is used by winapi/4 to store parameters prior to calling functions, and this has a default size of 64kb. In exceptional circumstances, for example when trying to pass large strings or collections of strings, this buffer might prove to be too small. There are two options here: either create memory files (see fcreate/5) to store the large data objects, or increase the size of the special buffer by calling the winsze/1 predicate, which lets you retrieve or modify the amount of memory allocated for this purpose.

The winapi/4 predicate itself is relatively robust, maintaining stack levels even if the incorrect number of arguments has been passed to a function; however, please remember that C, C++ and other lower-level languages have little or no runtime type checking, and it is perfectly possible to crash WIN-PROLOG and even Windows itself by making erroneous winapi/4 calls. It is especially important to respect text and other pointer-based parameters, and not to pass in arbitrary numeric values.

Together with wintxt/4, winapi/4 can handle Windows API and DLL functions that require or return text and even complex data structures. The latter can best be handled through input/output functions such as getx/2 and putx/2, either using string I/O via the <~/2 and ~>2 predicates, or direct I/O to and from memory files created with fcreate/5.

As a footnote, and whilst trying to avoid the risk of over-selling winapi/4, it should be stated that nearly all of WIN-PROLOG’s window-handling and graphics predicates, together with the entire development environment and debugger interface, has been developed directly in terms of this powerful predicate, with support, where necessary, from wintxt/4. Other interfaces, such as the "Intelligence Server", "ProWeb", "DDE Interface", "OLE Automation Interface" and the "TCP/IP Library" have also been made possible because of the massive power and flexibility of winapi/4.
**window_handler/2**

get or set the window handler for the given window

```
window_handler( Window, Handler )
```

+Window <atom> or <integer> in the domain {0,1}

+Handler <atom> or <variable>

**Comments**

This gets or sets the name of an arity-4 predicate that will be used as the window Handler for the named top-level (dialog, text or user) Window. If Handler is an unbound variable, it returns the name of the existing handler; otherwise, it sets the given Handler name to be associated with the Window.

**Examples**

The following program uses `wdcreate/7` and `wccreate/8` to create a small dialog containing a single "default push" button:

```
foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Bstyle = [ws_visible,ws_child,bs_defpushbutton],
    wdcreate( foo, 'foo', 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, 'bar', 10, 10, 80, 20, Bstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen (subsequently clicking the button will hide (but not close) the dialog):

```
?- foo.
yes
```

This behaviour is down to the dialog's default window handler, `window_handler/4`, as can be shown by the following call, which picks up the name of "foo"s current handler:

```
?- window_handler( foo, H ).
H = window_handler
```

Now consider the following window handler:

```
foo_handler( (foo,1), msg_button, _, _ ) :-
```
time(_, _, _, H, M, S, _),
write( H-M-S ) => String,
wtext( (foo,1), String ).

foo_handler( Window, Message, Data, Result ) :-
  window_handler( Window, Message, Data, Result ).

The first clause simply intercepts "msg_button" messages destined for the button control, "(foo,1)", and rather than allowing the button to hide the dialog, writes the current time into a string, using this to rename the button. All other messages are picked up by the second clause, which simply passes them on to window_handler/4 for default processing. Once compiled, this program can be attached to the "foo" dialog with the following call:

?- window_handler( foo, foo_handler ).

Now, when the dialog is displayed with a call to foo/0, clicking the button will simply change its label to display the current time, without closing the dialog.

Notes

The window_handler/2 simply maintains a local database of the names of top-level windows and the names of their corresponding "window handler" predicates: when a message arrives for any given window, this database is checked for a match. If the top-level window is found, the corresponding handler is called with four arguments:

<handler>( Window, Message, Data, Result )

The first three arguments are bound as follows: Window contains the handle of the window that received the message (<window_handle>), Message contains the name of the message (<atom> if known, otherwise <integer>), and Data contains supplementary information (message dependent). See Appendix D for further information about messages.

The final argument, Result, is an unbound variable. If the window handler binds this to any value, it causes the top-level window (usually a dialog) to terminate (be hidden): if the window was invoked by call_dialog/2, this value is passed back as the latter predicate's second argument. Consider the following variant on the window handler shown above:

foo_handler( (foo,1), msg_button, _, (H,M,S) ) :-
  time( _, _, _, H, M, S, _ ).

foo_handler( Window, Message, Data, Result ) :-

window_handler( Window, Message, Data, Result ).

This time the first clause simply returns the time. Once compiled, the dialog can be called "modally" as follows:

?- foo, call_dialog( foo, R ).
R = (10, 8, 42)

To remove the association between a window and its user-defined handler, window_handler/2 is called to "re-associate" the window with the default handler, window_handler/4:

?- window_handler( foo, window_handler ).

Now when the dialog is called modally, it simply returns the name of the button when clicked:

?- foo, call_dialog( foo, R ).
R = bar

Please note that window_handler/2 only associates "top-level" windows ("Dialog", "Text" and "User" windows, together with the "main window" ("0") and "console window" ("1"): it is not possible to associate window handlers with individual controls within these windows. When a window handler is called, its first argument specifies the precise window (top level or control as appropriate) that has received the message, and it is up to the programmer to decide how to use this information.

In the example above, the first clause specifies the exact window ("(foo,1)") and message ("msg_button") that is being processed, but either or both of these parameters can be specified as fully or partially uninstantiated. For example, the following clause would pick up "msg_button" messages for any button within any dialog for which foo_handler/4 was set as the handler, returning its integer ID rather than its name:

foo_handler( (Dialog, ID), msg_button, _, ID ).

Any one handler predicate can serve an arbitrary number of top-level windows (indeed, window_handler/4 serves all windows until replaced by a user-defined handler). Furthermore, it is possible to associate window and handler names by calling window_handler/2 even before the windows and/or handler predicates exist. All that is required is that the named, arity-4 predicate is compiled and ready by the time a window is created for which that predicate is the selected handler.
**window_handler/4**

*default window handler*

\[
\text{window_handler}( \text{Window}, \text{Message}, \text{Data}, \text{Result} )
\]

+Window & <window_handle> \\
+Message & <atom> \\
+Data & <atom>, <integer> or <conjunction> \\
?Result & <variable> or <atom>

**Comments**
This provides standard, default processing of window messages, specified by the given Window handle, Message name and Data item. In certain cases, it attempts to unify Result with an atom that is used to indicate completion of a dialog.

**Examples**
The following program uses \text{wdcreate/7} and \text{wccreate/8} to create a small dialog containing a single "default push" button:

\[
\text{foo} :-
\begin{align*}
\text{Dstyle} &= \{\text{ws_visible, ws_popup, ws_caption}\}, \\
\text{Bstyle} &= \{\text{ws_visible, ws_child, bs_defpushbutton}\}, \\
\text{wdcreate}( \text{foo}, \text{'foo'}, 10, 10, 100, 60, \text{Dstyle} ), \\
\text{wccreate}( (\text{foo},1), \text{button, 'bar'}, 10, 10, 80, 20, \text{Bstyle} ).
\end{align*}
\]

Once compiled, the following call will display the dialog "modally" near the top left corner of the screen (subsequently clicking the button will hide (but not close) the dialog):

\[
\begin{align*}
?\cdot \text{foo}, \text{call_dialog}( \text{foo}, \text{R} ). \\
\text{R} &= \text{bar}
\end{align*}
\]

This behaviour is down to the dialog's default window handler, \text{window_handler/4}, as can be shown by calling this predicate directly with a "msg_button" message:

\[
\begin{align*}
?\cdot \text{window_handler}( (\text{foo},1), \text{msg_button, _, R} ). \\
\text{R} &= \text{bar}
\end{align*}
\]

**Notes**
The \text{window_handler/4} predicate provides default processing of window messages, and is attached to all top-level windows unless replaced in a call to \text{window_handler/2}. The present predicate ignores most messages apart from "msg_button" and "msg_close": here, it
tries to unify \texttt{Result} to a lower case atom containing (in the former case) the button's name or (in the latter case) the word, "close". See \texttt{window_handler/2} for a more detailed discussion.
winsze/1

return or adjust the Windows API parameter memory allocation

\texttt{winsze( \textit{Size} )}

+\textit{Size} <integer> or <variable>

Comments
This is used to check or set the amount of memory allocated to the \texttt{winapi/4} predicate for storing parameters before calling external Windows API or DLL functions. If \textit{Size} is a variable, the current memory allocation is returned, in kilobytes; if \textit{Size} is an integer, this is interpreted as the number of kilobytes to allocated. A \textit{Size} of zero resets the default allocation of 64kb.

Examples
The following call returns the current size of the Windows API parameter buffer, in kilobytes:

\begin{verbatim}
?- winsze( Size ).
Size = 64
\end{verbatim}

Suppose a forthcoming call to \texttt{winapi/4} is going to include a Unicode string containing 96,000 characters: this will require at least 192,000 bytes of storage, so (with spare space for other overheads), the call:

\begin{verbatim}
?- winsze( 256 ).
yes
\end{verbatim}

can be used to ensure that this string will fit.

Notes
A special memory buffer is used by \texttt{winapi/4} to store parameters prior to calling functions, and this has a default size of 64kb. In exceptional circumstances, for example when trying to pass large strings or collections of strings, this buffer might prove to be too small. There are two options here: either create memory files (see \texttt{fcreate/5}) to store the large data objects, or increase the size of the special buffer by calling the \texttt{winsze/1} predicate, which lets you retrieve or modify the amount of memory allocated for this purpose.

Although the \texttt{winsze/1} predicate does not prevent you from doing so, it is not a good idea to set very small sizes for the Windows API parameter buffer, since this may interfere with the proper functioning of the \texttt{WIN-PROLOG} development environment.
wintxt/4

get or set data in a memory file or numerical address

\[
\text{wintxt( Address, Length, Code, String )}
\]

+Address \<atom>, \<conjunction> or \<integer>  
+Length \<integer>  
+Code \<integer> in the range \[0..3\]  
?String \<variable> or \<string>

Comments

This reads or writes data at a given memory Address according to the specified Length, Code and String. If String is an unbound variable, the current data at the Address is copied into it; if String is bound to a string, then this data is copied to the specified Address. If Length is greater than zero, it specifies the number of bytes to copy, truncating or null-padding an existing string as necessary; if Length is equal to zero, the String is treated as "null-terminated" data. The Address can consist of any of the following data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;atom&gt;</td>
<td>the atom must name a memory file previously created with fcreate/5</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>the 32-bit integer is treated as an address</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>either of the above cases can be combined with a linear offset as a conjunction of the form, (&lt;atom&gt;,&lt;integer&gt;) or (&lt;integer&gt;,&lt;integer&gt;): in both cases, the first element is processed as before, and the second element is added to it</td>
</tr>
</tbody>
</table>

The Code determines which of four character encodings to use to copy text between WIN-PROLOG and memory, as defined in the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8-bit text is passed directly</td>
</tr>
<tr>
<td>1</td>
<td>16-bit text is converted to/from 8-bit &quot;multibyte&quot;</td>
</tr>
<tr>
<td>2</td>
<td>16-bit text is passed directly</td>
</tr>
<tr>
<td>3</td>
<td>32-bit text is passed directly</td>
</tr>
</tbody>
</table>

Please see the entry winapi/4 and Appendix L for more information about character sets, encodings and formats.
Examples

Many Windows API and DLL functions return text in the form of "LPSTR" pointers; winapi/4 returns these as simple integers, but they can be converted to WIN-PROLOG strings by another predicate, wintxt/4. The following program uses the "lstrcat" function to concatenate two strings:

```prolog
glue( Head, Tail, Text ) :-
    winapi( (kernel32,lstrcat), [Head,Tail], 0, Address ),
    wintxt( Address, 0, 0, Text ).
```

Once compiled, this program can be called as follows:

```prolog
?- glue( `hello`, `world`, T ).
T = `helloworld`
```

Notes

The wintxt/4 allows data to be read or written in two formats: as fixed-length, "binary data", all 256 possible byte values can be transferred between memory and WIN-PROLOG strings. When reading data from memory (String is a variable), the exact number of bytes specified in Length is copied into the string. When writing data (String is already a string), the number of bytes specified in Length is copied out to memory: if the String is shorter than Length, an appropriate number of <null> characters are copied to fill the memory area; if the String is longer than Length, it is truncated to that length.

Where Length is specified as zero ("0"), data is treated as a "null-terminated string". When reading data from memory (String is a variable), all characters up to, but not including, the first <null> character (character code 00h) are copied into the string. When writing data (String is already a string), any <null> characters in the string are replaced by <space> characters (character code 20h) while copying out to memory, and a single <null> is appended, so one more byte than the string's length is copied.

The processing of null-terminated strings greatly facilitates the interfacing of WIN-PROLOG to C, C++ and other languages. With the support of wintxt/4, winapi/4 can handle Windows API and DLL functions that require or return text and even complex data structures. The latter can best be handled through input/output functions such as getx/2 and putx/2, either using string I/O via the =~/2 and ~>/2 predicates, or direct I/O to and from memory files created with fcreate/5. See the entry for winapi/4 for more details.
follow the link between one window and another

\texttt{wlink(Source, Flag, Target)}

\begin{itemize}
  \item [+Source] \texttt{<window\_handle>}
  \item [+Flag] \texttt{<integer>} in the range \([-1..5]\)
  \item [-Target] \texttt{<variable>}
\end{itemize}

Comments

This follows the link specified in Flag from a given Source window to another window, and binds the latter's window handle to Target. The following table lists the values and meanings of Flag:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>link to parent of window</td>
</tr>
<tr>
<td>0</td>
<td>link to first sibling of window</td>
</tr>
<tr>
<td>1</td>
<td>link to last sibling of window</td>
</tr>
<tr>
<td>2</td>
<td>link to next sibling of window</td>
</tr>
<tr>
<td>3</td>
<td>link to previous sibling of window</td>
</tr>
<tr>
<td>4</td>
<td>link to owner of window</td>
</tr>
<tr>
<td>5</td>
<td>link to first child of window</td>
</tr>
</tbody>
</table>

Examples

The following call links from WIN-PROLOG's "main window" ("0") to its first child window (the MDI Client window, ",(0,1)"):  

\texttt{?\textasciitilde wlink( 1, -1, W ).}  
\texttt{W = (0,1)}

Notes

The \texttt{wfind/3} predicate can be used to search the desktop for windows belonging to any application, including main application windows, dialogs, message boxes, and so on. Once any such window is found, \texttt{wlink/3} can be used to find sibling or child windows: between these two predicates, it is possible to navigate the desktop to locate any window.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4},
can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
add an item to a listbox control

\[ \text{wlstadd( Window, Position, String, Item )} \]

+Window  \hspace{1cm} \text{<window_handle>}
+Position \hspace{1cm} \text{<integer>}
+String \hspace{1cm} \text{<string>}
+Item \hspace{1cm} \text{<integer>}

**Comments**

This adds a \textit{String}, together with a user-defined numerical \textit{Item}, to a "listbox" control \textit{Window} at the specified \textit{Position}. The first entry in the control is numbered zero ("0"); if \textit{Position} is set to "-1", the \textit{String} is inserted at its default position, which depends upon the listbox style (see \textit{Appendix C}).

**Examples**

Consider a dialog called "foo" that includes a listbox control with an ID of 9, the "lbs_sort" style, and the following items:

\begin{verbatim}
  aardvark 314
  badger   273
  dingo    981
\end{verbatim}

The following call will insert the string, "cheetah", with a user-defined item, "123", at its default (sorted) location between "badger" and "dingo", maintaining alphabetical order:

\begin{verbatim}
?- \text{wlstadd( (foo,9 ), -1, `cheetah`, 123 ).}
  yes
\end{verbatim}

The next call inserts the string, "zebra", together with the item, "456", but forces it to become the second entry, immediately after "aardvark", even though this spoils the alphabetical order:

\begin{verbatim}
?- \text{wlstadd( (foo,9 ), 1, `zebra`, 456 ).}
  yes
\end{verbatim}

**Notes**

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful \textit{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \textit{winapi/4} and its helper predicate,
wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wlstdel/2

delete an item from a listbox control

\texttt{wlstdel( \textit{Window}, \textit{Position} )}

+\textit{Window} \hspace{1cm} \texttt{<window\_handle>}
+\textit{Position} \hspace{1cm} \texttt{<integer>}

Comments
This deletes an entry in a "listbox" control \textit{Window} from the specified \textit{Position}. The first entry in this type of control is numbered zero ("0").

Examples
Consider a dialog called "foo" that includes a listbox control with an ID of 9 and containing the following items:

\begin{verbatim}
aardvark 314
badger   273
dingo    981
\end{verbatim}

The following call will delete the second entry ("badger"):

\begin{verbatim}
?- \texttt{wlstdel( \textit{(foo,9)}, 1 ).}  \hspace{1cm} \texttt{<enter>}
yes
\end{verbatim}

Notes
This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
wlstfnd/4

find an item in a listbox control

\[ \text{wlstfnd( Window, Position, String, Match )} \]

+Window \quad <\text{window}\_\text{handle}>
+Position \quad <\text{integer}>
+String \quad <\text{string}>
-Match \quad <\text{variable}?>

Comments
This searches for a \text{String} in a "listbox" control \text{Window}, starting one place after the specified \text{Position}. The first entry in this type of control is numbered zero ("0"), so in order to search from the top of the list, \text{Position} should be set to ",-1". The position of the first string whose first characters match the \text{String} is bound to \text{Match}.

Examples
Consider a dialog called "foo" that includes a listbox control with an ID of 9 and containing the following items:

\begin{verbatim}
aardvark 314
badger   273
dingo    981
\end{verbatim}

The following call will search from the top of the list for an entry that begins with the letters, "BaD". Note that the search is not case-sensitive, and so will match "badger", at position ("1"):

\[ ?- \text{wlstfnd( (foo,9 ), -1, `BaD`, M ).} \quad <\text{enter}> \]
M = 1

Notes
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned \text{Match} parameter may be passed directly into \text{wlstget/4}, \text{wlstsel/2} or \text{wlstsel/3} in order to obtain or select the matched string.

This is one of a family of \text{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \text{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \text{winapi/4} and its helper predicate, \text{wintxt/4}, can be used to access virtually any function in the \text{Windows API} as well as those implemented in third-party DLLs.
**wlstget/4**

*get an item from a listbox control*

```
wlstget( Window, Position, String, Item )
```

+Window            <window_handle>
+Position          <integer>
-String            <variable>
-Item              <variable>

**Comments**
This gets an item from a "listbox" control *Window* at the specified *Position*, and binds its text to *String* and user-defined item value to *Item*. The first entry in this type of control is numbered zero ("0").

**Examples**
Consider a dialog called "foo" that includes a listbox control with an ID of 9 and containing the following items:

```
aardvark 314
badger   273
dingo    981
```

The following call will pick up the second entry ("badger", with an its item value):

```
?- wlstget( (foo,9 ), 1, S, I ).  <enter>
S = `badger` ,
I = 273
```

**Notes**
This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful *winapi/4* to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, *winapi/4* and its helper predicate, *wintxt/4*, can be used to access virtually any function in the **Windows API** as well as those implemented in third-party DLLs.
wlstsel/2

get or set the selection in a single-choice listbox control

\[ \text{wlstsel( Window, Position )} \]

+Window <window_handle>
+Position <integer>

Comments
This gets or sets the selection \( \text{Position} \) in a single-choice "listbox" control \( \text{Window} \). The first entry in this type of control is numbered zero ("0"). If \( \text{Position} \) is an unbound variable, it returns the currently selected entry; if \( \text{Position} \) is an integer, it selects the indicated entry accordingly. A \( \text{Position} \) of "-1" can be used to clear the selection, and if this value is returned, it indicates that no entry is currently selected.

Examples
Consider a dialog called "foo" that includes a listbox control with an ID of 9 and containing the following items:

\[
\begin{align*}
aardvark & \quad 314 \\
bogey & \quad 273 \\
dingo & \quad 981 \\
dinga & \quad 582 \\
dinga & \quad 582 \\
\end{align*}
\]

The following call will select the second entry ("badger"):

\[
\text{?- wlstsel( (foo,9 ), 1 ).} \quad <\text{enter}> \\
\text{yes}
\]

Suppose the user has clicked on the "dingo" entry; the following call will return this selection:

\[
\text{?- wlstsel( (foo,9 ), P ).} \quad <\text{enter}> \\
P = 2
\]

Notes
This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful \( \text{winapi/4} \) to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \( \text{winapi/4} \) and its helper predicate, \( \text{wintxt/4} \), can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wlstsel/3**

*get or set a selection in a listbox control*

\[
\text{wlstsel( Window, Position, State )}
\]

+Window <window_handle>
+Position <integer>
?State <variable> or <integer> in the domain \( \{0,1\} \)

**Comments**

This gets or sets the selection State of a "listbox" control Window at the specified Position. The first entry in this type of control is numbered zero ("0"). If State is an unbound variable, it returns the current state at the specified Position; if State is an integer, it sets the selection state accordingly. Where a multiple-choice listbox is involved, a Position of "-1" can be used to set all entries in a single call. States and their meanings are described in the following table:

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>item is not selected</td>
</tr>
<tr>
<td>1</td>
<td>item is selected</td>
</tr>
</tbody>
</table>

**Examples**

Consider a dialog called "foo" that includes a listbox control with an ID of 9 and containing the following items:

- aardvark 314
- badger 273
- dingo 981

The following call will select the second entry ("badger"):  

\[- \text{wlstsel( (foo,9 ), 1, 1 ).} \]

**Notes**

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wmclose/1

close a menu

    wmclose( Menu )

    +Menu <atom>

Comments
This closes the named Menu, freeing its resources.

Examples
The following call will create an empty menu called "foo":

    |- wmccreate( foo ).
    yes

The next call adds an item to "foo", with the label "Bar", where "B" is the hotkey, and menu item number "1000":

    |- wmnuadd( foo, 0, `&Bar`, 1000 ).
    yes

This menu can now be added to the second position of the main menu bar, with the label, "Sux", where "x" is the hotkey:

    |- wmnuadd( 0, 1, `Su&x`, foo ).
    yes

This removes the new menu entry, before closing the menu:

    |- wmnudel( 0, 1 ), wmclose( foo ).
    yes

Notes
When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely
to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
**wmcreate/1**

create a menu

```
wmcreate( Menu )
```

+Menu <atom>

**Comments**

This creates the named `Menu`; any existing menu of the same name is closed before the new menu is created.

**Examples**

The following call will create an empty menu called "foo":

```
?- wmcreate( foo ).
yes
```

The next call adds an item to "foo", with the label "Bar", where "B" is the hotkey, and menu item number "1000":

```
?- wmnuaadd( foo, 0, `&Bar`, 1000 ).
yes
```

This menu can now be added to the second position of the main menu bar, with the label, "Sux", where "x" is the hotkey:

```
?- wmnuaadd( 0, 1, `Su&x`, foo ).
yes
```

This removes the new menu entry, before closing the menu:

```
?- wmnudel( 0, 1 ), wmclose( foo ).
yes
```

**Notes**

Once created with `wmcreate/1`, menus are built up through successive calls to `wmnuaadd/4`; the latter predicate can add menu items, separators, and even submenus to an existing menu, as well as place completed menus on the main menu bar. Menus can be closed with `wmclose/1`, but should first have been unliked from any menus with `wmnudel/2`. See the entry for `wmnuaadd/4` to find out more about menu programming.

When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result
in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
wmdict/2

return a dictionary of currently defined menus

wmdict( Flag, Dict )

+Flag <integer> in the domain {-1,0,1}
-Dict <variable>

Comments This returns a list of all the currently defined menus, as specified in the given Flag (see dict/2), and binds it to the variable Dict.

Examples The following call will create an empty menu called "foo":

?- wmcupdate( foo ).
yes

The newly created menu will be included along with any other named menus in the list returned by the next command:

?- wmdict( 0, D ).
D = [search, options, file, help, edit, foo, run]

The next call closes the new menu:

?- wmcupdate( foo ).
yes

Notes When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
add an item to a menu

wmnuadd( Menu, Position, String, Item )

+Menu <atom>, <integer> or <tuple>
+Position <integer>
+String <string>
+Item <integer> or <atom>

Comments
This adds an Item to a Menu, using the given String label at the specified Position. The first entry in a menu is numbered zero ("0"); if Position is set to "-1", the Item is appended to the menu. The Item may be any of the following:

<table>
<thead>
<tr>
<th>Item</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero (0)</td>
<td>the menu item is a separator</td>
</tr>
<tr>
<td>other &lt;integer&gt;</td>
<td>the 16-bit integer specifies the data field of a msg_menu or msg_sysmenu message</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>the atom names a linked submenu</td>
</tr>
</tbody>
</table>

The Menu will normally be an atom naming a menu created with wmcreate/1, but may also be one of the following two special integers:

<table>
<thead>
<tr>
<th>Menu</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the WIN-PROLOG main menu bar</td>
</tr>
<tr>
<td>1</td>
<td>the WIN-PROLOG system menu</td>
</tr>
</tbody>
</table>

Finally, the Menu may be a tuple of the form, "menu(<integer>)", where the integer is the raw handle of a menu obtained by some other means (for example, by a call to winapi/4).

Examples
The following call will create an empty menu called "foo":

```
?- wmcreate( foo ).
yes
```

The next call adds an item to "foo", with the label "Bar", where "B" is the hotkey, and menu item number "1000":

```
```
?- wmnude( foo, 0, `&Bar`, 1000 ).
yes

This menu can now be added to the second position of the main menu bar, with the label, "Sux", where "x" is the hotkey:

?- wmnude( 0, 1, `Su&x`, foo ).
yes

This removes the new menu entry, before closing the menu:

?- wmnudel( 0, 1 ), wmclose( foo ).
yes

Notes
There are three types of menu item: numeric, separator, and submenu. The first of these is the most common, and is used to launch a command specified by a 16-bit integer. Numbers in the range 0..999 are used internally by the WIN-PROLOG development environment, while those in the range 61440..65535 are used by Windows itself. It is therefore recommended that user applications choose numbers in the range 1000..61439.

When a numeric menu item is selected by the user, it results in a "msg_menu" message being sent to whichever MDI child ("console", "text" or "user") window is currently in focus. The message's "data" field contains the 16-bit menu item. For example, consider the following window handler:

    menu_handler( Window, msg_menu, Data, _ ) :-
        msgbox( menu(Window), Data, 48, _ ).

    menu_handler( Window, Message, Data, Result ) :-
        window_handler( Window, Message, Data, Result ).

This contains a single clause which picks up all "msg_menu" messages with data values in the range 1000..61439, and displays a message box; all other messages are passed to window_handler/4. This handler can be attached to the "console window" with the following call:

?- window_handler( 1, menu_handler ).
yes

The next command adds a numeric data item directly to the WIN-PROLOG main menu bar, between "File" and "Edit" if the system is not currently maximised:
Now, whenever the new menu item is clicked or the user types \texttt{<alt-T>} while the console is in focus, a message box is displayed with the title "menu(<window\_handle>)", together with a message consisting of the menu number. By assigning a different number to each menu command, a handler such as this can be built up to process any number of menu items easily, either system-wide or window by window:

\begin{verbatim}
menu\_handler( Window, msg\_menu, 1000, _ ) :-
    ... perform function 1000 on "Window" ...

menu\_handler( Window, msg\_menu, 1001, _ ) :-
    ... perform function 1001 on "Window" ...

... etc
\end{verbatim}

Numeric items added to the \textbf{WIN-PROLOG} system menu ("1") generate "msg\_sysmenu" messages instead of "msg\_menu": however, these are programmed in much the same manner. See \texttt{window\_handler/2} for more information about message handling in general.

The remaining two menu item types are simpler: where an integer of zero ("0") is given as the \texttt{Item}, a separator is introduced into the menu. If the \texttt{Item} is bound to an atom, this must be the name of an existing menu, previously created with \texttt{wmcreate/1}: this defines a submenu. When submenus are being added, it is essential to avoid recursive loops: a given menu should never occur directly or indirectly as or on one of its own submenus; Windows has a tendency to crash rather nastily if this is done.

This is one of a family of \textbf{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wmnudel/2

delete an item from a menu

\[ \text{wmnudel( Menu, Position )} \]

+Menu \( \text{<atom>, <integer> or <tuple>} \)
+Position \( \text{<integer>} \)

Comments
This deletes an item in a Menu from the specified Position. The first entry in a menu is numbered zero ('0').

Examples
The following call will add a numeric menu item directly to the second position of the main menu bar ('0'), with the label, "Su&x", where "x" is the hotkey:

\[ ?- \text{wmnuadd( 0, 1, `Su&x`, 1000 ).} \]

yes

This entry can be removed by the next call:

\[ ?- \text{wmnudel( 0, 1 ).} \]

yes

Notes
When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful \text{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \text{winapi/4} and its helper predicate, \text{wintxt/4}, can be used to access virtually any function in the \text{Windows API} as well as those implemented in third-party DLLs.
wmnuget/4

get an item from a menu

\[
\text{wmnuget}( \text{Menu, Position, String, Item} )
\]

+Menu <atom>, <integer> or <tuple>
+Position <integer>
-String <variable>
-Item <variable>

Comments
This gets a label and item from a Menu at the specified Position, and binds them to String and Item respectively. The first entry in a menu is numbered zero ("0") (see wmnuadd/4 for a description of Item types).

Examples
The following call picks up the first (number "0") entry from the WIN-PROLOG main menu bar (the "File" menu if the system is not currently maximised), and then picks up this menu's own first entry ("New"):

```prolog
?- wmnuget( 0, 0, S, I ), wmnuget( I, 0, T, N ).
S = `&File`,
I = file,
T = `&New`,
N = 101
```

Notes
When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wmnunbl/3

get or set an item's enable status in a menu

wmnunbl(Menu, Position, Status)

+Menu <atom>, <integer> or <tuple>
+Position <integer>
?Status <variable> or <integer> in the range [0..2]

Comments
This gets or sets the "enable status" of an item in a Menu at the specified Position; the first entry in a menu is numbered zero ("0"). If Status is an unbound variable, it returns the existing enable status; otherwise, it is used to enable or disable the menu item. The following table shows the values and meanings of Status:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the item is disabled and greyed out</td>
</tr>
<tr>
<td>1</td>
<td>the item is enabled and not greyed out</td>
</tr>
<tr>
<td>2</td>
<td>the item is disabled but not greyed out</td>
</tr>
</tbody>
</table>

Examples
The following call disables and greys out the sixth (number "5") entry from the WIN-PROLOG main menu bar (the "Window" menu if the system is not currently maximised):

?- wmnunbl(0, 5, 0).
yes

Notes
When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wmnusel/3

get or set an item's selection status in a menu

wmnusel( Menu, Position, Status )

+Menu <atom>, <integer> or <tuple>
+Position <integer>
?Status <variable> or <integer> in the range [0..2]

Comments

This gets or sets the "selection status" of an item in a Menu at the specified Position; the first entry in a menu is numbered zero ("0"). If Status is an unbound variable, it returns the existing enable status; otherwise, it is used to enable or disable the menu item. The following table shows the values and meanings of Status:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the item is not ticked</td>
</tr>
<tr>
<td>1</td>
<td>the item is ticked</td>
</tr>
</tbody>
</table>

Examples

The following call picks up the sixth (number "5") entry from the WIN-PROLOG main menu bar (the "Window" menu if the system is not currently maximised), and ticks this menu's own first entry ("Cascade"):  

?- wmnuset( 0, 5, S, I ), wmnusel( I, 0, 1 ).
S = `&Window`,
I = menu(2192)<enter>

Notes

When creating and using menus, it is important to remember that they are low-level Windows resources, and that their misuse can result in unpredictable behaviour. In particular, a menu should never be closed while it is attached to another menu, and menus should never be linked recursively.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wndgbdy/2**

*get or set the buddy window of a nudgebar control*

```prolog
wndgbdy( Window, Buddy )
```

+Window <window_handle>

?Buddy <window_handle> or <variable>

**Comments**

This gets or sets the Buddy window of a "nudgebar" (up-down) control Window. If Buddy is an unbound variable, it is bound to the existing buddy window; otherwise it is used to set the buddy window.

**Examples**

The following program uses `wdcreate/7` and `wccreate/8` to create a small dialog containing a single vertical nudgebar and an associated "buddy" edit window:

```prolog
foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Estyle = [ws_visible,ws_child,ws_ex_clientedge,es_number],
    Nstyle = [ws_visible,ws_child,uds_setbuddyint,uds_alignright],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,2), edit, ``, 10, 10, 80, 20, Estyle ),
    wccreate( (foo,1), nudgebar, ``, 70, 10, 20, 20, Nstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen:

```prolog
?- foo.
```

yes

Initially, the nudgebar created in this example will display on top of the edit window; the following call causes the nudgebar to become embedded within the edit window, which becomes linked to the former as its "buddy":

```prolog
?- wndgbdy( (foo,1), (foo,2) ).
```

yes

When first created, a nudgebar has a default range of [100, 0]; this can be shown in the next call:
?- wndgrng( (foo,1), L, U ).
L = 100 ,
U = 0

The next two calls will change the nudgebar range to [0..1000], and set the position to "250", which value is automatically echoed to the buddy edit window:

?- wndgrng( (foo,1), 0, 1000 ).
yes

?- wndgpos( (foo,1), 250 ).
yes

Finally, after the nudgebar has been adjusted by the user, or its buddy window edited directly, its position can be queried:

?- wndgpos( (foo,1), T ).
T = 810

Notes

Three predicates, wndgbdy/2, wndgrng/3 and wndgpos/2, provide support for "nudgebar" controls, which are based on the Windows "common control" class, "mscts_updown32", commonly known as "Up-Down Controls": see wcreate/8 for more information.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wndgpos/2

get or set the position of a nudgebar control

\[
\text{wndgpos( Window, Position )}
\]

+Window \text{<window_handle>}

?Position \text{<integer> or <variable>}

Comments  
This gets or sets the \text{Position} of a "nudgebar" (up-down) control \text{Window}. If \text{Position} is an unbound variable, it is bound to the existing position; otherwise it is used to set the position.

Examples  
The following program uses \text{wdcreate/7} and \text{wccreate/8} to create a small dialog containing a single vertical nudgebar and an associated "buddy" edit window:

\[
\text{foo} :-
\]

\[
\text{Dstyle = [ws_visible,ws_popup,ws_caption],}
\]

\[
\text{Estyle = [ws_visible,ws_child,ws_ex_clientedge,es_number],}
\]

\[
\text{Nstyle = [ws_visible,ws_child,uds_setbuddyint,uds_alignright],}
\]

\[
\text{wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),}
\]

\[
\text{wccreate( (foo,2), edit, ``, 10, 10, 80, 20, Estyle ),}
\]

\[
\text{wccreate( (foo,1), nudgebar, ``, 70, 10, 20, 20, Nstyle ).}
\]

Once compiled, the following call will display the dialog near the top left corner of the screen:

\[
?- \text{foo}.
\]

yes

Initially, the nudgebar created in this example will display on top of the edit window; the following call causes the nudgebar to become embedded within the edit window, which becomes linked to the former as its "buddy":

\[
?- \text{wndgbdy( (foo,1), (foo,2) ).}
\]

yes

When first created, a nudgebar has a default range of [100..0]; this can be shown in the next call:
?- wndgrng( (foo,1), L, U ).  
   L = 100 ,  
   U = 0

The next two calls will change the nudgebar range to [0..1000], and set the position to "250", which value is automatically echoed to the buddy edit window:

?- wndgrng( (foo,1), 0, 1000 ).  
   yes

?- wndgpos( (foo,1), 250 ).  
   yes

Finally, after the nudgebar has been adjusted by the user, or its buddy window edited directly, its position can be queried:

?- wndgpos( (foo,1), T ).  
   T = 810

**Notes**

Three predicates, `wndgbdy/2`, `wndgrng/3` and `wndgpos/2`, provide support for "nudgebar" controls, which are based on the Windows "common control" class, "msctls_updown32", commonly known as "Up-Down Controls": see `wcreate/8` for more information.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the **Windows API** as well as those implemented in third-party DLLs.
**wndgrng/**

*get or set the range of a nudgebar control*


```prolog
wndgrng( Window, Lower, Upper )
```

+Window <window_handle>

?Lower <integer> or <variable>

?Upper <integer> or <variable>

**Comments**

This gets or sets the *Lower* and *Upper* limits of the range of a "nudgebar" (up-down) control *Window*. If either or both of *Lower* and *Upper* are unbound variables, they are bound to the existing lower and/or upper limits accordingly; otherwise they are used to set these limits.

**Examples**

The following program uses *wdcreate/7* and *wccreate/8* to create a small dialog containing a single vertical nudgebar and an associated "buddy" edit window:

```
foo :-
   Dstyle = [ws_visible,ws_popup,ws_caption],
   Estyle = [ws_visible,ws_child,ws_ex_clientedge,es_number],
   Nstyle = [ws_visible,ws_child,uds_setbuddyint,uds_alignright],
   wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
   wccreate( (foo,2), edit, ``, 10, 10, 80, 20, Estyle ),
   wccreate( (foo,1), nudgebar, ``, 70, 10, 20, 20, Nstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen:

```
?- foo.
yes
```

Initially, the nudgebar created in this example will display on top of the edit window; the following call causes the nudgebar to become embedded within the edit window, which becomes linked to the former as its "buddy":

```
?- wndgbdy( (foo,1), (foo,2) ).
yes
```

When first created, a nudgebar has a default range of [100..0]; this can be shown in the next call:
?- wndgrng( (foo,1), L, U ).  
L = 100 ,  
U = 0

The next two calls will change the nudgebar range to [0..1000], and set the position to "250", which value is automatically echoed to the buddy edit window:

?- wndgrng( (foo,1), 0, 1000 ).<enter>
yes

?- wndgpos( (foo,1), 250 ).<enter>
yes

Finally, after the nudgebar has been adjusted by the user, or its buddy window edited directly, its position can be queried:

?- wndgpos( (foo,1), T ).<enter>
T = 810

**Notes**

Three predicates, \texttt{wndgbdy/2}, \texttt{wndgrng/3} and \texttt{wndgpos/2}, provide support for "nudgebar" controls, which are based on the Windows "common control" class, "msctls\_updown32", commonly known as "Up-Down Controls": see \texttt{wcreate/8} for more information.

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \texttt{Windows API} as well as those implemented in third-party DLLs.
**wndhdl/2**

Convert between logical and raw window handle

```
wndhdl( Window, Raw )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt; or &lt;variable&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Raw</td>
<td>&lt;integer&gt; or &lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**
This converts between a `WIN-PROLOG` Window handle and its Raw, integer counterpart. When Window is an unbound variable, Raw must be an integer: an attempt is made to check that this integer identifies a window, and if not the predicate fails. If successful, a further test is made to see whether the window was created by `WIN-PROLOG`: if so, its logical window handle is bound to Window; otherwise, the raw window handle is returned. If Window is a `WIN-PROLOG` window handle, a check is made to see if it defines a currently-existing window; if so, the raw handle of that window is unified with Raw. In both cases, `wndhdl/2` fails if the specified window does not exist. A logical window handle can take any of the following forms:

<table>
<thead>
<tr>
<th>Handle</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the <code>WIN-PROLOG</code> main window</td>
</tr>
<tr>
<td>1</td>
<td>the <code>WIN-PROLOG</code> console window</td>
</tr>
<tr>
<td>2</td>
<td>the <code>WIN-PROLOG</code> banner window</td>
</tr>
<tr>
<td>3</td>
<td>the <code>WIN-PROLOG</code> tooltip window</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>apart from the above, all remaining integers are treated as raw window handles</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>the name of a window created by wcreate/8 or one of its variants</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>this contains two or more elements, the first of which is one of the above types, and the remainder of which are integers representing the nested ID of one of its controls</td>
</tr>
</tbody>
</table>

**Examples**
The following call returns the raw handle of `WIN-PROLOG`'s "main window" ("0"):

```
?- wndhdl( 0, H ). <enter>
H = 6428
```

**Notes**
The `wndhdl/2` predicate performs two primary functions: firstly, it provides a quick test for the existence of any given window: "wndhdl( Window, _ )" will simply succeed or fail depending upon whether or not the window exists. Secondly, it converts window handles to and from their "raw" integer state. Most predicates within `WIN-PROLOG` work with "logical" window handles; however, external functions called
with \textit{winapi/4} cannot interpret such handles. Consider, for example, a simplified implementation of the \textit{sndmsg/5} predicate:

\begin{verbatim}
sndmsg( Window, Message, Wparam, Lparam, Result ) :-
  wndhdl( Window, Handle ),
  message_name( Message, Number ),
  winapi( (user32,'SendMessageA'), [Handle,Number,Wparam,Lparam], 1, Result ).
\end{verbatim}

The \textit{winapi/4} call to "SendMessageA" requires a window handle, but there is no point in trying to pass a \textbf{WIN-PROLOG} "logical" handle, because Windows' "user32" module could not interpret it. A call is therefore made to \textit{wndhdl/2} to convert the given \textit{Window} into a raw \textit{Handle}, and it is the latter that is used in the call to \textit{winapi/4}.

A third, special use of \textit{wndhdl/2} is also tied into \textit{winapi/4}: a number of Windows API functions require something called the process "Instance", which identifies a specific incarnation of an application. By using the special "window" handle of ",-1", this elusive value can be picked up:

\begin{verbatim}
?- wndhdl( -1, I ).
I = 4406
\end{verbatim}

This is one of a small family of \textbf{WIN-PROLOG} window handling predicates directly implemented in the assembler-coded kernel, enabling special low-level housekeeping functions to be handled efficiently. Most of the other window handling predicates are implemented as a "thin layer" over the all-powerful \textit{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \textit{winapi/4} and its helper predicate, \textit{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
word_wrap/1

get or set the word wrap to window mode

    word_wrap( Mode )

?Mode <variable> or <integer> in the range [0..1]

Comments
This gets or sets the word wrap Mode of the console and other text windows. When specified as a variable, Mode is bound to an integer giving the current setting; when specified as an integer, it defines the new Mode, causing word wrap to be turned on or off, as follows:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>disable word wrap to window</td>
</tr>
<tr>
<td>1</td>
<td>enable word wrap to window</td>
</tr>
</tbody>
</table>

Examples
The following call returns the current word wrap mode:

    ?- word_wrap( Mode ).
    Mode = 0

The next call turns on word wrap to window:

    ?- word_wrap( 1 ).
    yes

Notes
The "Word Wrap to Window" mode in WIN-PROLOG causes long lines of text to wrap at the edge of the window, much as in word-processor programs; when it is disabled, long lines extend to an arbitrary distance typically of thousands of characters of width, requiring use of horizontal scrolling to view them. The actual contents of windows are completely unaffected by this feature; its sole purpose is to change the display while editing or running Prolog queries. With one exception, the contents are also identical with respect to WIN-PROLOG programs: this one exception is the wrchpxy/4 predicate, which converts position logical offsets within a file into physical X/Y (column and row) counts: as lines wrap, and the wrapping changes as windows are resized, this mapping changes to reflect the displayed text.
**wrange/5**

*get or set the range of a scrollbar control*

**wrange( Window, Flag, Lower, Upper, Page )**

- **+Window** <window_handle>
- **+Flag** <integer> in the range [0..2]
- **?Lower** <integer> or <variable>
- **?Upper** <integer> or <variable>
- **?Page** <integer> or <variable>

**Comments**

This gets or sets the **Lower** and **Upper** limits of the range of a "scrollbar" control, together with its **Page** increment, as identified by a combination of the given **Window** and **Flag**. If any or all of **Lower**, **Upper** and **Page** are unbound variables, they are bound to the existing lower limit, upper limit and/or page increment accordingly; otherwise they are used to set these limits or increment. This predicate can handle windows that are scrollbar controls, as well as other types of windows that contain scrollbars: **Flag** specifies which case to process:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>process the window as a scrollbar</td>
</tr>
<tr>
<td>1</td>
<td>process the window's horizontal scrollbar</td>
</tr>
<tr>
<td>2</td>
<td>process the window's vertical scrollbar</td>
</tr>
</tbody>
</table>

**Examples**

The following program uses **wdcreate/7** and **wccreate/8** to create a small dialog containing a single horizontal scrollbar:

```
foo :-
    Dstyle = [ws_visible, ws_popup, ws_caption],
    Sstyle = [ws_visible, ws_child, sbs_horz],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), scrollbar, ``, 10, 10, 80, 20, Sstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen:

```
?- foo.
<enter>
yes
```
At this point, any attempt to move the scrollbar will fail, because both limits of its range are initially set to zero ("0"); this can be shown in the next call:

```
?- wrange( (foo,1), 0, L, U, P ).
L = U = P = 0
```

The next two calls will give the scrollbar a range of [0..100], with a page increment of 10, and set the thumb to "25", a quarter of the way along the scrollbar:

```
?- wrange( (foo,1), 0, 0, 100, 10 ).
yes

?- wthumb( (foo,1), 0, 25 ).
yes
```

Finally, after the scrollbar has been moved by the user, its thumb position can be queried:

```
?- wthumb( (foo,1), 0, T ).
T = 81
```

**Notes**

Up to and including **WIN-PROLOG 5.000**, the predicate, `wrange/4`, was used to set up or check the range of a scrollbar; there was no support for setting or getting the page increment value, which was simply assumed to be the pixel height or width of the client area of the window which contained the scroll bars. The new predicate provides considerable extra flexibility with its page increment argument.

Two predicates, `wsclrng/3` and `wsclpos/2`, provide support for "scrollbar" controls. Unlike other types of control (buttons, listboxes, etc), scrollbars can either occur as stand-alone windows (see the above example) or as features built in to other windows. When handling stand-alone scrollbars, you can use the above two predicates; when handling the automatically generated scrollbars within other windows, instead you should use `wrange/5` and `wthumb/3` respectively.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
perform or check a clipboard function in a rich control window

\[ \text{wrchclp}( \text{Window}, \text{Function} ) \]

+Window \hspace{1cm} <window_handle>
+Function \hspace{1cm} <integer> in the range \{+/- 1..6\}

Comments
This performs or checks for the given standard clipboard Function with respect to the given rich Window. When Function is a positive integer, it performs one of the following clipboard functions:

<table>
<thead>
<tr>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cut</td>
</tr>
<tr>
<td>2</td>
<td>Copy</td>
</tr>
<tr>
<td>3</td>
<td>Paste</td>
</tr>
<tr>
<td>4</td>
<td>Clear</td>
</tr>
<tr>
<td>5</td>
<td>Undo</td>
</tr>
<tr>
<td>6</td>
<td>Redo</td>
</tr>
</tbody>
</table>

When Function is a negative integer, this predicate succeeds or fails depending upon whether the corresponding (positive) function would succeed if performed.

Examples
Consider a "text" window called "foo" whose rich control contains the following text:

\[
\text{‘Twas brillig, and the slithy toves}
\text{Did gyre and gimble in the wabe;}
\text{All mimsy were the borogoves,}
\text{And the mome raths outgrabe.}
\]

The following call will select the whole of its first line:

\[ ?- \text{wrchsel}( (\text{foo},1), 0, 35 ) . \]
\[ \text{yes} \]
Once selected, this text can be copied from "foo" into the Windows clipboard, and then pasted from this into the "console window" with the next command:

```prolog
?- wrchclp( (foo,1), 2 ), wrchclp( (1,1), 3 ), nl.
    'Twas brillig, and the slithy toves
    yes
```

**Notes**

The `wrchclp/2` predicate can only perform clipboard functions using the standard "text" clip type, and only works with "edit" controls. Other predicates, such as `wtext/2`, can be used to process the text in other classes of window.

Note that the console window's rich control, "(1,1)"), is designed to allow updates only in its final portion, the "input zone", which is the area after the final prompt or output. Any attempt to update earlier parts of the console will fail, other than when setting the entire text of this window, for example with a call to `wtext/2`.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
**wrchfnd/6**

find a text string in a rich control window

```
wrchfnd( Window, From, To, String, Start, Finish )
```

+Window  <window_handle>
+From     <integer>
+To        <integer>
+String    <string>
?Start     <variable> or <integer>
?Finish    <variable> or <integer>

**Comments**
This searches rich control Window for the given text String between the offsets From and To, unifying the start and finish offsets of any match with Start and Finish respectively. As a special case, the String may be specified as the empty string ('`'), in which case the Top and End of the next space-delimited token is returned. This predicate fails if no match is found within the specified limits.

**Examples**
Consider a "text" window called "foo" whose rich control contains the following text:

'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.

The following call returns the start and finish offsets of the first instance of the word, "gimble":

```
?- wrchfnd( (foo,1), 0, 9999, `gimble`, S, F ).
```

S = 51 ,
F = 57

**Notes**
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned Start and Finish parameters may be passed directly into wrchsel/3 in order to highlight the matched string.

Please note that wrchfnd/6 only performs exact-case (case sensitive) searches. The find/3 predicate, on the other hand, can be used to perform both case sensitive and case insensitive searches, and thanks to WIN-PROLOG's string and memory file input/output predicates, the latter predicate can easily be used in conjunction with text searches within windows.
get plain or rich text from a rich control into a memory file

\[ \textbf{wrchget( Window, Mode, Buffer )} \]

+\textit{Window} \hspace{2cm} \textless \textit{window\_handle}\textgreater 
+\textit{Mode} \hspace{2cm} \textless \textit{integer} \textgreater \text{ in the domain } \{-2,-1,0,1,2,3\}
+\textit{Buffer} \hspace{2cm} \textless \textit{atom}\textgreater 

Comments

This gets the text in the given rich \textit{Window}, according to the given \textit{Mode}, and stores it in a memory file \textit{Buffer}. Any existing file with the name, \textit{Buffer}, is closed, and a new one created of sufficient size to contain the indicated text in Unicode format (see \textit{wrchtxt/3} for a description of modes).

Examples

Consider a "text" window called "foo" whose rich control contains the following text:

\begin{verbatim}
'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.
\end{verbatim}

The following call will read the entire window into a memory file called "myfile":

?\(- \text{ wrchget( (foo,1), 0, myfile ).} \text{ \hspace{2cm} <enter>}
yes

The next call reads the first line from the memory file:

\[ ?\(- \text{ input( myfile ), fread( s, 0, 0, Line ), input( 0 ).} \text{ \hspace{2cm} <enter>}
\text{Line = '}'Twas brillig, and the slithy toves~M~J' \]

Notes

A related set of predicates, \textit{wrchget/3}, \textit{wrchlen/3}, \textit{wrchlod/3}, \textit{wrchsav/3}, \textit{wrchset/3} and \textit{wrchtxt/3}, share a common "Mode" argument which allows them to be used in various ways. Modes 1 and 2 treat the entire window text as plain (Unicode) or rich (RTF) data, while modes 3 and 4 do the same for the selected area only. Two special modes, -1 and -2 bypass the normal text filtering routines to handle text in its "Raw" mode: in this state, end of line is marked by a single <cr> character, rather than the usual <cr><lf> pair. These latter two modes tie in best with with \textit{wrchfnd/6}, \textit{wrchlin/4}, \textit{wrchpxy/4} and \textit{wrchsel/3}, while the former four modes work best in conjunction with files.
wrchlen/3

gain length of plain or rich text in a rich control

\[
\text{wrchlen( Window, Mode, Length )}
\]

+Window    \text{<window_handle>}
+Mode      \text{<integer> in the domain \{-2,-1,0,1,2,3\}}
-Length    \text{<variable>}

Comments
This gets the length of text in the given \text{Window}, according to the given \text{Mode}, and bind the result character count to \text{Length} (see \text{wrchtxt/3} for a description of modes).

Examples
Consider a "text" window called "foo" whose rich control contains the following text:

\[
\begin{align*}
&T was brillig, and the slithy toves
&D id gyre and gimble in the wabe;
&A ll mimsy were the borogoves,
&A nd the mome raths outgrabe.
\end{align*}
\]

The following call will return the length, in characters, of the entire window, assuming Unicode encoding:

?- \text{wrchlen( (foo,1), 0, Length ).}
\text{Length = 136}

The next call returns the same length, but this time assuming RTF encoding:

?- \text{wrchlen( (foo,1), 1, Length ).}
\text{Length = 452}

Notes
A related set of predicates, \text{wrchget/3}, \text{wrchlen/3}, \text{wrchlod/3}, \text{wrchsav/3}, \text{wrchset/3} and \text{wrchtxt/3}, share a common "Mode" argument which allows them to be used in various ways. Modes 1 and 2 treat the entire window text as plain (Unicode) or rich (RTF) data, while modes 3 and 4 do the same for the selected area only. Two special modes, -1 and -2 bypass the normal text filtering routines to handle text in its "Raw" mode: in this state, end of line is marked by a single \text{<cr>} character, rather than the usual \text{<cr><lf>} pair. These latter two modes tie in best with with \text{wrchfnd/6}, \text{wrchlin/4}, \text{wrchpxy/4} and \text{wrchset/3}, while the former four modes work best in conjunction with files.
wrchlin/4

get or check the offsets of a line in a rich control window

wrchlin( Window, Offset, Start, Finish )

+Window <window_handle>
+Offset <integer>
?Start <variable> or <integer>
?Finish <variable> or <integer>

Comments
This picks up the start and finish offsets of the line of text which includes the given Offset within the given rich Window, and unifies these with Start and Finish respectively. The returned values include everything on the given line, excluding the terminating <cr><lf> pair.

Examples
Consider a "text" window called "foo" whose rich control contains the following text:

'Twas brillig, and the slithy toves
    Did gyre and gimble in the wabe;
All mimsy were the borogoves,
    And the mome raths outgrabe.

The following call returns the start and finish offsets of the line of text which includes character offset "81":

?- wrchlin( (foo,1), 81, S, F ).  <enter>
S = 71 ,
F = 100

Notes
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned Start and Finish parameters may be passed directly into wrchsel/3 in order to highlight the matched line.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wrchlod/3**

load plain or rich text from disk file into a rich control

\[ wrchlod( \text{Window, Mode, File} ) \]

+Window  \(<\text{window_handle}>\)
+Mode  \(<\text{integer}>\) in the domain \{-2,-1,0,1,2,3\}
+File  \(<\text{atom}>\)

**Comments**

This loads the text in the given disk `File`, according to the given `Mode`, into a rich `Window`. The file may be in RTF, ISO/IEC 8859-1 or UTF-16LE format as appropriate (see `wrchttx/3` for a description of modes).

**Examples**

Consider a disk file, "myfile.txt", which contains the following text in ISO/IEC 8859-1 format:

'Twas brillig, and the slithy toves  
    Did gyre and gimble in the wabe;  
    All mimsy were the borogoves,  
    And the mome raths outgrabe.

The following call will load an entire text window, called "foo", with the text of this file:

\[ ?- \text{wrchlod( (foo,1), 0, 'myfile.txt' ).} \]

yes

The next call will do a similar job, but this time loading a rich text (.RTF) file:

\[ ?- \text{wrchlod( (foo,1), 1, 'myfile.rtf' ).} \]

yes

**Notes**

A related set of predicates, `wrchget/3`, `wrchlen/3`, `wrchlod/3`, `wrchsav/3`, `wrchset/3` and `wrchttx/3`, share a common "Mode" argument which allows them to be used in various ways. Modes 1 and 2 treat the entire window text as plain (Unicode) or rich (RTF) data, while modes 3 and 4 do the same for the selected area only. Two special modes, -1 and -2 bypass the normal text filtering routines to handle text in its "Raw" mode: in this state, end of line is marked by a single `<cr>` character, rather than the usual `<cr>` `<lf>` pair. These latter two modes tie in best with with `wrchfnd/6`, `wrchlin/4`, `wrchpxy/4` and `wrchsel/3`, while the former four modes work best in conjunction with files.
wrchpxy/4

convert between offsets and (x,y) coords in a rich window

\[
\text{wrchpxy( Window, Offset, Xpos, Ypos )}
\]

+Window <window_handle>
?Offset <variable> or <integer>
?Xpos <variable> or <integer>
?Ypos <variable> or <integer>

Comments
This converts between linear offsets and (x,y) coordinates in the given rich Window. If Offset is an unbound variable, Xpos and Ypos must be bound to integers which specify the column and row coordinates of a character in the Window: its linear offset is computed and bound to Offset. If Offset is an integer, the corresponding (x,y) coordinates of the indicated character are computed and unified with Xpos and Ypos respectively.

Examples
Consider a "text" window called "foo" whose rich control contains the following text:

'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
   All mimsy were the borogoves,
       And the mome raths outgrabe.

The following call returns the offset of character number "20" in line "2" (this is the "b" in "borogoves"):

?- wrchpxy( (foo,1), P, 20, 2 ).
P = 91

The next call performs the reverse operation:

?- wrchpxy( (foo,1), 91, X, Y ).
X = 20 ,
Y = 2

Notes
This predicate does not perform any side effects: it is used for the gathering of information only; however, the returned Offset parameter may be passed directly into wrchsel/3 in order to highlight the start or end of a text block. When converting from (X,Y) to linear offset,
both values are automatically limited to their maximum values for the given text line (Xpos) or line number (Ypos). Please note that all offsets are numbered from zero ("0").
**wrchsav/3**

_save plain or rich text from a rich control into a disk file_

\[
\text{wrchsav( Window, Mode, File )}
\]

+Window   <window_handle>
+Mode      <integer> in the domain \{-2,-1,0,1,2,3\}
+File      <atom>

**Comments**

This saves the text in the given rich Window, according to the given Mode, into a named disk File. Any existing file with the name, File, is deleted, and a new one created to contain the indicated text in ISO/IEC 8859-1 or UTF-16LE format as appropriate (see wrchtxt/3 for a description of modes).

**Examples**

Consider a "text" window called "foo" whose rich control contains the following text:

'Twas brillig, and the slithy toves  
Did gyre and gimble in the wabe;  
All mimsy were the borogoves,  
And the mome raths outgrabe.

The following call will save the entire window into a plain text disk file called "myfile.txt":

\[
?- \text{wrchsav( (foo,1), 0, 'myfile.txt' ).}
\]

yes

The next call will do a similar job, but this time creating a rich text (.RTF) file:

\[
?- \text{wrchsav( (foo,1), 1, 'myfile.rtf' ).}
\]

yes

**Notes**

A related set of predicates, wrchget/3, wrchlen/3, wrchlod/3, wrchsav/3, wrchset/3 and wrchtxt/3, share a common "Mode" argument which allows them to be used in various ways. Modes 1 and 2 treat the entire window text as plain (Unicode) or rich (RTF) data, while modes 3 and 4 do the same for the selected area only. Two special modes, -1 and -2 bypass the normal text filtering routines to handle text in its "Raw" mode: in this state, end of line is marked by a single \(<cr>\) character, rather than the usual \(<cr><lf>\) pair. These latter two modes tie in best with with wrchfnd/6, wrchlin/4, wrchpxy/4 and wrchsel/3, while the former four modes work best in conjunction with files.
wrchsel/3

get or set the selection area in a rich control window

\[ \text{wrchsel( Window, Start, Finish )} \]

+Window \quad \langle \text{window_handle} \rangle

?Start \quad \langle \text{variable} \rangle \text{ or } \langle \text{integer} \rangle

?Finish \quad \langle \text{variable} \rangle \text{ or } \langle \text{integer} \rangle

Comments
This gets or sets the \textit{Start} and \textit{Finish} offsets of the text selection in the given rich \textit{Window}. If either or both of \textit{Start} and \textit{Finish} is an unbound variable, it is bound to an integer containing the current value; otherwise, the rich window is side-effected to set the \textit{Start} and/or \textit{End} of its selection accordingly.

Examples
Consider a "text" window called "foo" whose rich control contains the following text:

\begin{verbatim}
'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.
\end{verbatim}

The following call will select the whole of the third line:

\[ \text{?- wrchsel( (foo,1), 71, 100 ).} \quad \text{<enter>} \]

yes

Notes
The \textit{wrchsel/3} predicate is used together with \textit{wrchfnl/6}, \textit{wrchlin/4} and \textit{wrchpxy/4} to change or query the text selection area; together with \textit{wrchtxt/3} it can also be used to replace selected portions of text in rich control windows.

This is one of a family of \textbf{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \textit{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \textit{winapi/4} and its helper predicate, \textit{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
**wrchset/3**

set plain or rich text into a rich control from a memory file

```
wrchset( Window, Mode, Buffer )
```

+Window <window_handle>
+Mode <integer> in the domain {-2,-1,0,1,2,3}
+Buffer <atom>

**Comments**

This sets the text in the given rich Window, according to the given Mode, from a memory file Buffer that contains text in Unicode format. After loading, the file, Buffer, is closed (see wrchtxt/3 for a description of modes).

**Examples**

Consider a "text" window called "foo" whose rich control contains the following text:

'\text{Twas brillig, and the slithy toves}
\hspace{1em}Did gyre and gimble in the wabe;
\hspace{1em}All mimsy were the borogoves,
\hspace{1em}And the mome raths outgrabe.

The following call will read the entire window into a memory file called "myfile", in rich text (.RTF) format:

```
?- wrchget( (foo,1), 1, myfile ).
yes
```

The next call store this text in a second text window, "bar", complete with all text formatting, before closing "myfile":

```
?- wrchset( (bar,1), 1, myfile ).
yes
```

**Notes**

A related set of predicates, *wrchget/3, wrchlen/3, wrchlod/3, wrchsav/3, wrchset/3 and wrchtxt/3*, share a common "Mode" argument which allows them to be used in various ways. Modes 1 and 2 treat the entire window text as plain (Unicode) or rich (RTF) data, while modes 3 and 4 do the same for the selected area only. Two special modes, -1 and -2 bypass the normal text filtering routines to handle text in its "Raw" mode: in this state, end of line is marked by a single `<cr>` character, rather than the usual `<cr>` `<lf>` pair. These latter two modes tie in best with *wrchfnd/6, wrchlin/4, wrchpxy/4 and wrchsel/3*, while the former four modes work best in conjunction with files.
wrchtxt/3

get or set plain or rich text in a rich control window

wrchtxt( Window, Mode, Text )

+Window <window_handle>
+Mode <integer> in the domain {-2,-1,0,1,2,3}
?Text <variable> or <string>

Comments
This gets or sets the Text in the given rich Window, according to the given Mode. If Text is an unbound variable, it is bound to a string containing the current text in the window; otherwise, the rich window is side-effected to replace the existing text with the given Text string, according to the Mode setting:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>process whole window as plain text (Unicode)</td>
</tr>
<tr>
<td>1</td>
<td>process whole window as rich text (RTF)</td>
</tr>
<tr>
<td>2</td>
<td>process selected area as plain text (Unicode)</td>
</tr>
<tr>
<td>3</td>
<td>process selected area as rich text (RTF)</td>
</tr>
<tr>
<td>-1</td>
<td>process whole window as raw text (no &lt;LF&gt;)</td>
</tr>
<tr>
<td>-2</td>
<td>process selected area as raw text (no &lt;LF&gt;)</td>
</tr>
</tbody>
</table>

Examples
Consider a "text" window called "foo" whose rich control contains the following text:

'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.

The following call will select and return the word "brillig":

?- wrchsel( (foo,1), 6, 13 ), wrchtxt( (foo,1), 2, T ). <enter>
T = `brillig`
The next call replaces this word with "squirmy":

?- \texttt{wrchtxt((foo,1), 2, \textquoteright squirmy\textquoteright)}.\texttt{\textbackslash \textbackslash enter}\texttt{\textbackslash \textbackslash yes}\texttt{<enter>}

This call will reselect the word, "squirmy", and then obtain its text in rich text (RTF) format:

?- \texttt{wrchsel((foo,1), 6, 13), wrchtxt((foo,1), 3, T)}.\texttt{\textbackslash \textbackslash enter}\texttt{T = '{\textquotedbl}rtf1\textbackslash ansi\textbackslash ansicpg1252\textbackslash deff0...squirmy\textbackslash \textbackslash M\textbackslash \textbackslash J'{\textbackslant enter}}

\textbf{Notes}

The \texttt{wrchtxt/3} predicate is used together with \texttt{wrchfnd/6}, \texttt{wrchlin/4}, \texttt{wrchpxy/4} and \texttt{wrchsel/3}, to locate, select, obtain and replace text in rich control windows; \texttt{wtext/2} can be used to set or get the entire text content of any type of window.

A related set of predicates, \texttt{wrchget/3}, \texttt{wrchlen/3}, \texttt{wrchlod/3}, \texttt{wrchsav/3}, \texttt{wrchset/3} and \texttt{wrchtxt/3}, share a common "Mode" argument which allows them to be used in various ways. Modes 1 and 2 treat the entire window text as plain (Unicode) or rich (RTF) data, while modes 3 and 4 do the same for the selected area only. Two special modes, -1 and -2 bypass the normal text filtering routines to handle text in its "Raw" mode: in this state, end of line is marked by a single <cr> character, rather than the usual <cr><lf> pair. These latter two modes tie in best with with \texttt{wrchfnd/6}, \texttt{wrchlin/4}, \texttt{wrchpxy/4} and \texttt{wrchsel/3}, while the former four modes work best in conjunction with files.

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
write/1

write a term to the current output stream unquoted

    write( Term )

?Term <term>

Comments  This writes the given Term to the current output stream, in unquoted Edinburgh syntax.

Examples  The following command outputs the given term without quotes, so that atoms and strings appear the same:

?- write( `String`+'Atom'=Var ), nl.
String + Atom = _1
Var = _

Notes  This predicate outputs terms using "unquoted" Edinburgh syntax, which means that neither strings nor atoms are delimited by quotes. Additionally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are output literally:

?- write( 'hello~M~Jworld' ), nl.
hello
world
yes

Terms output to a file in unquoted syntax are not generally readable, even if they are followed by a period (.) and at least one space or control character, because unquoted strings and atoms can be confused with each other and even with variables, or may contain characters which give rise to syntax errors.

This predicate always writes unbound variables using the form, "_nnn", where "nnn" is a numerical value; if it is desired to output terms with specific variable names, use can be made of the "ewrite/n" predicates: ewrite/1 is entirely equivalent to write/1, but forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output with correct bracketing with respect to any operator precedences (see ewrite/2, eprint/2 and eprint/3 respectively).
**write_canonical/1**

`write a term to current output quoted, without operators`

```
write_canonical( Term )

?Term <term>
```

**Comments**

This writes the given `Term` to the current output stream, in quoted Edinburgh syntax, without the use of operators.

**Examples**

The following command outputs the given term with quotes where necessary:

```
?- write_canonical( `String`+'Atom'=Var ), nl.
=+(`String`,'Atom'),_1
Var = _
```

**Notes**

This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (``), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes ('). Finally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see `eprint/1`).

```
?- write_canonical( 'hello~M~Jworld' ), nl.
'hello~M~Jworld'
yes
```

Terms output to a file in quoted syntax are readable provided they are followed by a period (.) and at least one space or control character. This is *not* provided automatically, and must therefore be output explicitly if readability is required:

```
?- write_canonical( foo(X) ), ewrite( '.' ), nl.
foo(_1).
X = _
```

This predicate always writes unbound variables using the form, "_nnn", where "nnn" is a numerical value; if it is desired to output terms with specific variable names, use can be made of the "eprint/n" predicates: `eprint/1` is entirely equivalent to `writeq/1`, but forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output with correct bracketing with respect to any operator precedences (see `eprint/2` and `eprint/3`).
writeq/1

write a term to current output quoted

\[\text{writeq}(\text{Term})\]

?Term                  <term>

Comments
This writes the given Term to the current output stream, in quoted Edinburgh syntax.

Examples
The following command outputs the given term with quotes where necessary:

\[?-\text{writeq('}\text{String}'+\text{'}\text{Atom}'=\text{Var}),\text{nl.}\]

\`
\text{String}'+\text{'}\text{Atom}' = _1
\text{Var} = _
\`

Notes
This predicate outputs terms using "quoted" Edinburgh syntax, which means that any strings are delimited by backward quotes (\`), and any atoms beginning with the underscore (_), uppercase letters or containing a mix of lexical types are delimited by single quotes (\'). Finally, any control characters (those with character codes in the range 00h..21h) within strings or atoms are shown using "tilde" notation (see eprint/1).

\[?-\text{writeq('}\text{hello-M-Jworld' ),}\text{nl.}\]

\`
\text{hello-M-Jworld}
\text{yes}
\`

Terms output to a file in quoted syntax are readable provided they are followed by a period (.) and at least one space or control character. This is not provided automatically, and must therefore be output explicitly if readability is required:

\[?-\text{writeq( }\text{foo(X) },\text{ewrite(')}\text{.' },\text{nl.}\]

\`
\text{foo(_1).}
\text{X = _}
\`

This predicate always writes unbound variables using the form, "_nnn", where "nnn" is a numerical value; if it is desired to output terms with specific variable names, use can be made of the "eprint/n" predicates: eprint/1 is entirely equivalent to writeq/1, but forms part of a family of more advanced term output predicates that allow variables to be named and subterms to be output with correct bracketing with respect to any operator precedences (see eprint/2 and eprint/3).
wsclpos/2

get or set the position of a scrollbar control

wsclpos( Window, Position )

+Window <window_handle>

?Position <integer> or <variable>

Comments This gets or sets the Position of a "scrollbar" control Window. If Position is an unbound variable, it is bound to the existing position; otherwise it is used to set the position. This predicate handles "scrollbar" controls only: to process scrollbars within other windows, see wthumb/3.

Examples The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single horizontal scrollbar:

foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Sstyle = [ws_visible,ws_child,sbs_horz],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), scrollbar, ` `, 10, 10, 80, 20, Sstyle ).

Once compiled, the following call will display the dialog near the top left corner of the screen:

?- foo.  <enter>
yes
At this point, any attempt to move the scrollbar will fail, because both limits of its range are initially set to zero ("0"); this can be shown in the next call:

?- wsclrng( (foo,1), L, U ).  <enter>
L = U = 0

The next two calls will give the scrollbar a range of [0..100], and set the position to "25", a quarter of the way along the scrollbar:

?- wsclrng( (foo,1), 0, 100 ).  <enter>
yes
Finally, after the scrollbar has been moved by the user, its position can be queried:

?- wsclpos( (foo,1), T ).

T = 81

Notes

Two predicates, wsclrng/3 and wsclpos/2, provide support for "scrollbar" controls. Unlike other types of control (buttons, listboxes, etc), scrollbars can either occur as stand-alone windows (see the above example) or as features built in to other windows. When handling stand-alone scrollbars, you can use the above two predicates; when handling the automatically generated scrollbars within other windows, instead you should use wrange/5 and wthumb/3 respectively. Furthermore, wrange/5 provides access to the "page increment" of a scrollbar, which determines how far it increases or decreases its value when the mouse is clicked in the zones between the thumb itself and the end arrows.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wsclrng/3

get or set the range of a scrollbar control

\[ wsclrng( \text{Window, Lower, Upper} ) \]

+Window <window_handle>

?Lower <integer> or <variable>

?Upper <integer> or <variable>

Comments  This gets or sets the Lower and Upper limits of the range of a "scrollbar" control Window. If either or both of Lower and Upper are unbound variables, they are bound to the existing lower and/or upper limits accordingly; otherwise they are used to set these limits. This predicate handles "scrollbar" controls only: to process scrollbars within other windows, see wrange/5.

Examples  The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single horizontal scrollbar:

\[
\text{foo} \leftarrow \ \\
\text{Dstyle} = [\text{ws_visible,ws_popup,ws_caption}], \ \\
\text{Sstyle} = [\text{ws_visible,ws_child,sbs_horz}], \ \\
\text{wdcreate}((\text{foo,`foo'},10,10,100,60,\text{Dstyle})), \ \\
\text{wccreate}((\text{foo,1}),\text{scrollbar,``},10,10,80,20,\text{Sstyle}).
\]

Once compiled, the following call will display the dialog near the top left corner of the screen:

\[
?- \text{foo}.
\]

yes

At this point, any attempt to move the scrollbar will fail, because both limits of its range are initially set to zero ('0'); this can be shown in the next call:

\[
?- \text{wsclrng((foo,1), L, U)}.
\]

\[
L = U = 0
\]

The next two calls will give the scrollbar a range of [0..100], and set the position to "25", a quarter of the way along the scrollbar:

\[
?- \text{wsclrng((foo,1), 0, 100)}.
\]
Finally, after the scrollbar has been moved by the user, its position can be queried:

?- \text{wsclpos(}\text{foo,1)}, T\text{).} \quad \text{<enter>}
T = 81

**Notes**

Two predicates, \text{wsclrng/3} and \text{wsclpos/2}, provide support for "scrollbar" controls. Unlike other types of control (buttons, listboxes, etc), scrollbars can either occur as stand-alone windows (see the above example) or as features built in to other windows. When handling stand-alone scrollbars, you can use the above two predicates; when handling the automatically generated scrollbars within other windows, instead you should use \text{wrange/5} and \text{wthumb/3} respectively.

This is one of a family of \text{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \text{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \text{winapi/4} and its helper predicate, \text{wintxt/4}, can be used to access virtually any function in the \text{Windows API} as well as those implemented in third-party DLLs.
**wshow/2**

*get or set the show status of a window*

```prolog
wshow( Window, Status )
```

+Window <window_handle>

?Status <variable> or <integer> in the range [0..3]

**Comments**

This gets or sets the "show status" of the given Window: if Status is an unbound variable, it returns the existing show status; otherwise, it is used to show or hide the Window. The following table shows the values and meanings of Status:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hide the window</td>
</tr>
<tr>
<td>1</td>
<td>show the window normalised</td>
</tr>
<tr>
<td>2</td>
<td>show the window minimised</td>
</tr>
<tr>
<td>3</td>
<td>show the window maximised</td>
</tr>
</tbody>
</table>

**Examples**

The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single "default push" button:

```prolog
foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Bstyle = [ws_visible,ws_child,bs_defpushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ).
```

Once compiled, the following call will display the dialog briefly near the top left corner of the screen, before hiding it:

```prolog
?- foo, wshow( foo, 0 ).
yes
```

In order to make the dialog visible, it must be shown; once this is done, it can be rehidden by clicking the button:

```prolog
?- wshow( foo, 1 ).
yes
```
Notes

When `wshow/2` is used to hide a window, all of its child windows (controls, etc.) are also hidden (see the first example above); however, it is also possible to hide individual child windows independently of the parent dialog. In this way, individual buttons, edit controls, etc., can be "removed" when inappropriate, and reshown when required. Note that hiding the WIN-PROLOG "main window" also hides the "console window" together with any program windows, but not any pop-up dialogs. In order to avoid the user being permanently locked out if, say, a program crashed during development while the console was hidden, the primary WIN-PROLOG windows are automatically shown and enabled whenever an attempt is made to read from the console ("user") input device.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
get or set the position and size of a window

wsizexx{5}{wsize}/5 

\[
\text{wsizexx{5}{wsize}( Window, Left, Top, Width, Depth )}
\]

+Window \quad <\text{window_handle}>

?Left \quad <\text{integer}> or <\text{variable}>

?Top \quad <\text{integer}> or <\text{variable}>

?Width \quad <\text{integer}> or <\text{variable}>

?Depth \quad <\text{integer}> or <\text{variable}>

Comments
This gets or sets the position and size of a given Window, specified by its Left and Top screen coordinates and Width and Height dimensions. If any of these latter four arguments are unbound variables, they are bound to the appropriate current value; otherwise, they are used to modify the corresponding window parameters. All numerical values are in pixel units, where (0,0) is the left top of the screen.

Examples
Running on an XGA resolution screen (1024*768), the following call picks up the current position and sets a new size for the WIN-PROLOG "main window" (window "0"):

\[
?\text{-} \text{wsize}(0, L, T, 800, 600).
\]

L = T = 0

Notes
The wsize/5 predicate can be used to get or change the position and size of a window; the related predicate, warea/5, provides useful information about the size of window's "client area".

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wstppos/2**

*get or set the position of a stripbar control*

```
wstppos( Window, Position )
```

+Window <window_handle>
?Position <integer> or <variable>

**Comments**

This gets or sets the *Position* of a "stripbar" (progress bar) control *Window*. If *Position* is an unbound variable, it is bound to the existing position; otherwise it is used to set the position.

**Examples**

The following program uses `wdcreate/7` and `wccreate/8` to create a small dialog containing a single horizontal stripbar, using the "smooth" (non-segmented) style:

```prolog
foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Sstyle = [ws_visible,ws_child,pbs_smooth],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), stripbar, ``, 10, 10, 80, 20, Sstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen:

```
?- foo.
yes
```

When first created, a stripbar has a default range of [0..100]; this can be shown in the next call:

```
?- wstprng( (foo,1), L, U ).
L = 0 ,
U = 100
```

The next two calls will change the stripbar range to [0..1000], and set the position to "250", a quarter of the way along the stripbar:

```
?- wstprng( (foo,1), 0, 1000 ).
yes
```
Two predicates, `wstprng/3` and `wstppos/2`, provide support for "stripbar" controls, which are based on the Windows "common control" class, "msctls_progress32", commonly known as "Progress Bars": see `wcreate/8` for more information.

This is one of a family of `WIN-PROLOG` window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the `Windows API` as well as those implemented in third-party DLLs.
wstprng/3

get or set the range of a stripbar control

wstprng( Window, Lower, Upper )

+Window              <window_handle>
?Lower               <integer> or <variable>
?Upper               <integer> or <variable>

Comments
This gets or sets the Lower and Upper limits of the range of a "stripbar" (progress bar) control Window. If either or both of Lower and Upper are unbound variables, they are bound to the existing lower and/or upper limits accordingly; otherwise they are used to set these limits.

Examples
The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single horizontal stripbar, using the "smooth" (non-segmented) style:

```prolog
foo :-
    Dstyle = [ws_visible, ws_popup, ws_caption],
    Sstyle = [ws_visible, ws_child, pbs_smooth],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), stripbar, ``, 10, 10, 80, 20, Sstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen:

```prolog
?- foo.
yes
```

When first created, a stripbar has a default range of [0..100]; this can be shown in the next call:

```prolog
?- wstprng( (foo,1), L, U ).
L = 0 ,
U = 100
```

The next two calls will change the stripbar range to [0..1000], and set the position to "250", a quarter of the way along the stripbar:

```prolog
?- wstprng( (foo,1), 0, 1000 ).
```
Notes

Two predicates, \textit{wstprng/3} and \textit{wstppos/2}, provide support for "stripbar" controls, which are based on the Windows "common control" class, "msctls_progress32", commonly known as "Progress Bars": see \textit{wcreate/8} for more information.

This is one of a family of \textbf{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \textit{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \textit{winapi/4} and its helper predicate, \textit{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
wstyle/2

get or set the style of a window

wstyle( Window, Style )

+Window <window_handle>
?Style <variable> or <integer>

Comments This gets or sets the "style" of the given Window: if Style is an unbound variable, it returns the existing style; otherwise, it is used to set the style of the Window (see Appendix C for more information about window styles).

Examples The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single "push" button:

foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Bstyle = [ws_visible,ws_child,bs_pushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ).

Once compiled, the following call will display the dialog near the top left corner of the screen:

?- foo.
  yes <enter>

Repositioning the WIN-PROLOG "main window" ("0") without clicking on the dialog button, so that the dialog can be seen while typing commands, the next call will change the appearance of the button to "default push" (a heavier border):

?- wstyle( (foo,1), 16'50000001 ).
  yes <enter>

Notes When wstyle/2 is used to set a window's style, care should be taken: not all window styles can be modified safely after a window has been created. In most cases, wstyle/2 should be used to "borrow" styles from existing windows when trying to create new windows of the same style (see wcreate/8). The example shown above, where a button's appearance is switched from "push" to "default push", is perfectly safe (as is the reverse operation), and is widely used in dialog programming. Not so recommended are calls such as the following, which in this case converts the existing button to an "auto checkbox":


?- wstyle( (foo,1), 16'50000003 ).
   yes

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wtabadd/4**

*add an item to a tabbox control*

```
wtabadd( Window, Position, String, Item )
```

+Window <window_handle>
+Position <integer>
+String <string>
+Item <integer>

**Comments**  
This adds a String, together with a user-defined numerical Item, to a "tabbox" control Window at the specified Position. The first entry in the control is numbered zero ("0").

**Examples**  
Consider a dialog called "foo" that includes a tabbox control with an ID of 9, and the following items:

```
aardvark  314  
badger    273  
dingo     981  
```

The following call will insert the string, "cheetah", with a user-defined item, "123", at the specified location between "aardvark" and "badger":

```
?- wtabadd( (foo,9 ), 1, 'cheetah', 123 ).  
yes
```

**Notes**  
A series of "wtab???/n" predicates provide support for "tabbox" controls, which are based on the Windows "common control" class, "SysTabControl32", commonly known as "Tab Controls": see wcreate/8 for more information.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wtabdel/2

delete an item from a tabbox control

wtabdel( Window, Position )

+Window <window_handle>
+Position <integer>

Comments
This deletes an entry in a "tabbox" control Window from the specified Position. The first entry in this type of control is numbered zero ("0").

Examples
Consider a dialog called "foo" that includes a tabbox control with an ID of 9 and containing the following items:

    aardvark  314
    badger    273
    dingo     981

The following call will delete the second entry ("badger"):?

    wtabdel( (foo,9 ), 1 ).

yes

Notes
A series of "wtab??/n" predicates provide support for "tabbox" controls, which are based on the Windows "common control" class, "SysTabControl32", commonly known as "Tab Controls": see wcreate/8 for more information.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
**wtabget/4**

*get an item from a tabbox control*

```
wtabget( Window, Position, String, Item )
```

- **+Window** `<window_handle>`
- **+Position** `<integer>`
- **-String** `<variable>`
- **-Item** `<variable>`

**Comments**  
This gets an item from a "tabbox" control *Window* at the specified *Position*, and binds its text to *String* and user-defined item value to *Item*. The first entry in this type of control is numbered zero ("0").

**Examples**  
Consider a dialog called "foo" that includes a tabbox control with an ID of 9 and containing the following items:

```
aardvark 314
badger   273
dingo    981
```

The following call will pick up the second entry ("badger", with an its item value):

```
?- wtabget( (foo,9 ), 1, S, I ). <enter>
S = `badger` ,
I = 273
```

**Notes**  
A series of "wtab???/n" predicates provide support for "tabbox" controls, which are based on the Windows "common control" class, "SysTabControl32", commonly known as "Tab Controls": see wcreate/8 for more information.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful *winapi/4* to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, *winapi/4* and its helper predicate, *wintxt/4*, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
**wtabrea/5**

get or check display area in a tabbox control

\[wtabrea( \text{Window}, \text{Left}, \text{Top}, \text{Width}, \text{Depth} )\]

+Window \text{<window_handle>}

?-Left \text{<integer> or <variable>}

?-Top \text{<integer> or <variable>}

?-Width \text{<integer> or <variable>}

?-Depth \text{<integer> or <variable>}

**Comments**

This gets the position and size of a given "tabbox" control Window's display area, unifying its (X,Y) screen coordinates with Left and Top, and its (W,D) dimensions with Width and Height. All numerical values are in pixel units, where (0,0) is the left top of the screen.

**Examples**

Consider a dialog called "foo" that includes a tabbox control with an ID of 9; the following call returns its display area:

?- wtabrea( (foo,9), L, T, W, D ).

L = 425 ,
T = 349 ,
W = 430 ,
D = 326

**Notes**

The wtabrea/5 predicate can be used to get or check the position and size of a tabbox control's display area, but not to alter it: this is the area in which a "tabbed dialog" page should be displayed. The tabbox control's display area can shrink as tabs items are added, depending upon the style in use (see Appendix C); it is not possible to adjust the display area directly: for this purpose it is necessary to use the related predicate, wsize/5.

A series of "wtab???/n" predicates provide support for "tabbox" controls, which are based on the Windows "common control" class, "SysTabControl32", commonly known as "Tab Controls": see wcreate/8 for more information.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wtabsel/2

get or set the selection in a tabbox control

\[
\text{wtabsel}( \text{Window, Position} )
\]

+Window <window_handle>
+Position <integer>

Comments
This gets or sets the selection \text{Position} in a "tabbox" control \text{Window}. The first entry in this type of control is numbered zero ("0"). If \text{Position} is an unbound variable, it returns the currently selected entry; if \text{Position} is an integer, it selects the indicated entry accordingly. A \text{Position} of ",-1" can be used to clear the selection, and if this value is returned, it indicates that no entry is currently selected.

Examples
Consider a dialog called "foo" that includes a tabbox control with an ID of 9 and containing the following items:

\begin{verbatim}
aardvark 314  
badger   273  
dingo    981  
\end{verbatim}

The following call will select the second entry ("badger"):

\begin{verbatim}
?- wtabsel( (foo,9 ), 1 ).  
yes
\end{verbatim}

Suppose the user has clicked on the "dingo" entry; the following call will return this selection:

\begin{verbatim}
?- wtabsel( (foo,9 ), P ).  
P = 2
\end{verbatim}

Notes
A series of "\text{wtab???/n}" predicates provide support for "tabbox" controls, which are based on the Windows "common control" class, "SysTabControl32", commonly known as "Tab Controls": see \text{wcreate/8} for more information.

This is one of a family of \text{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \text{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \text{winapi/4} and its helper predicate, \text{wintxt/4}, can be used to access virtually any function in the \text{Windows API} as well as those implemented in third-party DLLs.
**wtcreate/6**

create a text window

\[ wtcreate( \text{Window, Text, Left, Top, Width, Depth } ) \]

+Window <window_handle>
+Text <string>
+Left <integer>
+Top <integer>
+Width <integer>
+Depth <integer>

**Comments**
This creates a "text window" (MDI Child text editing window) with the given Window handle, Text (title), (Left,Top) position and (Width,Depth) size.

**Examples**
The following call will display a small text window near the top left corner of the WIN-PROLOG "main window" client area:

\[ ?- \text{wtcreate( foo, `foo`, 10, 10, 100, 60 ).} \]

yes

**Notes**
This predicate creates a window as part of the WIN-PROLOG "MDI" (Multiple Document Interface): such windows are children of the "MDI Client" window, "(0,1)", and coexist with the "console window" and any program windows.

This is one of a family of WIN-PROLOG window creation predicates, implemented in terms of the lower-level predicate, wcreate/8, which expects an integer style value as its final argument. The wccreate/8 and wdcreate/7 predicates expect Style to be specified as a list of atoms, each denoting a style names, which are converted into the style integer; wtcreate/6 and wucreate/6 have no Style parameter, and simply use a default style integer: all four of these then call wcreate/8 with their computed or default style.
wtext/2

get or set the text of a window

wtext( Window, Text )

+Window <window_handle>
?Text <variable> or <string>

Comments This gets or sets the text of the given Window: if Text is an unbound variable, it returns a string containing the existing text; otherwise, it is used to replace the Window's entire text.

Examples The following call returns the title of the WIN-PROLOG "main window" (window "0"), (this result assumes the system is not currently max-imised):

?- wtext( 0, T ).
T = `WIN-PROLOG`

The next call will replace the entire contents of the "console window" with the string "Hello World":

?- wtext( (1,1), `Hello World` ), nl.
...console is cleared and replaced with...
Hello World
yes

Notes When used with control windows (buttons, edit controls, etc.), wtext/2 gets or sets their text contents; when used with top-level windows (dialogs, text and user windows, as well as the main and console windows), it gets or sets their title. There is a limit of 65536 bytes on the amount of text wtext/2 can handle: see winapi/4 for an example of code that allows unlimited amounts of text to be transferred between windows and files. See wedttxt/2 for information about how to replace selected portions of text in edit controls.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wthumb/3

get or set the thumb position of a scroll bar control

\[ \text{wthumb} \left( \text{Window}, \text{Flag}, \text{Thumb} \right) \]

+Window \quad \langle \text{window_handle} \rangle
+Flag \quad \langle \text{integer} \rangle \text{ in the range } [0..2]
?Thumb \quad \langle \text{integer} \rangle \text{ or } \langle \text{variable} \rangle

Comments

This gets or sets the position of the "Thumb" in a "scrollbar" control, as identified by a combination of the given Window and Flag. If Thumb is an unbound variable, it is bound to the existing thumb position; otherwise it is used to set the position. This predicate can handle windows that are scrollbar controls, as well as other types of windows that contain scrollbars: Flag specifies which case to process:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>process the window as a scrollbar</td>
</tr>
<tr>
<td>1</td>
<td>process the window's horizontal scrollbar</td>
</tr>
<tr>
<td>2</td>
<td>process the window's vertical scrollbar</td>
</tr>
</tbody>
</table>

Examples

The following program uses \text{wdcreate/7} and \text{wccreate/8} to create a small dialog containing a single horizontal scrollbar:

foo :-
    Dstyle = [ws_visible, ws_popup, ws_caption],
    Sstyle = [ws_visible, ws_child, sbs_horz],
    \text{wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle )},
    \text{wccreate( (foo,1), scrollbar, `` , 10, 10, 80, 20, Sstyle )}.

Once compiled, the following call will display the dialog near the top left corner of the screen:

?- foo.
yes

At this point, any attempt to move the scrollbar will fail, because both limits of its range are initially set to zero ("0"); this can be shown in the next call:
?- wrange( (foo,1), 0, L, U, P ).
L = U = P = 0

The next two calls will give the scrollbar a range of [0..100], with a page increment of 10, and set the thumb to "25", a quarter of the way along the scrollbar:

?- wrange( (foo,1), 0, 0, 100, 10 ).
yes

?- wthumb( (foo,1), 0, 25 ).
yes

Finally, after the scrollbar has been moved by the user, its thumb position can be queried:

?- wthumb( (foo,1), 0, T ).
T = 81

Notes

Two predicates, \texttt{wrange/5} and \texttt{wthumb/3}, provide support for "scrollbar" controls. Unlike other types of control (buttons, listboxes, etc), scrollbars can either occur as stand-alone windows (see the above example) or as features built in to other windows. For this reason, these two predicates are named as general window handling predicates, rather than as control-related ones ("wsclrng/4" and "wscithm/3" or whatever).

The \texttt{Lower} and \texttt{Upper} limits of scrollbars are specified as 16-bit signed integers (-32768..32767); in practice, an even smaller range might be supported: it is recommended to keep the limits approximately in the range -16384..16383, and to use arithmetic to scale output values if required (see \texttt{is/2}).

This is one of a family of \texttt{WIN-PROLOG} window handling predicates, implemented as a "thin layer" over the all-powerful \texttt{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \texttt{winapi/4} and its helper predicate, \texttt{wintxt/4}, can be used to access virtually any function in the \texttt{Windows API} as well as those implemented in third-party DLLs.
**wtrkpos/2**

*get or set the position of a trackbar control*

```prolog
wtrkpos( Window, Position )
```

+Window <window_handle>
?Position <integer> or <variable>

**Comments**

This gets or sets the *Position* of a "trackbar" (track bar) control *Window*. If *Position* is an unbound variable, it is bound to the existing position; otherwise it is used to set the position.

**Examples**

The following program uses `wdcreate/7` and `wccreate/8` to create a small dialog containing a single horizontal trackbar:

```prolog
foo :-
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Tstyle = [ws_visible,ws_child,tbs_horz],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), trackbar, ``, 10, 10, 80, 20, Tstyle ).
```

Once compiled, the following call will display the dialog near the top left corner of the screen:

```prolog
?- foo.
yes
```

When first created, a trackbar has a default range of [0..100]; this can be shown in the next call:

```prolog
?- wtrkrng( (foo,1), L, U ).
L = 0 ,
U = 100
```

The next two calls will change the trackbar range to [0..1000], and set the position to "250", a quarter of the way along the trackbar:

```prolog
?- wtrkrng( (foo,1), 0, 1000 ).
yes
```
?- wtrkpos( (foo,1), 250 ).
yes

Finally, after the trackbar has been moved by the user, its position can be queried:

?- wtrkpos( (foo,1), T ).
T = 810

Notes

Two predicates, wtrkrg/3 and wtrkpos/2, provide support for "trackbar" controls, which are based on the Windows "common control" class, "msctls_trackbar32", commonly known as "Track Bars": see wcreate/8 for more information.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wtrkrng/3

get or set the range of a trackbar control

\[\text{wtrkrng( Window, Lower, Upper )}\]

+Window <window_handle>
?Lower <integer> or <variable>
?Upper <integer> or <variable>

Comments
This gets or sets the Lower and Upper limits of the range of a "trackbar" (track bar) control Window. If either or both of Lower and Upper are unbound variables, they are bound to the existing lower and/or upper limits accordingly; otherwise they are used to set these limits.

Examples
The following program uses wdcreate/7 and wccreate/8 to create a small dialog containing a single horizontal trackbar:

\[\text{foo :-}\]
\[\text{Dstyle} = [\text{ws_visible, ws_popup, ws_caption}],\]
\[\text{Tstyle} = [\text{ws_visible, ws_child, tbs_horz}],\]
\[\text{wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),}\]
\[\text{wccreate( (foo,1), trackbar, ``, 10, 10, 80, 20, Tstyle ).}\]

Once compiled, the following call will display the dialog near the top left corner of the screen:

\[?- \text{foo}.\]
\[\text{yes}\]

When first created, a trackbar has a default range of [0..100]; this can be shown in the next call:

\[?- \text{wtrkrng( (foo,1), L, U ).}\]
\[L = 0 ,\]
\[U = 100\]

The next two calls will change the trackbar range to [0..1000], and set the position to "250", a quarter of the way along the trackbar:

\[?- \text{wtrkrng( (foo,1), 0, 1000 ).}\]
\[\text{yes}\]
Finally, after the trackbar has been moved by the user, its position can be queried:

```
?- wtrkpos( (foo,1), 250 ).
yes

T = 810
```

### Notes

Two predicates, `wtrkrng/3` and `wtrkpos/2`, provide support for "trackbar" controls, which are based on the Windows "common control" class, "msctls_trackbar32", commonly known as "Track Bars": see `wcreate/8` for more information.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
**wtvwadd/5**

*add an item to a treeview control*

```
wtvwadd( Window, ID, String, Item, New )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>&lt;window_handle&gt;</td>
</tr>
<tr>
<td>ID</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>String</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>Item</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>New</td>
<td>&lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**

This adds a `String`, together with a user-defined numerical `Item`, to a "treeview" control `Window` at the specified existing Node `ID`, returning the Node ID of the `New` item. The root node of the treeview control has an ID of zero ("0").

**Examples**

Consider a dialog called "foo" that includes a treeview control with an ID of 9, and an existing node with an ID of "24680"; the following call will append the string, "cheetah", with a user-defined item, "123", at end of the given node's subtree, returning the ID new of the new node that has just been created:

```
?- wtvwadd( (foo,9 ), 24680, `cheetah`, 123, N ).
N = 97531
```

**Notes**

Whenever accessing a treeview control, the position within the tree is identified by a "Node ID", which forms the second argument of each of the `wtvw???/n` predicates. From any given node, it is possible to navigate to the siblings, parents or first children of an given node with the `wtvwlnk/4` predicate; the root node is, by definition, always numbered zero ("0").

When adding new nodes to a treeview control with `wtvwadd/5`, the fifth (variable) argument returns the Node ID of the newly added node. This can be used as the second argument to future calls to `wtvwadd/5`, in order to add a subtree to the new node.

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the **Windows API** as well as those implemented in third-party DLLs.
### wtvwbtn/3

**get or set the selection state of a button in a treeview control**

\[
\text{wtvwbtn}( \text{Window, ID, State} )
\]

- **Window**: `<window_handle>`
- **ID**: `<integer>`
- **State**: `<integer>` or `<variable>`

**Comments**

This gets or sets the selection State of a button in a "treeview" control `Window` at the specified existing Node `ID`. Where `State` is an unbound variable, it returns the current button state; where it is an integer, it sets the button's state.

**Examples**

Consider a dialog called "foo" that includes a treeview control with an ID of 9, and an existing node with an ID of "24680"; the following call will set the button to display its second state image, which by default, is a tick:

\[
?- \text{wtvwbtn}( (\text{foo,9 }), 24680, 2 ).
\]

**Notes**

Whenever accessing a treeview control, the position within the tree is identified by a "Node ID", which forms the second argument of each of the `wtvw???/n` predicates. From any given node, it is possible to navigate to the siblings, parents or first children of a given node with the `wtvwlnk/4` predicate; the root node is, by definition, always numbered zero ("0").

If a treeview button is set to a state of zero ("0"), the button is hidden and the user cannot click on it; if set to a positive integer, it will display the appropriate image for its selection state. By default, buttons have just two states, "1" and "2", but if an image list is applied to the treeview control's buttons, there can be any number of states, matching the number of images in the list minus one. For example, if an image list with 8 images is applied to the treeview, and the state of any given button is not zero, then the user can cycle through 7 states, each showing the next image in the list.

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the *Windows API* as well as those implemented in third-party DLLs.
wtvwdel/2

delete an item from a treeview control

\[
\text{wtvwdel( Window, ID )}
\]

+Window \text{<window_handle>}
+ID \text{<integer>}

Comments

This deletes the node and all its subtrees in a "treeview" control \textit{Window} at the specified existing Node \textit{ID}; if the \textit{ID} is specified as zero ("0"), the entire tree is deleted.

Examples

Consider a dialog called "foo" that includes a treeview control with an ID of 9, and an existing node with an ID of "24680"; the following call will delete that node and all its subtrees:

\[
?- \text{wtvwdel( (foo,9 ), 24680 ).}
\]

yes

Notes

Whenever accessing a treeview control, the position within the tree is identified by a "Node ID", which forms the second argument of each of the wtvw???/n predicates. From any given node, it is possible to navigate to the siblings, parents or first children of an given node with the wtvwlnk/4 predicate; the root node is, by definition, always numbered zero ("0").

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful \textit{winapi/4} to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, \textit{winapi/4} and its helper predicate, \textit{wintxt/4}, can be used to access virtually any function in the \textit{Windows API} as well as those implemented in third-party DLLs.
### wtvwget/4

**get an item from a treeview control**

```prolog
wtvwget( Window, ID, String, Item )
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>&lt;window_handle&gt;</td>
</tr>
<tr>
<td>ID</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>String</td>
<td>&lt;variable&gt;</td>
</tr>
<tr>
<td>Item</td>
<td>&lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Comments**  
This gets an item from a "treeview" control `Window` at the specified `ID`, and binds its text to `String` and user-defined item value to `Item`. The first entry in this type of control is numbered zero ("0").

**Examples**  
Consider a dialog called "foo" that includes a treeview control with an ID of 9, and an existing node with an ID of "24680"; the following call will pick up the entry (say, "badger", with an its item value):

```prolog
?- wtvwget( (foo,9 ), 24680, S, I ).
```

```
S = `badger` ,
I = 273
```

**Notes**  
Whenever accessing a treeview control, the position within the tree is identified by a "Node ID", which forms the second argument of each of the `wtvw???/n` predicates. From any given node, it is possible to navigate to the siblings, parents or first children of a given node with the `wtvwlink/4` predicate; the root node is, by definition, always numbered zero ("0").

This is one of a family of **WIN-PROLOG** window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the **Windows API** as well as those implemented in third-party DLLs.
wtvwlnk/4

link from one item to another in a treeview control

wtvwlnk( Window, ID, Flag, Target )

+Window        <window_handle>
+ID            <integer>
+Flag          <integer> in the range [-1..5]
?Target        <variable>

Comments This links from a "treeview" control Window at the specified existing Node ID, to a Target node, its Node ID. The root node of the treeview control has an ID of zero ("0"). The following table lists the values and meanings of Flag:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Retrieves the topmost node in a TreeView control</td>
</tr>
<tr>
<td>1</td>
<td>Retrieves the next sibling node</td>
</tr>
<tr>
<td>2</td>
<td>Retrieves the previous sibling node</td>
</tr>
<tr>
<td>3</td>
<td>Retrieves the parent node</td>
</tr>
<tr>
<td>4</td>
<td>Retrieves the first child node</td>
</tr>
<tr>
<td>5</td>
<td>Retrieves the first visible node</td>
</tr>
<tr>
<td>6</td>
<td>Retrieves the next visible node</td>
</tr>
<tr>
<td>7</td>
<td>Retrieves the previous visible node</td>
</tr>
<tr>
<td>8</td>
<td>Retrieves the target node of a drag and drop</td>
</tr>
<tr>
<td>9</td>
<td>Retrieves the currently selected node</td>
</tr>
<tr>
<td>10</td>
<td>Retrieves the last visible node</td>
</tr>
</tbody>
</table>

Examples Consider a dialog called "foo" that includes a treeview control with an ID of 9, and an existing node with an ID of "24680"; the following call will link to its parent node:

?- wtvwlnk( (foo,9 ), 24680, 3, N ).
N = 13579
Whenever accessing a treeview control, the position within the tree is identified by a "Node ID", which forms the second argument of each of the `wtvw???/n` predicates. From any given node, it is possible to navigate to the siblings, parents or first children of an given node with the `wtvwinl/4` predicate; the root node is, by definition, always numbered zero ("0").

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful `winapi/4` to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, `winapi/4` and its helper predicate, `wintxt/4`, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wtvwsel/2

get or set the selection in a treeview control

wtvwsel( Window, ID )

+Window <window_handle>
?ID <integer>

Comments
This gets or sets the selection ID in a "treeview" control Window at the specified ID.

Examples
Consider a dialog called "foo" that includes a treeview control with an ID of 9, and an existing node with an ID of "24680"; the following call will select this node:

?- wtvwsel( (foo,9), 24680 ).
yes

To find out which node is selected, a variable is used as the ID; the following call will return this selection:

?- wtvwsel( (foo,9), P ).
P = 24680

Notes
Whenever accessing a treeview control, the position within the tree is identified by a "Node ID", which forms the second argument of each of the wtvw???/n predicates. From any given node, it is possible to navigate to the siblings, parents or first children of an given node with the wtvwlink/4 predicate; the root node is, by definition, always numbered zero ("0").

This is one of a family of WIN-PROLOG window handling predicates, implemented as a "thin layer" over the all-powerful winapi/4 to simplify common Windows programming tasks. While most applications can be fully satisfied with these simple predicates, there are always likely to be certain combinations of functions which cannot be performed easily. On these occasions, winapi/4 and its helper predicate, wintxt/4, can be used to access virtually any function in the Windows API as well as those implemented in third-party DLLs.
wucreate/6

create a user window

wucreate( Window, Text, Left, Top, Width, Depth )

+Window <window_handle>
+Text <string>
+Left <integer>
+Top <integer>
+Width <integer>
+Depth <integer>

Comments
This creates a "user window" (user-definable MDI Child window) with the given Window handle, Text (title), (Left,Top) position and (Width,Depth) size.

Examples
The following program uses wucreate/6 and wccreate/8 to create a user window containing a single "default push" button:

foo :-
    Bstyle = [ws_visible, ws_child, bs_defpushbutton],
    wucreate( foo, `foo`, 10, 10, 100, 60 ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ).

Once compiled, the following call will display the window near the top left corner of the WIN-PROLOG "main window" client area (clicking the button will not hide a user window):

?- foo.
    yes

Notes
This predicate creates a window as part of the WIN-PROLOG "MDI" (Multiple Document Interface): such windows are children of the "MDI Client" window, "(0,1)", and coexist with the "console window" and any program windows.

This is one of a family of WIN-PROLOG window creation predicates, implemented in terms of the lower-level predicate, wcreate/8, which expects an integer style value as its final argument. The wccreate/8 and wdcreate/7 predicates expect Style to be specified as a list of atoms, each denoting a style names, which are converted into the style integer; wtcreate/6 and wucreate/6 have no Style parameter, and simply use a default style integer: all four of these then call wcreate/8 with their computed or default style.
return or adjust the total amounts of space allocated

\[ \text{xinit}(B, L, R, H, T, P, S, I, O) \]

\[ ?B \quad \text{<integer> or <variable>} \]
\[ ?L \quad \text{<integer> or <variable>} \]
\[ ?R \quad \text{<integer> or <variable>} \]
\[ ?H \quad \text{<integer> or <variable>} \]
\[ ?T \quad \text{<integer> or <variable>} \]
\[ ?P \quad \text{<integer> or <variable>} \]
\[ ?S \quad \text{<integer> or <variable>} \]
\[ ?I \quad \text{<integer> or <variable>} \]
\[ ?O \quad \text{<integer> or <variable>} \]

Comments
If all arguments are variables, this returns the total amount of space, in kilobytes, requested for each of these memory areas: backtrack stack \((B)\), local stack \((L)\), reset stack \((R)\), term heap \((H)\), text heap \((T)\), program heap \((P)\), system stack \((S)\), string input buffer \((I)\) and string output buffer \((O)\); if all arguments are integers, the respective memory areas are adjusted to the desired sizes. When setting memory sizes, each value is treated as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>reset to default setting</td>
</tr>
<tr>
<td>0</td>
<td>leave at current setting</td>
</tr>
<tr>
<td>&lt;other&gt;</td>
<td>set to given number of kilobytes</td>
</tr>
</tbody>
</table>

Examples
Suppose that \text{WIN-PROLOG} has been run with the command line switch, "/T1234", which sets the text heap to 1234kb (See Appendix B for more information about \text{WIN-PROLOG}'s command line switches); the following call returns the total amount of text and program space originally allocated:

\[ ?- \text{xinit}(_, _, _, _, T, P, _, _, _). \]

\[ T = 1234, \]
\[ P = 8192 \]

The following call will simultaneously reset text space to its default value, while increasing program space to 9999kb:
?- xinit( 0, 0, 0, 0, -1, 9999, 0, 0, 0 ).
   yes

The new settings can quickly be confirmed by repeating the original call and noting its new output:

?- xinit( _, _, _, _, T, P, _, _, _ ).
   T = 2048 ,
   P = 9999

Notes

The total/9 predicate returns the total size of each of the nine configurable memory areas, but does not indicate how much of this space is used and how much is free. Another predicate, free/9, returns the amounts of memory currently free in the same nine areas: by comparing the results of calling free/9 and total/9, it is possible to compute additional statistics, such as the amount of space being used, or the percentage of space remaining free. See the entries for free/9 and gc/1 for additional information regarding garbage collection.

The present predicate allows changes to be made to the memory allocation at runtime: in order perform this memory reallocation, xinit/9 performs a full "compacting" garbage collection on all stacks and heaps, before writing their contents together other data buffers to a temporary file in the current working directory. Memory is then reallocated according to the user's supplied parameters, before the temporary file is reloaded and then deleted.

Here is an illustrated transcript to show xinit/9 in action; note that parameters are specified in kilobytes, with the two special cases of zero ("0"), which causes the given data area to remain unchanged, and minus one ("-1"), which resets a memory area to its default size, with changes shown in bold:

| ?- ver( 0 ).                          
| BDS WIN-PROLOG 7.000 X86 S/N 0020596320 28 Feb 2019
| Copyright 1989-2019 Brian D Steel (www.solanum.org)
| Licensed To: LPA Development and Documentation Team
| B=96 L=64 R=96 H=256 T=2048 P=8192 S=64 I=256 O=256
| yes

| ?- xinit( -1, 0, 0, 512, 0, 0, 0, 0, 0 ), ver( 0 ).
| BDS WIN-PROLOG 7.000 X86 S/N 0020596320 28 Feb 2019
| Copyright 1989-2019 Brian D Steel (www.solanum.org)
| Licensed To: LPA Development and Documentation Team
| B=64 L=64 R=96 H=512 T=2048 P=8192 S=64 I=256 O=256
| yes
**xload/1**

restore a memory image and continue execution

```
xload( File )
+File <atom>
```

**Comments**

This loads a memory image from the given *File*, and continues execution of the saved computation from the first available choicepoint.

**Examples**

The following call saves a memory image in a file called "foo", before writing the word, "Hello":

```
?- xsave( foo ) -> write( `Hello-M-J` ) ; write( `World-M-J` ).
Hello
yes
```

The following call loads the memory image from the file, "foo", and continues execution at the next available choice point:

```
?- xload( foo ).
World
yes
```

**Notes**

The *xsave/1* and *xload/1* predicates are used, respectively, to save and restore an image of the entire state of WIN-PROLOG at any point in program execution. This memory image includes not only all programs, dynamic data, atom dictionaries, etc., but also the heaps and stacks that are active at the time of the call to *xsave/1*: execution continues from the point of this call, but at any time in the future, even in a completely different WIN-PROLOG session, this file can be reloaded with *xload/1* to restore the exact state of execution that existed when *xsave/1* was called, with one difference: the latter predicate is deemed to have "failed". The result of this intentional failure, is to transfer control to the next available choice point, as seen in the above example, where "Hello" is written after the (successful) call to *xsave/1*, but "World" is written once this memory image is brought back in by the call to *xload/1*:

```
{ xsave( foo ) -> <continue execution here when xsave/1 is complete>
 ; <continue execution here when xload/1 is complete>
}
```

Because image files are designed to be reloaded at any arbitrary point in the future, certain resources that may no longer be physically
present, are closed down automatically before saving or loading. This includes all open files, and all open windows, as well as other re-
resources such as sockets, MIDI devices, etc. While this potentially limits the utility of xsave/1 and xload/1 to WIN-PROLOG sessions, the
principle purpose of these predicates was to support applications such as Weblog, where a genuine, backtracking Prolog program can
be run as a sequence of segments, potentially across multiple servers, interspersed by the submission of web pages to the user, and
continuing from where it left off, on receipt of a subsequent from submission.

      weblog :-
        <do some stuff>
        submit( 'foo.htm', [<some data>] ),
        <do more stuff>
        submit( 'bar.htm', [<more data>] ),
        ...

Each time a Weblog program calls "submit/2" to combine a web page template with whatever data has been computed, it saves a memory
image, and then closes down. When the user responds to the web page, maybe just by clicking a button saying "continue", or perhaps
first filling in a form with more information, the CGI server signals an instance of Weblog to load the memory image, and process the
incoming web form.

The beauty of this model, is that the Weblog programmer simply writes a Prolog program, using "submit/2" to output web pages where a
desktop application would perhaps simply use write/1: full backtracking is supported throughout the Weblog application, including between
submitted web pages, thanks to xsave/1 and xload/1.

The mechanism behind xsave/1 and xload/1 is also used to implement the xinit/9 predicate, which works by dumping all programs, atoms,
heaps and stacks into a memory image file, before clearing out and re-allocating memory as requested, and then reloading the image
file into the newly set-up memory.
xsave/1

save a memory image and continue execution

\[ \text{xsave}( \text{File} ) \]

+File <atom>

Comments
This saves a memory image from the given File, and continues execution normally from that point.

Examples
The following call saves a memory image in a file called "foo", before writing the word, "Hello":

\[ ?- \text{xsave}( \text{foo} ) -> \text{write}( \text{"Hello-M-J"} ) ; \text{write}( \text{"World-M-J"} ). \]

Hello
yes

The following call loads the memory image from the file, "foo", and continues execution at the next available choice point:

\[ ?- \text{xload}( \text{foo} ). \]

World
yes

Notes
The xsave/1 and xload/1 predicates are used, respectively, to save and restore an image of the entire state of WIN-PROLOG at any point in program execution. This memory image includes not only all programs, dynamic data, atom dictionaries, etc., but also the heaps and stacks that are active at the time of the call to xsave/1: execution continues from the point of this call, but at any time in the future, even in a completely different WIN-PROLOG session, this file can be reloaded with xload/1 to restore the exact state of execution that existed when xsave/1 was called, with one difference: the latter predicate is deemed to have "failed". The result of this intentional failure, is to transfer control to the next available choice point, as seen in the above example, where "Hello" is written after the (successful) call to xsave/1, but "World" is written once this memory image is brought back in by the call to xload/1:

\[
\{ \text{xsave}( \text{foo} ) \\
-> \langle \text{continue execution here when xsave/1 is complete} \rangle \\
; \langle \text{continue execution here when xload/1 is complete} \rangle 
\}
\]

Because image files are designed to be reloaded at any arbitrary point in the future, certain resources that may no longer be physically
present, are closed down automatically before saving or loading. This includes all open files, and all open windows, as well as other re-
sources such as sockets, MIDI devices, etc. While this potentially limits the utility of xsave/1 and xload/1 to WIN-PROLOG sessions, the
principle purpose of these predicates was to support applications such as Weblog, where a genuine, backtracking Prolog program can
be run as a sequence of segments, potentially across multiple servers, interspersed by the submission of web pages to the user, and
continuing from where it left off, on receipt of a subsequent from submission.

```
weblog :-
    <do some stuff>
    submit( 'foo.htm', [<some data>] ),
    <do more stuff>
    submit( 'bar.htm', [<more data>] ),
   ...
```

Each time a Weblog program calls "submit/2" to combine a web page template with whatever data has been computed, it saves a memory
image, and then closes down. When the user responds to the web page, maybe just by clicking a button saying "continue", or perhaps
first filling in a form with more information, the CGI server signals an instance of Weblog to load the memory image, and process the
incoming web form.

The beauty of this model, is that the Weblog programmer simply writes a Prolog program, using "submit/2" to output web pages where a
desktop application would perhaps simply use write/1: full backtracking is supported throughout the Weblog application, including between
submitted web pages, thanks to xsave/1 and xload/1.

The mechanism behind xsave/1 and xload/1 is also used to implement the xinit/9 predicate, which works by dumping all programs, atoms,
heaps and stacks into a memory image file, before clearing out and re-allocating memory as requested, and then reloading the image
file into the newly set-up memory.
Appendix A - Term Comparison

A number of WIN-PROLOG predicates perform "less than" and "greater than" comparisons between terms: these include cmp/3, compare/3, sort/2, sort/3 and a collection of special "@" comparison operators. All WIN-PROLOG terms occupy a space in the so-called "Standard Ordering of Terms":

- If both terms are of different types (except for numbers - see below), comparison is defined by the relationship between these types.
- If both terms are of the same atomic type, or if one is an integer and the other is a float, comparison is based on their value.
- If both terms are of the same compound type, comparison is based recursively first on their respective heads, and then on their tails.

The following table lists the ten WIN-PROLOG data types, together with their type integers (see type/2) in ascending sort order; note that although they are different data types, integers and floats are considered to be of a single type during term comparisons:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;variable&gt;</td>
<td>0</td>
</tr>
<tr>
<td>&lt;integer&gt; and &lt;float&gt;</td>
<td>1 and 2</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>3</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>4</td>
</tr>
<tr>
<td>&lt;empty list&gt;</td>
<td>5</td>
</tr>
<tr>
<td>&lt;list&gt;</td>
<td>6</td>
</tr>
<tr>
<td>&lt;tuple&gt;</td>
<td>7</td>
</tr>
<tr>
<td>&lt;conjunction&gt;</td>
<td>8</td>
</tr>
<tr>
<td>&lt;disjunction&gt;</td>
<td>9</td>
</tr>
</tbody>
</table>

The following table explains how any two terms of a given atomic data type are compared (again, please note that integers and floats are considered to be of a single type during term comparisons):

<table>
<thead>
<tr>
<th>Type</th>
<th>Compared on</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;variable&gt;</td>
<td>heap address (visible as the &quot;nnn&quot; number in a variable when it is written in the form, &quot;,_nnn&quot;)</td>
</tr>
<tr>
<td>&lt;integer&gt; and &lt;float&gt;</td>
<td>numerical value</td>
</tr>
<tr>
<td>&lt;atom&gt;</td>
<td>left-to-right character code, and if this fails, on length</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>left-to-right character code, and if this fails, on length</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>&lt;empty list&gt;</td>
<td>all empty lists have the same &quot;value&quot;</td>
</tr>
</tbody>
</table>

For the four compound data types, an attempt is made to compare their head elements recursively: if a pair of subterms is found to be different, its ordering determines that of the entire term. If the heads are found to be identical, the tails are compared recursively: again, if a pair of subterms is found to be different, its ordering determines that of the entire term. Comparison of compound terms continues either until the first difference is found, or until both terms are exhausted: in the latter case, the terms are deemed to be equal.
Appendix B - Initialisation and Switches

The WIN-PROLOG system can be configured in two main ways, namely through an "initialisation file" (.INI file) and by a number of "command line switches". In fact, the initialisation file can contain default settings for these switches, which can then be overridden as needed by the user.

Command Line Syntax

The WIN-PROLOG command line consists of three optional sections. The first is an "overlay override", which uses the <escargot> character ("@") to specify an overlay (.OVL file) to use in place of the standard one, "pro386w.ovl". This is followed by zero or more switches, each of which consists of a <slash> character (/) followed by one of the letters A..Z or a..z, which in turn may be followed by an unsigned integer: switch letters are not case-sensitive (/A" and "/a" are synonymous). There must be no spaces within a switch, and there must be at least one space between neighbouring command line entries.

The final section of the command line contains zero or more Prolog goals which will be executed once the development system has been initialised. For example, the following command initialises WIN-PROLOG to use an overlay called "FOO.OVL", with the "A" switch set to "0" and the "B" (backtrack stack) switch set to 128 and the term, "write( `Hello World` )." offered as a goal:

```
pro386w @foo /A /B128 write( `Hello World` ).
```

Merging .INI and Typed Command Lines

The "PRO386W.INI" file can contain a command line with all the features described above (see below for more information about the .INI file), and this is merged with any actual command line elements as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>overlay</td>
<td>an overlay override on the command line takes precedence over one in the .INI file</td>
</tr>
<tr>
<td>switches</td>
<td>switches in the .INI file are merged with those on the command line, the latter taking precedence</td>
</tr>
<tr>
<td>goals</td>
<td>goals in the .INI file are executed first, followed by any on the command line</td>
</tr>
</tbody>
</table>

The effect of these rules is to allow default settings to be stored in the .INI file, without losing the ability to change individual settings in any given session. By allowing .INI file goals to be executed in addition to command line goals, it is easy to load libraries or current projects automatically.
Memory Settings

Several switches are used to allocate the amount of memory used in each of the nine configurable WIN-PROLOG memory areas: each one specifies a number of kilobytes of memory to use (see Appendix H for a detailed discussion of memory management). If any given switch is not specified, its value will be set to "-1"; if it is specified without a numerical argument, or with an argument of zero, its value will be set to "0": in any of these cases, it is treated as the default setting. The memory switches are:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Meaning</th>
<th>Default Size (kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/B</td>
<td>backtrack stack</td>
<td>64</td>
</tr>
<tr>
<td>/L</td>
<td>local stack</td>
<td>64</td>
</tr>
<tr>
<td>/R</td>
<td>reset stack</td>
<td>64</td>
</tr>
<tr>
<td>/H</td>
<td>term heap</td>
<td>256</td>
</tr>
<tr>
<td>/T</td>
<td>text heap</td>
<td>2048</td>
</tr>
<tr>
<td>/P</td>
<td>program heap</td>
<td>8192</td>
</tr>
<tr>
<td>/S</td>
<td>system stack</td>
<td>64</td>
</tr>
<tr>
<td>/I</td>
<td>string input buffer</td>
<td>256</td>
</tr>
<tr>
<td>/O</td>
<td>string output buffer</td>
<td>256</td>
</tr>
</tbody>
</table>

Miscellaneous Settings

A handful of switches are used to set certain other aspects of WIN-PROLOG behaviour. If any given switch is not specified, its value will be "-1"; if it is specified without a numerical argument, or with an argument of zero, its value will be "0": in any of these cases, it is treated as the default setting. These switches are:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Meaning</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>/D</td>
<td>dynamic code calls</td>
<td>0 = direct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = logical</td>
</tr>
<tr>
<td>/V</td>
<td>banner flag</td>
<td>0 = show banner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = hide banner</td>
</tr>
</tbody>
</table>

Initialisation File Entries

The "PRO386W.INI" file contains two main classes of sections. The first comprises a single heading, and is processed by the WIN-PROLOG kernel very early in the boot process: this "[pro386w]" section can contain one of two main entries. The first entry, "profile=" , can be used to redirect all further processing to another .INI file, and allows network administrators to position such files on users' hard disks, private directories, current working directories, centrally,
or whatever. If present, it renders the rest of the "PRO386W.INI" file redundant, giving complete precedence to its named file. Note that this entry is only recognised in "PRO386W.INI" itself: this limits redirection to a single step, and therefore avoids "wild goose chases" or even endless looping.

The second entry, "command="", contains any default command line, as discussed above. Consider the following .INI file, shown here in oblique:

```
[pro386w]
command=/h512 /t1024 /p4096 consult( library(trace) ).
```

A command or Windows shortcut, shown here in bold, such as:

```
pro386w /h /p16384 /s128 write( hello ), nl.
```

would result in the following effective merged command line, shown here in mixed styles, including oblique and bold to identify the source of its component parts:

```
pro386w /t1024 /h /p16384 /s128 consult( library(trace) ). write( hello ), nl.
```

This will start WIN-PROLOG, returning the "term heap" allocation to its default, rather than the 512kb indicated in the .INI file, further increasing the "program heap" allocation to 16384kb, setting the "system stack" to 128kb, and using the .INI file setting of 1024kb for the "text heap". It will then execute the goal, "consult(library(trace)).", before in turn executing "write( hello ), nl.".

Some toolkits use additional entries in the "[pro386w]" section, but these are ignored by WIN-PROLOG itself, and so are beyond the scope of this manual.

The second class of sections includes a variety of headings, and lists various persistent settings that are saved whenever the "save_settings" Prolog flag has been set to "on" (see prolog_flag/3). Note that even if this flag is set to "off", any settings in the .INI file will still be loaded at startup: the only way to restore defaults is to delete this file (or, at the very least, its offending entries) prior to starting WIN-PROLOG. Here are some typical entries:

```
[prolog_flag]
save_settings=on
context=supervisor
... etc

>window]
console=0 0 814 452 3
main=0 0 1020 696 1
```
Appendix C - Window Style Names

When creating a window, it is usually necessary to specify its "style": this determines the window's appearance and behavioural properties. Some styles pertain to most windows: visibility, for example, or whether the window is a "child" or "popup" window. Other styles relate to specific window classes: a button control can appear and behave as a checkbox, radio button or simple pushbutton. Each style is associated with one or more bits in a 32-bit integer: the required set of styles must be combined using the "inclusive or" function, before being passed to \texttt{wcreate/8} to create a window. For example, the following program creates a "dialog" window with the combined styles, "ws_visible", "ws_popup" and "ws_caption", containing a "button" control with the combined styles, "ws_visible", "ws_child" and "bs_defpushbutton".

\begin{verbatim}
foo : -
    wcreate( foo, dialog, `foo`, 10, 10, 100, 60, 16'90C00000 ),
    wcreate( (foo,1), button, `bar`, 10, 10, 80, 20, 16'50000001 ).
\end{verbatim}

To make the specification of window styles easier, \textsc{WIN-PROLOG} includes two additional window creation predicates, \texttt{wdcreate/7} and \texttt{wccreate/8}, in which the final "style" argument can be presented as a list of style names, rather than as an integer; the following program produces exactly the same dialog as before, but uses style lists rather than precomputed integer styles:

\begin{verbatim}
foo : -
    Dstyle = [ws_visible,ws_popup,ws_caption],
    Bstyle = [ws_visible,ws_child,bs_defpushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, 16'50000001 ).
\end{verbatim}

Two further predicates, \texttt{wtcreate/6} and \texttt{wucreate/6} create "text" and "user" ("MDI Child") windows respectively: these window types have predefined styles, and so these predicates omit the style parameter altogether.

Please note that not all styles types can be combined: a button, for example, cannot simultaneously be a checkbox ("bs_checkbox"), radio button ("bs_radiobutton") and default pushbutton ("bs_defpushbutton").

The full set of style names that can be used in \texttt{wdcreate/7} and \texttt{wccreate/8}, together with the types of window to which they may pertain, are listed in the following sections. Their names are generally fairly self-explanatory, but a detailed explanation of each is beyond the scope of this manual: the reader is referred to Microsoft documentation for further discussion.
## Generic Styles

These are styles that can be applied to more than one class of window:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws_border</td>
<td>16'00800000</td>
</tr>
<tr>
<td>ws_caption</td>
<td>16'00c00000</td>
</tr>
<tr>
<td>ws_child</td>
<td>16'40000000</td>
</tr>
<tr>
<td>ws_clipchildren</td>
<td>16'02000000</td>
</tr>
<tr>
<td>ws_clipsiblings</td>
<td>16'04000000</td>
</tr>
<tr>
<td>ws_disabled</td>
<td>16'08000000</td>
</tr>
<tr>
<td>ws_dlgframe</td>
<td>16'00400000</td>
</tr>
<tr>
<td>ws_group</td>
<td>16'00020000</td>
</tr>
<tr>
<td>ws_hscroll</td>
<td>16'00100000</td>
</tr>
<tr>
<td>ws_maximize</td>
<td>16'01000000</td>
</tr>
<tr>
<td>ws_maximizebox</td>
<td>16'00010000</td>
</tr>
<tr>
<td>ws_minimize</td>
<td>16'20000000</td>
</tr>
<tr>
<td>ws_minimizebox</td>
<td>16'00020000</td>
</tr>
<tr>
<td>ws_overlapped</td>
<td>16'00000000</td>
</tr>
<tr>
<td>ws_popup</td>
<td>16'80000000</td>
</tr>
<tr>
<td>ws_sysmenu</td>
<td>16'00080000</td>
</tr>
<tr>
<td>ws_tabstop</td>
<td>16'00010000</td>
</tr>
<tr>
<td>ws_thickframe</td>
<td>16'00040000</td>
</tr>
<tr>
<td>ws_visible</td>
<td>16'10000000</td>
</tr>
<tr>
<td>ws_vscroll</td>
<td>16'00200000</td>
</tr>
</tbody>
</table>

## Button Styles

These are styles that can be applied to button controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
</table>

---

WIN-PROLOG 7.0 - Technical Reference
### Edit Styles

These are styles that can be applied to edit controls:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs_3state</td>
<td>16'00000005</td>
</tr>
<tr>
<td>bs_auto3state</td>
<td>16'00000006</td>
</tr>
<tr>
<td>bs_autocheckbox</td>
<td>16'00000003</td>
</tr>
<tr>
<td>bs_autoradiobutton</td>
<td>16'00000009</td>
</tr>
<tr>
<td>bs_bitmap</td>
<td>16'00000080</td>
</tr>
<tr>
<td>bs_bottom</td>
<td>16'00000800</td>
</tr>
<tr>
<td>bs_center</td>
<td>16'00000300</td>
</tr>
<tr>
<td>bs_checkbox</td>
<td>16'00000002</td>
</tr>
<tr>
<td>bs_defpushbutton</td>
<td>16'00000001</td>
</tr>
<tr>
<td>bs_flat</td>
<td>16'00000800</td>
</tr>
<tr>
<td>bs_groupbox</td>
<td>16'00000007</td>
</tr>
<tr>
<td>bs_icon</td>
<td>16'00000040</td>
</tr>
<tr>
<td>bs_left</td>
<td>16'00000100</td>
</tr>
<tr>
<td>bs_lefttext</td>
<td>16'00000020</td>
</tr>
<tr>
<td>bs_multiline</td>
<td>16'00002000</td>
</tr>
<tr>
<td>bs_notify</td>
<td>16'00004000</td>
</tr>
<tr>
<td>bs_ownerdraw</td>
<td>16'0000000b</td>
</tr>
<tr>
<td>bs_pushbutton</td>
<td>16'00000000</td>
</tr>
<tr>
<td>bs_pushlike</td>
<td>16'00001000</td>
</tr>
<tr>
<td>bs_radiobutton</td>
<td>16'00000004</td>
</tr>
<tr>
<td>bs_right</td>
<td>16'00000200</td>
</tr>
<tr>
<td>bs_text</td>
<td>16'00000000</td>
</tr>
<tr>
<td>bs_top</td>
<td>16'00000400</td>
</tr>
<tr>
<td>bs_userbutton</td>
<td>16'00000008</td>
</tr>
<tr>
<td>bs_vcenter</td>
<td>16'00000c00</td>
</tr>
<tr>
<td>Style</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>es_autohscroll</td>
<td>16'000000080</td>
</tr>
<tr>
<td>es_autovscroll</td>
<td>16'000000040</td>
</tr>
<tr>
<td>es_center</td>
<td>16'00000001</td>
</tr>
<tr>
<td>es_left</td>
<td>16'00000000</td>
</tr>
<tr>
<td>es_lowercase</td>
<td>16'00000010</td>
</tr>
<tr>
<td>es_multiline</td>
<td>16'00000004</td>
</tr>
<tr>
<td>es_nohidesel</td>
<td>16'00000100</td>
</tr>
<tr>
<td>es_number</td>
<td>16'00002000</td>
</tr>
<tr>
<td>es_oemconv</td>
<td>16'00000400</td>
</tr>
<tr>
<td>es_password</td>
<td>16'00000020</td>
</tr>
<tr>
<td>es_readonly</td>
<td>16'00000800</td>
</tr>
<tr>
<td>es_right</td>
<td>16'00000002</td>
</tr>
<tr>
<td>es_uppercase</td>
<td>16'00000008</td>
</tr>
<tr>
<td>es_wantreturn</td>
<td>16'00001000</td>
</tr>
</tbody>
</table>

### Listbox Styles

These are styles that can be applied to listbox controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbs_disablenoscroll</td>
<td>16'00001000</td>
</tr>
<tr>
<td>lbs_extendedsel</td>
<td>16'00000800</td>
</tr>
<tr>
<td>lbs_hasstrings</td>
<td>16'00000040</td>
</tr>
<tr>
<td>lbs_multicolumn</td>
<td>16'00000200</td>
</tr>
<tr>
<td>lbs_multiplesel</td>
<td>16'00000008</td>
</tr>
<tr>
<td>lbs_nodata</td>
<td>16'00002000</td>
</tr>
<tr>
<td>lbs_nointegralheight</td>
<td>16'00000100</td>
</tr>
<tr>
<td>lbs_noredraw</td>
<td>16'00000004</td>
</tr>
<tr>
<td>lbs_nosel</td>
<td>16'00000400</td>
</tr>
</tbody>
</table>
### Combobox Styles

These are styles that can be applied to combobox controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbs_autohscroll</td>
<td>16'000000040</td>
</tr>
<tr>
<td>cbs_disablenoscroll</td>
<td>16'000000800</td>
</tr>
<tr>
<td>cbs_dropdown</td>
<td>16'000000003</td>
</tr>
<tr>
<td>cbs_dropdownlist</td>
<td>16'0000000002</td>
</tr>
<tr>
<td>cbs_hasstrings</td>
<td>16'00000200</td>
</tr>
<tr>
<td>cbs_lowercase</td>
<td>16'00004000</td>
</tr>
<tr>
<td>cbs_nointegralheight</td>
<td>16'00000400</td>
</tr>
<tr>
<td>cbs_oemconvert</td>
<td>16'00000080</td>
</tr>
<tr>
<td>cbs_ownerdrawfixed</td>
<td>16'000000010</td>
</tr>
<tr>
<td>cbs_ownerdrawvariable</td>
<td>16'000000020</td>
</tr>
<tr>
<td>cbs_simple</td>
<td>16'00000001</td>
</tr>
<tr>
<td>cbs_sort</td>
<td>16'00000100</td>
</tr>
<tr>
<td>cbs_uppercase</td>
<td>16'00002000</td>
</tr>
</tbody>
</table>

### Static Styles

These are styles that can be applied to static controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>ss_bitmap</td>
<td>16'0000000e</td>
</tr>
<tr>
<td>ss_blackframe</td>
<td>16'00000007</td>
</tr>
<tr>
<td>ss_blackrect</td>
<td>16'00000004</td>
</tr>
<tr>
<td>ss_center</td>
<td>16'00000001</td>
</tr>
<tr>
<td>ss_centerimage</td>
<td>16'00000200</td>
</tr>
<tr>
<td>ss_endellipsis</td>
<td>16'00000000</td>
</tr>
<tr>
<td>ss_enhmetafile</td>
<td>16'0000000f</td>
</tr>
<tr>
<td>ss_etchedframe</td>
<td>16'00000012</td>
</tr>
<tr>
<td>ss_etchedhorz</td>
<td>16'00000010</td>
</tr>
<tr>
<td>ss_etchedvert</td>
<td>16'00000011</td>
</tr>
<tr>
<td>ss_grayframe</td>
<td>16'00000008</td>
</tr>
<tr>
<td>ss_grayrect</td>
<td>16'00000005</td>
</tr>
<tr>
<td>ss_icon</td>
<td>16'00000003</td>
</tr>
<tr>
<td>ss_left</td>
<td>16'00000000</td>
</tr>
<tr>
<td>ss_leftnowordwrap</td>
<td>16'0000000c</td>
</tr>
<tr>
<td>ss_noprefix</td>
<td>16'00000080</td>
</tr>
<tr>
<td>ss_notify</td>
<td>16'00000100</td>
</tr>
<tr>
<td>ss_ownerdraw</td>
<td>16'000000d</td>
</tr>
<tr>
<td>ss_pathellipsis</td>
<td>16'00000800</td>
</tr>
<tr>
<td>ss_realsizeimage</td>
<td>16'00000800</td>
</tr>
<tr>
<td>ss_right</td>
<td>16'00000002</td>
</tr>
<tr>
<td>ss_rightjust</td>
<td>16'00000400</td>
</tr>
<tr>
<td>ss_simple</td>
<td>16'0000000b</td>
</tr>
<tr>
<td>ss_sunken</td>
<td>16'00001000</td>
</tr>
<tr>
<td>ss_useritem</td>
<td>16'0000000a</td>
</tr>
<tr>
<td>ss_whiteframe</td>
<td>16'00000009</td>
</tr>
<tr>
<td>ss_whiterect</td>
<td>16'00000006</td>
</tr>
<tr>
<td>ss_wordellipsis</td>
<td>16'0000c000</td>
</tr>
</tbody>
</table>
Scrollbar Styles

These are styles that can be applied to scrollbar controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbs_bottomalign</td>
<td>16'0000000004</td>
</tr>
<tr>
<td>sbs_horz</td>
<td>16'0000000000</td>
</tr>
<tr>
<td>sbs_leftalign</td>
<td>16'0000000002</td>
</tr>
<tr>
<td>sbs_rightalign</td>
<td>16'0000000004</td>
</tr>
<tr>
<td>sbs_sizebox</td>
<td>16'0000000008</td>
</tr>
<tr>
<td>sbs_sizeboxbottomrightalign</td>
<td>16'0000000004</td>
</tr>
<tr>
<td>sbs_sizeboxtopleftalign</td>
<td>16'0000000002</td>
</tr>
<tr>
<td>sbs_sizegrip</td>
<td>16'0000000010</td>
</tr>
<tr>
<td>sbs_topalign</td>
<td>16'0000000002</td>
</tr>
<tr>
<td>sbs_vert</td>
<td>16'0000000001</td>
</tr>
</tbody>
</table>

Rich Styles

These are styles that can be applied to rich controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>es_autohscroll</td>
<td>16'0000000080</td>
</tr>
<tr>
<td>es_autovscroll</td>
<td>16'0000000040</td>
</tr>
<tr>
<td>es_center</td>
<td>16'0000000001</td>
</tr>
<tr>
<td>es_left</td>
<td>16'0000000000</td>
</tr>
<tr>
<td>es_multiline</td>
<td>16'0000000004</td>
</tr>
<tr>
<td>es_nohidesel</td>
<td>16'000000100</td>
</tr>
<tr>
<td>es_number</td>
<td>16'000002000</td>
</tr>
<tr>
<td>es_password</td>
<td>16'000000020</td>
</tr>
<tr>
<td>es_readonly</td>
<td>16'000000800</td>
</tr>
<tr>
<td>es_right</td>
<td>16'000000002</td>
</tr>
</tbody>
</table>
### NudgeBar Styles

These are styles that can be applied to nudgebar controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uds_alignleft</td>
<td>16'000000008</td>
</tr>
<tr>
<td>uds_alignright</td>
<td>16'000000004</td>
</tr>
<tr>
<td>uds_arrowkeys</td>
<td>16'000000020</td>
</tr>
<tr>
<td>uds_autobuddy</td>
<td>16'000000010</td>
</tr>
<tr>
<td>uds_horz</td>
<td>16'000000040</td>
</tr>
<tr>
<td>uds_hottack</td>
<td>16'000000100</td>
</tr>
<tr>
<td>uds_nothousands</td>
<td>16'000000080</td>
</tr>
<tr>
<td>uds_setbuddyint</td>
<td>16'000000002</td>
</tr>
<tr>
<td>uds_wrap</td>
<td>16'000000001</td>
</tr>
</tbody>
</table>

### StripBar Styles

These are styles that can be applied to stripbar controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs_smooth</td>
<td>16'000000001</td>
</tr>
</tbody>
</table>
TabBox Styles

These are styles that can be applied to tabbox controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcs_bottom</td>
<td>16'00000002</td>
</tr>
<tr>
<td>tcs_buttons</td>
<td>16'00000100</td>
</tr>
<tr>
<td>tcs_fixedwidth</td>
<td>16'00000400</td>
</tr>
<tr>
<td>tcs_flatbuttons</td>
<td>16'00000008</td>
</tr>
<tr>
<td>tcs_focusnever</td>
<td>16'00008000</td>
</tr>
<tr>
<td>tcs_focusonbuttondown</td>
<td>16'00001000</td>
</tr>
<tr>
<td>tcs_forceiconleft</td>
<td>16'00000010</td>
</tr>
<tr>
<td>tcs_forcelabelleft</td>
<td>16'00000020</td>
</tr>
<tr>
<td>tcs_hottrack</td>
<td>16'00000040</td>
</tr>
<tr>
<td>tcs_multiline</td>
<td>16'00000200</td>
</tr>
<tr>
<td>tcs_multiselect</td>
<td>16'00000004</td>
</tr>
<tr>
<td>tcs_ownerdrawfixed</td>
<td>16'00002000</td>
</tr>
<tr>
<td>tcs_raggedright</td>
<td>16'00000800</td>
</tr>
<tr>
<td>tcs_right</td>
<td>16'00000020</td>
</tr>
<tr>
<td>tcs_rightjustify</td>
<td>16'00000000</td>
</tr>
<tr>
<td>tcs_scrollopposite</td>
<td>16'00000001</td>
</tr>
<tr>
<td>tcs_singleline</td>
<td>16'00000000</td>
</tr>
<tr>
<td>tcs_tabs</td>
<td>16'00000000</td>
</tr>
<tr>
<td>tcs_tooltips</td>
<td>16'00004000</td>
</tr>
<tr>
<td>tcs_vertical</td>
<td>16'00000080</td>
</tr>
</tbody>
</table>

ToolTip Styles

These are styles that can be applied to tooltip controls:
<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tts_alwaystip</td>
<td>16'00000001</td>
</tr>
<tr>
<td>tts_balloon</td>
<td>16'00000040</td>
</tr>
<tr>
<td>tts_noanimate</td>
<td>16'00000010</td>
</tr>
<tr>
<td>tts_nofade</td>
<td>16'00000020</td>
</tr>
<tr>
<td>tts_noprefix</td>
<td>16'00000002</td>
</tr>
</tbody>
</table>

**TrackBar Styles**

These are styles that can be applied to trackbar controls:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tbs_autoticks</td>
<td>16'00000001</td>
</tr>
<tr>
<td>tbs_both</td>
<td>16'00000008</td>
</tr>
<tr>
<td>tbs_bottom</td>
<td>16'00000000</td>
</tr>
<tr>
<td>tbs_enableSelrange</td>
<td>16'00000020</td>
</tr>
<tr>
<td>tbs_fixedlength</td>
<td>16'00000040</td>
</tr>
<tr>
<td>tbs_horz</td>
<td>16'00000000</td>
</tr>
<tr>
<td>tbs_left</td>
<td>16'00000004</td>
</tr>
<tr>
<td>tbs_nothumb</td>
<td>16'00000080</td>
</tr>
<tr>
<td>tbs_noticks</td>
<td>16'00000010</td>
</tr>
<tr>
<td>tbs_reversed</td>
<td>16'00000200</td>
</tr>
<tr>
<td>tbs_right</td>
<td>16'00000000</td>
</tr>
<tr>
<td>tbs_tooltips</td>
<td>16'00000100</td>
</tr>
<tr>
<td>tbs_top</td>
<td>16'00000004</td>
</tr>
<tr>
<td>tbs_vert</td>
<td>16'00000002</td>
</tr>
</tbody>
</table>

**TreeView Styles**

These are styles that can be applied to treeview controls:
### Style

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tvs_hasbuttons</td>
<td>16'00000001</td>
</tr>
<tr>
<td>tvs_haslines</td>
<td>16'00000002</td>
</tr>
<tr>
<td>tvs_linesatroot</td>
<td>16'00000004</td>
</tr>
<tr>
<td>tvs_editlabels</td>
<td>16'00000008</td>
</tr>
<tr>
<td>tvs_disableddragdrop</td>
<td>16'00000010</td>
</tr>
<tr>
<td>tvs_showselectalways</td>
<td>16'00000020</td>
</tr>
<tr>
<td>tvs_rtlreading</td>
<td>16'00000040</td>
</tr>
<tr>
<td>tvs_notooltips</td>
<td>16'00000080</td>
</tr>
<tr>
<td>tvs_checkboxes</td>
<td>16'00000100</td>
</tr>
<tr>
<td>tvs_trackselect</td>
<td>16'00000200</td>
</tr>
<tr>
<td>tvs_singleexpand</td>
<td>16'00000400</td>
</tr>
<tr>
<td>tvs_infotip</td>
<td>16'00000800</td>
</tr>
<tr>
<td>tvs_fullrowselect</td>
<td>16'00001000</td>
</tr>
<tr>
<td>tvs_noscroll</td>
<td>16'00002000</td>
</tr>
<tr>
<td>tvs_nonevenheight</td>
<td>16'00004000</td>
</tr>
<tr>
<td>tvs_nohscroll</td>
<td>16'00008000</td>
</tr>
</tbody>
</table>

### Extended Styles

These are extended styles that can be applied to most windows:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws_ex_acceptfiles</td>
<td>16'00000010</td>
</tr>
<tr>
<td>ws_ex_appwindow</td>
<td>16'00004000</td>
</tr>
<tr>
<td>ws_ex_clientedge</td>
<td>16'00000200</td>
</tr>
<tr>
<td>ws_ex_contexthelp</td>
<td>16'00000400</td>
</tr>
<tr>
<td>ws_ex_controlparent</td>
<td>16'00001000</td>
</tr>
<tr>
<td>ws_ex_dlgmodalframe</td>
<td>16'00000001</td>
</tr>
<tr>
<td>ws_ex_left</td>
<td>16'00000000</td>
</tr>
</tbody>
</table>
Dialog Pseudo-Style

This is a pseudo-style that can be applied to dialog windows:

<table>
<thead>
<tr>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlg_ownedbyprolog</td>
<td>16’00000000</td>
</tr>
</tbody>
</table>
Appendix D - Window Messages

Like any other interactive Windows application, **WIN-PROLOG** contains a "message loop" that runs in the background, throughout a Prolog session, reacting to "messages" generated by the many events that take place in the environment. The vast majority of such messages are handled directly in the **WIN-PROLOG" kernel**, which is programmed in a mixture of C and Assembler code: trying to notify Prolog of every one would entail a considerable run-time overhead, so only a carefully selected set of messages are ever passed on.

Except in the simplest cases, each "dialog" and "user window" in an application needs to be associated with a "window handler" (see `window_handler/2`), and these special "hook" programs interrupt program execution whenever one of the selected messages is received. Consider the following program:

```prolog
foo( Time ) :-
    Dstyle = [ws_popup,ws_caption],
    Bstyle = [ws_visible,ws_child,bs_defpushbutton],
    wdcreate( foo, `foo`, 10, 10, 100, 60, Dstyle ),
    wccreate( (foo,1), button, `bar`, 10, 10, 80, 20, Bstyle ),
    window_handler( foo, foo_handler ),
    call_dialog( foo, Time ).

foo_handler( (foo,1), msg_button, _, (H,M,S) ) :-
    time( _, _, _, H, M, S, _ ).

foo_handler( Window, Message, Data, Result ) :-
    window_handler( Window, Message, Data, Result ).
```

The `foo/1` predicate creates a small dialog (`wdcreate/7`) containing a single button (`wccreate/8`), and associates the window handler, `foo_handler/4`, with this dialog (`window_handler/2`), before invoking the dialog itself (`call_dialog/2`).

The `foo_handler/4` predicate has just two clauses: the first picks up "msg_button" messages received by the "(foo,1)" window, and reacts by binding the handler’s fourth argument to a conjunction containing the time: this is a signal for `call_dialog/2` to terminate, returning this structure. All other messages are passed, by the second clause, to `window_handler/4`.

Messages can be processed at two levels in **WIN-PROLOG**; normally, as above, they are handled in a "window handler", where they have names, their data fields have been preprocessed, and a variable is presented which can be used to return a value to `call_dialog/2` and to hide the dialog itself. It is also possible to process raw messages with the low-level hook predicate, `?MESSAGE?/4`: in the following sections, each message is described in terms of both approaches.
Message Entries

The remainder of this appendix consists of an alphabetical listing of the window messages that are reported to WIN-PROLOG. Each message in this section is described, starting on a fresh page, in a standard format, as outlined below:

message_name

  simple description of message

  handler( Arg1 ... Arg4 )

  ?Arg1 <type1>
  ...
  ?Arg4 <type4>

Handler  The view of this message from a window handler.

hook( Arg1 ... Arg4 )

  ?Arg1 <type1>
  ...
  ?Arg4 <type4>

Hook  The view of this message from a message hook.

Default  An explanation of the default processing of this message.

Notes  Any special notes pertaining to this message.
msg_button

the given button window has been clicked

handler( Window, msg_button, 0, Result )

?Window <window_handle>
?Result <variable>

Handler
This is sent to a "button" Window when it is clicked by the user, or selected by a keyboard shortcut. The normal response is to bind Result to some term indicating the completion of the dialog that contains the button.

'?MESSAGE?'( Window, 6, 0, Goal )

?Window <window_handle>
?Goal <goal>

Hook
This is sent to a "button" Window when it is clicked by the user, or selected by a keyboard shortcut. To allow button names to be returned automatically to call_dialog/2, this message should be passed to message_hook/4 for further processing.

Default
The default WIN-PROLOG window handler reacts to msg_button by checking the button's style, and if it is one of "bs_pushbutton" or "bs_defpushbutton", binding Result to an atom containing the button's text (name), in lowercase and stripped of any single instances of the hotkey character, "&".

Notes
Normally when a dialog is called "modally" by call_dialog/2, its second argument is eventually bound to the name of the button that was clicked to complete the dialog. By writing a window handler to intercept this message, any arbitrary term may be returned. In a modeless dialog, invoked by show_dialog/1, it is more normal for buttons to initiate actions without closing the dialog. In this case, the window handler should simply avoid binding Result upon completing its processing.
**msg_change**

*a change has been made in the given edit window*

```prolog
handler( Window, msg_change, Level, Result )
```

<table>
<thead>
<tr>
<th>Level</th>
<th>Meaning</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>the caret has moved and/or the selection range has changed, but the text is unchanged</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>a successful change has been made to the text</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>an unsuccessful change has been made to the text, possibly resulting in truncation</td>
</tr>
</tbody>
</table>

```prolog
'MESSAGE?'( Window, 5, Level, Goal )
```

**Hook**  
This is sent to an "edit" Window when its selection area or contents are changed. The Level of change is signalled in an integer flag as shown in the table above. To maintain tracking of program modifications in the development environment, this message should be passed to `message_hook/4` for further processing.

**Default**  
The WIN-PROLOG development environment uses the `msg_change` message to keep track of modifications to program windows with respect both to saving and compiling.

**Notes**  
This gives applications an opportunity to perform edit-by-edit processing, for example to enable or disable a button depending upon whether its neighbouring edit control currently contains text.
msg_char

a character has been received by the given window

handler( Window, msg_char, Char, Result )

?Window   <window_handle>
?Char      <integer> or <atom>
?Result    <variable>

Handler

This is sent to a “grafix” Window when any character is typed, or to an "edit" Window when one of the unused control characters is typed. In most cases, Char returns the character code of the key that was returned; certain special keys, however, return negative codes which are automatically mapped to the following names:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>prior</td>
</tr>
<tr>
<td>-2</td>
<td>next</td>
</tr>
<tr>
<td>-3</td>
<td>end</td>
</tr>
<tr>
<td>-4</td>
<td>home</td>
</tr>
<tr>
<td>-5</td>
<td>left</td>
</tr>
<tr>
<td>-6</td>
<td>up</td>
</tr>
<tr>
<td>-7</td>
<td>right</td>
</tr>
<tr>
<td>-8</td>
<td>down</td>
</tr>
<tr>
<td>-9</td>
<td>select</td>
</tr>
<tr>
<td>-10</td>
<td>print</td>
</tr>
<tr>
<td>-11</td>
<td>execute</td>
</tr>
<tr>
<td>-12</td>
<td>snapshot</td>
</tr>
<tr>
<td>-13</td>
<td>insert</td>
</tr>
<tr>
<td>-14</td>
<td>delete</td>
</tr>
</tbody>
</table>

'?type MESSAGE?'( Window, 27, Char, Goal )

?Window   <window_handle>
Hook
This is sent to a "grafix" Window when any character is typed, or to an "edit" Window when one of the unused control characters is typed. In most cases, Char returns the character code of the key that was returned; certain special keys, however, return negative codes which are listed in the table above.

Default
The msg_char message is not used by WIN-PROLOG.

Notes
This message is only generated by three classes of window, "edit", "rich" and "grafix". In the first two cases, it is limited to reporting "unused" control characters. All other characters, including those control characters used by Windows, are processed internally. In the third case, however, there is no default processing of keyboard input, and all characters typed into a grafix window are reported as msg_char messages. A related message, msg_key, reports key down and key up events in most classes of WIN-PROLOG window.
**msg_close**

the given window has received a request to close

```
handler( Window, msg_close, Mode, Result )
```

?-Window <window_handle>
?-Mode <integer> in the domain {0,-1,-2,-3}
?-Result <variable>

**Handler**

This is sent to a Window when its close icon is clicked (Win95/WinNT), its system icon is double-clicked (all versions of Windows) or the user has pressed `<alt-F4>`. Depending upon circumstances, the Window may be closed (`wclose/1`) or hidden (`wshow/2`); if the window is a dialog, Result may be bound to a non-variable term: this will hide the dialog and return the term to any current call to `call_dialog/2`. The Mode indicates the circumstances under which the window has been requested to close:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a normal request has been made to close the given window</td>
</tr>
<tr>
<td>-1</td>
<td>an attempt has been made to shut down the current Windows session</td>
</tr>
<tr>
<td>-2</td>
<td>an attempt has been made to log off the current Windows user</td>
</tr>
<tr>
<td>-3</td>
<td>an attempt has been made to reboot the machine running the current Windows session</td>
</tr>
</tbody>
</table>

```
'?MESSAGE?'( Window, 2, Mode, Goal )
```

?-Window <window_handle>
?-Mode <integer> in the domain {0,-1,-2,-3}
?-Goal <goal>

**Hook**

This is sent to a Window when its close icon is clicked (Win95/WinNT), its system icon is double-clicked (all versions of Windows) or the user has pressed `<alt-F4>`. To enable the normal processing of this message according to window type, it should be passed to `message_hook/4` for further processing. The Mode indicates the circumstances under which the window has been requested to close (see the table above).

**Default**

The `msg_close` message is handled in a variety of ways in the WIN-PROLOG development environment, depending upon the window concerned:
<table>
<thead>
<tr>
<th>Window</th>
<th>Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (main)</td>
<td><strong>WIN-PROLOG</strong> enters its close-down sequence, offering to save any modified program files, before calling exit/1 with the given mode variable</td>
</tr>
<tr>
<td>1 (console)</td>
<td>a warning message box is displayed, and if the user agrees, <strong>WIN-PROLOG</strong> enters its close-down sequence, offering to save any modified program files before calling exit/1</td>
</tr>
<tr>
<td>text (program)</td>
<td>if the window has been modified but not saved, a warning message box is displayed giving the options to save before closing, or to cancel; when the window is closed, its programs are abolished</td>
</tr>
<tr>
<td>text/user</td>
<td>apart from the console and any program windows (see above), text and user windows are simply closed</td>
</tr>
<tr>
<td>dialog</td>
<td>the &quot;result&quot; argument is bound to the atom, &quot;close&quot;, which hides the window and returns this atom and control to call_dialog/2</td>
</tr>
</tbody>
</table>

**Notes**

When this message is hooked with `?MESSAGE?/4`, it is important to be aware of a special *Goal* of the form:

```
wait( (1,Result) )
```

This call to *wait/1* is made by *call_dialog/2*, and provides an efficient "idle loop" as well as a mechanism for returning an eventual *Result* to the latter predicate. If *Result* is bound by a message hook, *call_dialog/2* will hide the window concerned and succeed, returning this binding as its second argument.

The special cases where *Mode* has a value other than zero ("0") are only ever received by the "main window" ("0"), and indicate that an attempt has been made to terminate the current Windows session in one of three ways. This signal is only sent when the "shutdown flag" is set to "1" (see *safe/1*). The correct response, after performing appropriate housekeeping, is to call the *exit/1* predicate with the given *Mode*, in order to resume the Windows shutdown sequence.
msg_double

the given window has been double-clicked

handler( Window, msg_double, 0, Result )

?Window <window_handle>
?Result <variable>

**Handler**  This is sent to a Window when it is double-clicked by the user. The normal response is to bind Result to some term indicating the completion of the dialog in the same manner as if the default pushbutton has been clicked.

'?MESSAGE?'( Window, 8, 0, Goal )

?Window <window_handle>
?Goal <goal>

**Hook**  This is sent to a Window when it is double-clicked by the user.

**Default**  The msg_double message is not used by WIN-PROLOG.

**Notes**  Depending upon how a dialog is set up, double-clicking on certain controls may automatically generate a msg_button message for the default pushbutton, rather than msg_double for the control window that was double-clicked. Normally this happens when the dialog's default pushbutton is given the ID value of "1" (one).
msg_focus

focus has been set to the given window

handler( Window, msg_focus, Handle, Result )

?Window <window_handle>
?Handle <window_handle>
?Result <variable>

**Handler**
This is sent to a Window when it receives focus; the Handle of the window that previously had focus is given.

'?type'( Window, 3, Handle, Goal )

?Window <window_handle>
?Handle <integer>
?Goal <goal>

**Hook**
This is sent to a Window when it receives focus; the raw integer Handle of the window that previously had focus is given. To maintain proper menu enabling and disabling in the development environment, this message should be passed to message_hook/4 for further processing.

**Default**
The WIN-PROLOG development environment uses the msg_focus message as a cue to enable and disable menu items.

**Notes**
When received by a window, this message gives applications an opportunity to perform initialisations before the user gains control of the window.
**msg_fuzzy**

*focus has been lost by the given window*

```
handler( Window, msg_fuzzy, Handle, Result )
```

**Handler**

This is sent to a *Window* when it loses focus; the *Handle* of the window that has gained focus is given.

```
'MESSAGE'( Window, 4, Handle, Goal )
```

**Hook**

This is sent to a *Window* when it loses focus; the raw integer *Handle* of the window that has gained focus is given.

**Default**

The *msg_fuzzy* message is not used by **WIN-PROLOG**.

**Notes**

When received by a window, this message gives applications an opportunity to perform housekeeping after the user loses control of the window.
**msg_help**

context sensitive help has been requested in the given window

```
handler( Window, msg_help, (X,Y), Result )
```

- **Window**: <window_handle>
- **X**: <integer>
- **Y**: <integer>
- **Result**: <variable>

**Handler**

This is sent to a Window when a system help icon is picked up from the title bar and dropped into the client area: the location of the mouse at the drop point is given as an (X,Y) coordinate pair.

```
'?MESSAGE?'( Window, 29, Coords, Goal )
```

- **Window**: <window_handle>
- **Coords**: <integer>
- **Goal**: <goal>

**Hook**

This is sent to a Window when a system help icon is picked up from the title bar and dropped into the client area: the location of the mouse at the drop point is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The msg_help message is not used by **WIN-PROLOG**.

**Notes**

This message provides low-level support for "context sensitive help", but this functionality is currently not used in the environment. In order to be able to obtain this message, a dialog should be created with the "ws_ex_contexthelp" style: this places a system help icon on its title bar, and this can be clicked and dropped into the dialog to indicate that help is required on some item within. It is up to the application to provide this help.
**msg_horz**

*a change has been made to a horizontal scrollbar*

```prolog
handler( Window, msg_horz, Position, Result )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Position</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Result</td>
<td>&lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Handler**

This is sent to a horizontal "nudgebar", "scrollbar" or "trackbar" Window or a Window that contains a horizontal scrollbar when the position of the respective control's position has been moved. The new Position is given.

```
'?MESSAGE?'( Window, 11, Position, Goal )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Position</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Goal</td>
<td>&lt;goal&gt;</td>
</tr>
</tbody>
</table>

**Hook**

This is sent to a horizontal "nudgebar", "scrollbar" or "trackbar" Window or a Window that contains a horizontal scrollbar when the position of the respective control's position has been moved. The new Position is given.

**Default**

The msg_horz message is not used by WIN-PROLOG.

**Notes**

This gives applications an opportunity to modify the contents of numeric fields or scroll graphics in response to user interaction with a horizontal scrollbar.

Please note that, thanks to a quirk in the design of Windows, it is not possible for WIN-PROLOG to react to msg_horz or msg_vert scrollbar messages while the mouse button is held down: upon release, only the most recent scrollbar message is forwarded to WIN-PROLOG.
msg_key

*a keyboard event has been received by the given window*

```
handler( Window, msg_key, (Flag,Scan,Code), Result )
```

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>down</td>
<td>a key has been pressed</td>
</tr>
<tr>
<td>repeat</td>
<td>a key has repeated after being held down</td>
</tr>
<tr>
<td>up</td>
<td>a key has been released</td>
</tr>
</tbody>
</table>

```
'MESSAGE?'( Window, 28, Data, Goal )
```

Hook

This is sent to whichever Window is currently in focus when a keyboard event takes place. A 32-bit integer returns Data about the event: the top 2 bits determine whether the key was pressed, repeated or released; the next 14 bits contain the hardware scan code, and the least-significant 16 bits contain the virtual key code. To enable context-sensitive help in the development environment, this message should be passed to message_hook/4 for further processing.

Default

The WIN-PROLOG development environment uses the msg_key message to detect when the <F1> key is pressed, which is a cue for the delivery of context-sensitive help.

Notes

This message provides low-level support for the keyboard, and can be used to detect keystrokes even if they do not result in input.
**msg_leftdouble**

*the left button has been double-clicked in a grafix window*

```
handler( Window, msg_leftdouble, (X,Y), Result )
```

**Handler**
This is sent to a "grafix" Window when the left mouse button has been double-clicked within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

```
'?MESSAGE?'( Window, 15, Coords, Goal )
```

**Hook**
This is sent to a "grafix" Window when the left mouse button has been double-clicked within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**
The msg_leftdouble message is not used by **WIN-PROLOG**.

**Notes**
This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
**msg_leftdown**

The left mouse button has been clicked in a grafix window

```
handler(Window, msg_leftdown, (X,Y), Result )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?X</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Y</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Result</td>
<td>&lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Handler**

This is sent to a "grafix" Window when the left mouse button has been clicked within its client area: the location of the mouse is given as an (X,Y) coordinate pair.

```
'MESSAGE?'( Window, 14, Coords, Goal )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Coords</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Goal</td>
<td>&lt;goal&gt;</td>
</tr>
</tbody>
</table>

**Hook**

This is sent to a "grafix" Window when the left mouse button has been clicked within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The msg_leftdown message is not used by **WIN-PROLOG**.

**Notes**

This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
msg_leftup

the left mouse button has been released in a grafix window

handler(Window, msg_leftup, (X,Y), Result )

?Window <window_handle>
?X <integer>
?Y <integer>
?Result <variable>

Handler  This is sent to a "grafix" Window when the left mouse button has been released after clicking or double-clicking within its client area: the location of the mouse is given as an (X,Y) coordinate pair.

'?MESSAGE?( Window, 16, Coords, Goal )

?Window <window_handle>
?Coords <integer>
?Goal <goal>

Hook  This is sent to a "grafix" Window when the left mouse button has been released after clicking or double-clicking within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

Default The msg_leftup message is not used by WIN-PROLOG.

Notes  This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
msg_menu

*a menu item has been selected by the user*

```
handler(Window, msg_menu, Item, Result)
```

**Handler**  
This reports the `Window` that was in focus when the given menu `Item` was selected. The normal response to this message is to perform the function associated with `Item`, before allowing processing to continue.

```
' ?MESSAGE? ' ( Window, 0, Item, Goal )
```

**Hook**  
This reports the `Window` that was in focus when the given menu `Item` was selected. To maintain functionality in the development environment, this message should be passed to `message_hook/4` for further processing.

**Default**  
The `WIN-PROLOG` development environment processes `msg_menu` messages with `Item` values in the range 100..999 to support its menu functions, including access to plugins: it discards all other `msg_menu` messages.

**Notes**  
Menu items in the ranges 0..99 and 61440..65535 are processed directly by the `WIN-PROLOG` kernel and Windows respectively; those in the range 100..999 are normally reserved for use by the development environment, and can therefore only be hooked by `?MESSAGE?/4`, and not by a window handler: items in the range 1000..61439 can be processed by both techniques.
**msg_mouseenter**

The mouse has entered a window's client area

```
handler( Window, msg_mouseenter, (X,Y), Result )
```

- ?Window <window_handle>
- ?X <integer>
- ?Y <integer>
- ?Result <variable>

**Handler**

This is sent to a Window when the mouse is first moved into its client area: the location of the mouse is given as an (X,Y) coordinate pair.

```
'?MESSAGE?(' Window, 24, Coords, Goal )
```

- ?Window <window_handle>
- ?Coords <integer>
- ?Goal <goal>

**Hook**

This is sent to a Window when the mouse is first moved into its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The msg_mouseenter message is not used by WIN-PROLOG.

**Notes**

This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
msg_mousehover

the mouse has hovered over a window's client area

handler( Window, msg_mousehover, (X,Y), Result )

?Window <window_handle>
?X <integer>
?Y <integer>
?Result <variable>

Handler
This is sent to a Window when the mouse remains stationary for a short time within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

'?MESSAGE?(' Window, 25, Coords, Goal )

?Window <window_handle>
?Coords <integer>
?Goal <goal>

Hook
This is sent to a Window when the mouse remains stationary for a short time within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

Default
The msg_mousehover message is not used by WIN-PROLOG.

Notes
This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
**msg_mouseleave**

The mouse has left a window's client area

```
handler( Window, msg_mouseleave, (X,Y), Result )
```

| ?Window | <window_handle> |
| ?X      | <integer>       |
| ?Y      | <integer>       |
| ?Result | <variable>      |

**Handler**

This is sent to a Window when the mouse leaves its client area: the location of the mouse is given as an (X,Y) coordinate pair.

```
'?MESSAGE?'( Window, 26, Coords, Goal )
```

| ?Window | <window_handle> |
| ?Coords | <integer>       |
| ?Goal   | <goal>          |

**Hook**

This is sent to a Window when the mouse leaves its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The msg_mouseleave message is not used by WIN-PROLOG.

**Notes**

This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousedownmove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
msg_mousemove

the mouse has moved in window's client area

handler( Window, msg_mousemove, (X,Y), Result )

?Window   <window_handle>
?X         <integer>
?Y         <integer>
?Result    <variable>

**Handler**
This is sent to a Window when the mouse is moved within its client area: the location of the mouse is given as an (X,Y) coordinate pair.

'?MESSAGE?'( Window, 20, Coords, Goal )

?Window   <window_handle>
?Coords   <integer>
?Goal     <goal>

**Hook**
This is sent to a Window when the mouse is moved within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**
The msg_mousemove message is not used by WIN-PROLOG.

**Notes**
This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
**msg_move**

_the position of the given window has changed_

```prolog
handler( Window, msg_move, (X,Y), Result )
```

- **?Window**: <window_handle>
- **?X**: <integer>
- **?Y**: <integer>
- **?Result**: <variable>

**Handler**

This is sent to a `Window` when it is repositioned: its new position is given as an \((X,Y)\) coordinate pair.

```prolog
'MESSAGE'( Window, 10, Coords, Goal )
```

- **?Window**: <window_handle>
- **?Coords**: <integer>
- **?Goal**: <goal>

**Hook**

This is sent to a `Window` when it is repositioned: its new position is given in the 32-bit integer, `Coords`, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The `msg_move` message is not used by **WIN-PROLOG**.

**Notes**

This gives applications an opportunity to reposition or resize other windows in response to modifications to a given window.
msg_paint

the given grafix or button window requires repainting

handler( Window, msg_paint, Type, Result )

?Window <window_handle>
?Type <atom>
?Result <variable>

Handler This is sent to a "grafix" or "button" Window when its contents need to be repainted; the required Type of repainting is represented as one of three atoms, as shown below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>grafix</td>
<td>this is a grafix window whose &quot;dirty&quot; region must be repainted (see gfx_paint/1)</td>
</tr>
<tr>
<td>button_up</td>
<td>this is a button that may be painted in its &quot;up&quot; (released) position (see gfx_begin/1)</td>
</tr>
<tr>
<td>button_down</td>
<td>this is a button that may be painted in its &quot;down&quot; (pressed) position (see gfx_begin/1)</td>
</tr>
</tbody>
</table>

'?MESSAGE?'( Window, 13, Type, Goal )

?Window <window_handle>
?Type <integer>
?Goal <goal>

Hook This is sent to a "grafix" or "button" Window when its contents need to be repainted; the required Type of repainting is represented as an integer which maps onto one of the three atoms shown above: "0" means "grafix", "1" means "button_up" and "2" means "button_down".

Default The msg_paint message is not used by WIN-PROLOG.

Notes When an application receives this message in a "grafix" window, it should repaint the window's "dirty region" by calling gfx_paint/1, gfx/1 and gfx_end/1 as required to restore its appearance. With the two "button" cases, painting is entirely optional, and only necessary when graphical buttons are desired in an application. In this case, however, it is always necessary to paint the entire button, so gfx_begin/1 should normally be called in place of gfx_paint/1.
msg_popup

context menu has been requested in the given window

handler( Window, msg_popup, (X,Y), Result )

?Window <window_handle>
?X <integer>
?Y <integer>
?Result <variable>

Handler This is sent to a Window when the right mouse button is clicked and released: the location of the mouse is given as an (X,Y) coordinate pair.

'?MESSAGE?'( Window, 30, Coords, Goal )

?Window <window_handle>
?Coords <integer>
?Goal <goal>

Hook This is sent to a Window when the right mouse button is clicked and released: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

Default The msg_popup message is used by WIN-PROLOG to display and track a popup edit menu in the console and program windows.

Notes This message provides low-level support for "context menus", allowing window handlers to react intelligently to right clicks. In the environment, it is used to display and track a popup version of the "Edit" menu.
**msg_rightdouble**

The right button has been double-clicked in a grafix window

```
handler( Window, msg_rightdouble, (X,Y), Result )
```

- ?Window <window_handle>
- ?X <integer>
- ?Y <integer>
- ?Result <variable>

**Handler** This is sent to a "grafix" Window when the right mouse button has been double-clicked within its client area: the location of the mouse is given as an (X,Y) coordinate pair.

```
'?><MESSAGE?><'( Window, 18, Coords, Goal )
```

- ?Window <window_handle>
- ?Coords <integer>
- ?Goal <goal>

**Hook** This is sent to a "grafix" Window when the right mouse button has been double-clicked within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default** The msg_rightdouble message is not used by WIN-PROLOG.

**Notes** This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
**msg_rightdown**

The right mouse button has been clicked in a grafix window

```
handler( Window, msg_rightdown, (X,Y), Result )
```

- **?Window**  
  `<window_handle>`
- **?X**  
  `<integer>`
- **?Y**  
  `<integer>`
- **?Result**  
  `<variable>`

**Handler**

This is sent to a "grafix" Window when the right mouse button has been clicked within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

```
'MESSAGE?'( Window, 17, Coords, Goal )
```

- **?Window**  
  `<window_handle>`
- **?Coords**  
  `<integer>`
- **?Goal**  
  `<goal>`

**Hook**

This is sent to a "grafix" Window when the right mouse button has been clicked within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The msg_rightdown message is not used by WIN-PROLOG.

**Notes**

This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemouse) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
**msg_rightup**

the right mouse button has been released in a grafix window

```
handler( Window, msg_rightup, (X,Y), Result )
```

- **Window**: <window_handle>
- **X**: <integer>
- **Y**: <integer>
- **Result**: <variable>

**Handler**  
This is sent to a "grafix" Window when the right mouse button has been released after clicking or double-clicking within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

```
'MESSAGE'( Window, 19, Coords, Goal )
```

- **Window**: <window_handle>
- **Coords**: <integer>
- **Goal**: <goal>

**Hook**  
This is sent to a "grafix" Window when the right mouse button has been released after clicking or double-clicking within its client area: the location of the mouse is given in the 32-bit integer, **Coords**, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**  
The `msg_rightup` message is not used by WIN-PROLOG.

**Notes**  
This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (`msg_leftdown`, `msg_leftdouble`, `msg_leftup`, `msg_rightdown`, `msg_rightdouble`, `msg_rightup`, `msg_wheeldown`, `msg_wheeldouble` and `msg_wheelup`) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (`msg_mouseenter`, `msg_moused hover`, `msg_mouseleave` and `msg_mou semove`) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window’s client area respectively.
**msg_select**

the selection has changed in a listbox or combobox window

```
handler( Window, msg_select, Index, Result )
```

**Handler**

This is sent to a "combobox", "listbox" or "tabbox" Window when its selection is changed: the zero-based Index of the selected item is returned.

```
'?):MESSAGE?('( Window, 7, Index, Goal )
```

**Hook**

This is sent to an "edit" Window when its selection area or contents are changed: the zero-based Index of the selected item is returned.

**Default**

The msg_select message is not used by WIN-PROLOG.

**Notes**

This gives applications an opportunity to perform selection-by-selection processing, for example to enable or disable a button depending upon whether its neighbouring listbox or combobox control currently contains a selection.

Please note that, thanks to a quirk in the design of Windows, listbox controls will only generated msg_select messages if the "lbs_notify" style is included when they are created; "combobox" controls, on the other hand, always generate this message, and so have no corresponding style.
**msg_size**

*the size of the given window has changed*

```prolog
handler( Window, msg_size, (X,Y), Result )
```

| ?Window     | <window_handle> |
| ?X          | <integer>       |
| ?Y          | <integer>       |
| ?Result     | <variable>      |

**Handler**  
This is sent to a *Window* when it is resized: its new size is given as an \((X,Y)\) coordinate pair.

```prolog
'?MESSAGE?'( Window, 9, Coords, Goal )
```

| ?Window     | <window_handle> |
| ?Coords     | <integer>       |
| ?Goal       | <goal>          |

**Hook**  
This is sent to a *Window* when it is resized: its new size is given in the 32-bit integer, Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**  
The *msg_size* message is not used by *WIN-PROLOG*.

**Notes**  
This gives applications an opportunity to reposition or resize other windows in response to modifications to a given window.
msg_sysmenu

*a system menu item has been selected by the user*

```
handler( Window, msg_sysmenu, Item, Result )
```

- ?Window <window_handle>
- ?Item <integer> in the range [1000..61439]
- ?Result <variable>

**Handler**

This reports the *Window* that was in focus when the given system menu *Item* was selected. The normal response to this message is to perform the function associated with *Item*, before allowing processing to continue.

```
'MESSAGE?'( Window, 1, Item, Goal )
```

- ?Window <window_handle>
- ?Item <integer> in the range [100..61439]
- ?Goal <goal>

**Hook**

This reports the *Window* that was in focus when the given system menu *Item* was selected.

**Default**

The msg_sysmenu message is not used by **WIN-PROLOG**.

**Notes**

Menu items in the ranges 0..99 and 61440..65535 are processed directly by the **WIN-PROLOG** kernel and Windows respectively; those in the range 100..999 are normally reserved for use by the development environment, and can therefore only be hooked by 'MESSAGE?/4, and not by a window handler: items in the range 1000..61439 can be processed by both techniques.
msg_touch

a touch event has occurred

\[
\text{handler( Window, msg\_touch, Touch, Result )}
\]

?Window <window\_handle>
?Touch <list>
?Result <variable>

**Handler**
This is sent to any Window which has been touch-enabled, when a touch event occurs, with ID, state and coordinate information for all currently active touch points returned as a list of tuples in Touch.

'\text{?MESSAGE?}'( Window, 80, Handle, Goal )

?Window <window\_handle>
?Handle <integer>
?Goal <goal>

**Hook**
This is sent to any Window which has been touch-enabled, when a touch event occurs, with an already-cancelled touch Handle given for integrity checking.

**Default**
The msg\_touch message is not used by WIN-PROLOG.

**Notes**
Touch input is available on appropriately configured computers running Windows 7 or later, and WIN-PROLOG can track and report up to 16 simultaneous points. Windows must be individually enabled for touch input, using the touch/3 predicate, which is also used to return touch information in the "Hook" version of this message. Another predicate, touch/0, can be used to test whether the operating system supports the touch APIs, although it cannot detect whether a multi-touch input device is present.

In its "Handler" form, this message receives pre-processed output from touch/3, massaged into a list of tuples, each of the form: ID(State,(X,Y)), where "ID" is a unique integer within a given series of touch events, "State" is an atom in the domain, {down,move,up}, and "(X,Y)" are the X and Y client coordinates of the touch point, expressed in pixels relative to the top left of the window. An ID of 0 is given to the "primary" touch point in any given sequence, which is effectively the mouse location. In its "Hook" form, a call must be made to touch/3 with three variables: the first will return a handle, which should match the Handle passed into the message hook: if the values do not match, there is an error in message processing and this should be flagged appropriately. The second argument returned is a timestamp in milliseconds, and the third, a list of integers, each successive group of four representing the ID, State, X and Y coordinates of a touch point.
msg_tvcancel

a treeview cancel event has occurred

handler( Window, msg_tvcancel, Id, Result )

?Window <window_handle>
?Id <integer>
?Result <variable>

Handler This is sent to any "treeview" control when a treeview "cancel" event occurs, with the node's handle being returned in \textit{Id}.

'?MESSAGE?'( Window, 33, Id, Goal )

?Window <window_handle>
?Id <integer>
?Goal <goal>

Hook This is sent to any "treeview" control when a treeview "cancel" event occurs, with the node's handle being returned in \textit{Id}.

Default The \textit{msg_tvcancel} message is not used by \textit{WIN-PROLOG}.

Notes There are eight notification messages for "treeview" controls, none of which are responded to by \textit{WIN-PROLOG} itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
**msg_tvdrag**

*a treeview drag event has occurred*

```prolog
handler( Window, msg_tvdrag, Id, Result )
```

**Handler**

This is sent to any "treeview" control when a treeview "drag" event occurs, with the node's handle being returned in \( \text{Id} \).

```prolog
'RMESSAGE'( Window, 38, Id, Goal )
```

**Hook**

This is sent to any "treeview" control when a treeview "drag" event occurs, with the node's handle being returned in \( \text{Id} \).

**Default**

The `msg_tvdrag` message is not used by WIN-PROLOG.

**Notes**

There are eight notification messages for "treeview" controls, none of which are responded to by WIN-PROLOG itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
msg_tvedit

a treeview edit event has occurred

handler( Window, msg_tvedit, Id, Result )

?Window <window_handle>
?Id <integer>
?Result <variable>

Handler This is sent to any "treeview" control when a treeview "edit" event occurs, with the node's handle being returned in Id.

'?MESSAGE?'( Window, 31, Id, Goal )

?Window <window_handle>
?Id <integer>
?Goal <goal>

Hook This is sent to any "treeview" control when a treeview "edit" event occurs, with the node's handle being returned in Id.

Default The msg_tvedit message is not used by WIN-PROLOG.

Notes There are eight notification messages for "treeview" controls, none of which are responded to by WIN-PROLOG itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or some such, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
**msg_tventer**

*a treeview enter event has occurred*

```
handler( Window, msg_tventer, Id, Result )
```

*Handler* This is sent to any "treeview" control when a treeview "enter" event occurs, with the node's handle being returned in *Id*.

```
'?MESSAGE?(' Window, 37, Id, Goal )
```

*Hook* This is sent to any "treeview" control when a treeview "enter" event occurs, with the node's handle being returned in *Id*.

*Default* The *msg_tventer* message is not used by WIN-PROLOG.

*Notes* There are eight notification messages for "treeview" controls, none of which are responded to by WIN-PROLOG itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
msg_tvhide

a treeview hide event has occurred

handler( Window, msg_tvhide, Id, Result )

?Window       <window_handle>
?Id            <integer>
?Result        <variable>

Handler
This is sent to any "treeview" control when a treeview "hide" event occurs, with the node's handle being returned in Id.

'?MESSAGE?'( Window, 35, Id, Goal )

?Window       <window_handle>
?Id            <integer>
?Goal          <goal>

Hook
This is sent to any "treeview" control when a treeview "hide" event occurs, with the node's handle being returned in Id.

Default
The msg_tvhide message is not used by WIN-PROLOG.

Notes
There are eight notification messages for "treeview" controls, none of which are responded to by WIN-PROLOG itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
msg_tvleave

a treeview leave event has occurred

handler( Window, msg_tvleave, Id, Result )

?Window <window_handle>
?Id <integer>
?Result <variable>

Handler

This is sent to any "treeview" control when a treeview "leave" event occurs, with the node's handle being returned in Id.

'?MESSAGE?'( Window, 36, Id, Goal )

?Window <window_handle>
?Id <integer>
?Goal <goal>

Hook

This is sent to any "treeview" control when a treeview "leave" event occurs, with the node's handle being returned in Id.

Default

The msg_tvleave message is not used by WIN-PROLOG.

Notes

There are eight notification messages for "treeview" controls, none of which are responded to by WIN-PROLOG itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
msg_tvrename

*a treeview rename event has occurred*

```prolog
handler( Window, msg_tvrename, Id, Result )
```

**Handler**
This is sent to any "treeview" control when a treeview "rename" event occurs, with the node's handle being returned in \( \text{Id} \).

```prolog
'?MESSAGE?(' Window, 32, Id, Goal )
```

**Hook**
This is sent to any "treeview" control when a treeview "rename" event occurs, with the node's handle being returned in \( \text{Id} \).

**Default**
The msg_tvrename message is not used by **WIN-PROLOG**.

**Notes**
There are eight notification messages for "treeview" controls, none of which are responded to by **WIN-PROLOG** itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
**msg_tvshow**

*a treeview show event has occurred*

```prolog
handler( Window, msg_tvshow, Id, Result )
```

**Handler**  
This is sent to any "treeview" control when a treeview "show" event occurs, with the node's handle being returned in *Id*.

```prolog
'MESSAGE'( Window, 34, Id, Goal )
```

**Hook**  
This is sent to any "treeview" control when a treeview "show" event occurs, with the node's handle being returned in *Id*.

**Default**  
The *msg_tvshow* message is not used by **WIN-PROLOG**.

**Notes**  
There are eight notification messages for "treeview" controls, none of which are responded to by **WIN-PROLOG** itself, but which can enable user programs to react in real time as these controls are navigated, folded, unfolded and edited by the user. Even if none of these messages is responded to, "treeview" controls will function as expected within a dialog, and can be queried after the user presses an "OK" button or somesuch, to see what was selected or what the control contains.

You can find out more about "treeview" controls, their messages, and how to control them, in Appendix T.
**msg_vert**

*a change has been made to a vertical scrollbar*

```prolog
handler( Window, msg_vert, Position, Result )
```

**Handler**

This is sent to a vertical "nudgebar", "scrollbar" or "trackbar" Window or a Window that contains a vertical scrollbar when the position of the respective control's position has been moved. The new Position is given.

```prolog
'?type MESSAGE?'( Window, 12, Position, Goal )
```

**Hook**

This is sent to a vertical "nudgebar", "scrollbar" or "trackbar" Window or a Window that contains a vertical scrollbar when the position of the respective control's position has been moved. The new Position is given.

**Default**

The msg_vert message is not used by **WIN-PROLOG**.

**Notes**

This gives applications an opportunity to modify the contents of numeric fields or scroll graphics in response to user interaction with a vertical scrollbar.

Please note that, thanks to a quirk in the design of Windows, it is not possible for **WIN-PROLOG** to react to msg_horz or msg_vert scrollbar messages while the mouse button is held down: upon release, only the most recent scrollbar message is forwarded to **WIN-PROLOG**.
**msg_wheeldouble**

The wheel button has been double-clicked in a grafix window

```
handler( Window, msg_wheeldouble, (X,Y), Result )
```

- `?Window` <window_handle>
- `?X` <integer>
- `?Y` <integer>
- `?Result` <variable>

**Handler**

This is sent to a "grafix" Window when the wheel mouse button has been double-clicked within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

```
'?action'( Window, 21, Coords, Goal )
```

- `?Window` <window_handle>
- `?Coords` <integer>
- `?Goal` <goal>

**Hook**

This is sent to a "grafix" Window when the wheel mouse button has been double-clicked within its client area: the location of the mouse is given in the 32-bit integer, `Coords`, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The `msg_wheeldouble` message is not used by WIN-PROLOG.

**Notes**

This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (`msg_leftdown`, `msg_leftdouble`, `msg_leftup`, `msg_rightdown`, `msg_rightdouble`, `msg_rightup`, `msg_wheeldown`, `msg_wheeldouble` and `msg_wheelup`) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (`msg_mouseenter`, `msg_mousehover`, `msg_mouseleave` and `msg_mousemove`) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
msg_wheeldown

the wheel mouse button has been clicked in a grafix window

handler( Window, msg_wheeldown, (X,Y), Result )

?Window <window_handle>
?X <integer>
?Y <integer>
?Result <variable>

Handler This is sent to a "grafix" Window when the wheel mouse button has been clicked within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

'?MESSAGE?'( Window, 22, Coords, Goal )

?Window <window_handle>
?Coords <integer>
?Goal <goal>

Hook This is sent to a "grafix" Window when the wheel mouse button has been clicked within its client area: the location of the mouse is given in the 32-bit integer, Coords, which encodes the \("X" and \("Y\) components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

Default The msg_wheeldown message is not used by WIN-PROLOG.

Notes This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
**msg_wheelup**

the wheel mouse button has been released in a grafix window

```
handler( Window, msg_wheelup, (X,Y), Result )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?X</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Y</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Result</td>
<td>&lt;variable&gt;</td>
</tr>
</tbody>
</table>

**Handler**

This is sent to a "grafix" Window when the wheel mouse button has been released after clicking or double-clicking within its client area: the location of the mouse is given as an \((X,Y)\) coordinate pair.

```
'?typeMESSAGE?'( Window, 23, Coords, Goal )
```

<table>
<thead>
<tr>
<th>?Window</th>
<th>&lt;window_handle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Coords</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>?Goal</td>
<td>&lt;goal&gt;</td>
</tr>
</tbody>
</table>

**Hook**

This is sent to a "grafix" Window when the wheel mouse button has been released after clicking or double-clicking within its client area: the location of the mouse is given in the 32-bit integer, \(\)Coords, which encodes the "X" and "Y" components as a pair of 16-bit integers in its least-significant and most-significant halves respectively.

**Default**

The msg_wheelup message is not used by WIN-PROLOG.

**Notes**

This is one of a family of messages that gives applications an opportunity to react to mouse events at a low level.

Nine messages (msg_leftdown, msg_leftdouble, msg_leftup, msg_rightdown, msg_rightdouble, msg_rightup, msg_wheeldown, msg_wheeldouble and msg_wheelup) are specific to "Grafix" windows, and report when the left, right or wheel mouse button is single-clicked, double-clicked and released respectively.

Four further messages (msg_mouseenter, msg_mousehover, msg_mouseleave and msg_mousemove) are sent to all window classes, reporting when the mouse enters, hovers over, leaves, or simply moves across the window's client area respectively.
Appendix E - The GraFiX Language

This appendix discusses the "GraFiX" language which provides sophisticated, high-level support for graphics within WIN-PROLOG. The typical GraFiX application performs its operations at three specific levels:

<table>
<thead>
<tr>
<th>Level</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>maintenance of physical devices and objects</td>
</tr>
<tr>
<td>2</td>
<td>management of device contexts and physical devices</td>
</tr>
<tr>
<td>3</td>
<td>application of graphics calls to device contexts</td>
</tr>
</tbody>
</table>

The next three sections expand on this theme.

GraFiX: Level 1 Code

The first level of GraFiX is concerned with the creation, deletion and management of graphics devices. These include windows, printers, enhanced metafiles and other physical and logical objects. The level 1 code to prepare for window graphics looks something like this:

```
... wcreate( foo, dialog, ... ),
  wcreate( (foo,123), grafix, ... ),
  ...
  wclose( foo ),
  ...
```

Equivalent code for a printer would look something like this:

```
... prnini( ... ),
  prnpag( _ ),
  ...
  prnend( 0 ),
  ...
```

A key feature of level 1 code is that it is device dependent: the requirements for setting up and managing windows is quite distinct from that required to
set up and manage printers or enhanced metafiles.

**GraFiX: Level 2 Code**

The second level of GraFiX involves the management of "device contexts". These are logical devices which map onto the physical devices created in level 1. Once obtained, a device context can be treated virtually without regard to the physical attributes of the device to which it belongs. Level 2 code has the form:

```prolog
... gfx_begin( ... ),
... gfx_end( ... ),
...
```

The calls to `gfx_begin/1` and `gfx_end/1` identify a window, printer or other physical device which has been set up in level 1, and brackets the level 3 code which performs the actual graphics. Level 2 code is semi-device dependent.

**GraFiX: Level 3 Code**

The third, innermost level of GraFiX comprises the set of predicates concerned with actually performing graphics. Typical code has the form:

```prolog
... gfx( ... ),
... gfx( ... ),
... gfx( ... ),
...
```

Level 3 code is completely device independent, working on the "current device context" set up by the most recent level 2 call to `gfx_begin/1`. Within level 3, in addition to GraFiX calls (`gfx/1`), default mappings, origins and initial object selections can be obtained, and hit testing can be performed.

**The GraFiX Predicates**

The GraFiX subsystem comprises some 50 predicates, all of whose names begin with the characters, "gfx_" with the sole exception of `gfx/1` itself, the main GraFiX predicate. The predicates fall into a number of logical groups, as well as into all three of the levels described above. The following sections examines each of the groups in turn.
GraFiX Predicates: Object Maintenance

As noted above, objects such as brushes and pens are created, deleted and maintained independently of GraFiX calls. Each of nearly a dozen types of object has one or more associated predicates. These include:

<table>
<thead>
<tr>
<th>Object</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>backgrounds</td>
<td><code>gfx_back_close/1</code> <code>gfx_back_create/4</code> <code>gfx_back_dict/2</code> <code>gfx_back_handle/2</code></td>
</tr>
<tr>
<td>bitmaps</td>
<td><code>gfx_bitmap_close/1</code> <code>gfx_bitmap_dict/2</code> <code>gfx_bitmap_handle/2</code> <code>gfx_bitmap_load/2</code></td>
</tr>
<tr>
<td>brushes</td>
<td><code>gfx_brush_close/1</code> <code>gfx_brush_create/5</code> <code>gfx_brush_dict/2</code> <code>gfx_brush_handle/2</code></td>
</tr>
<tr>
<td>cursors</td>
<td><code>gfx_cursor_handle/2</code></td>
</tr>
<tr>
<td>fonts</td>
<td><code>gfx_font_close/1</code> <code>gfx_font_create/4</code> <code>gfx_font_dict/2</code> <code>gfx_font_handle/2</code></td>
</tr>
<tr>
<td>foregrounds</td>
<td><code>gfx_fore_close/1</code> <code>gfx_fore_create/4</code> <code>gfx_fore_dict/2</code> <code>gfx_fore_handle/2</code></td>
</tr>
<tr>
<td>icons</td>
<td><code>gfx_icon_close/1</code> <code>gfx_icon_dict/2</code> <code>gfx_icon_handle/2</code> <code>gfx_icon_load/3</code></td>
</tr>
<tr>
<td>metatiles</td>
<td><code>gfx_metafile_close/1</code> <code>gfx_metafile_dict/2</code> <code>gfx_metafile_handle/2</code> <code>gfx_metafile_load/2</code></td>
</tr>
<tr>
<td>pens</td>
<td><code>gfx_pen_close/1</code> <code>gfx_pen_create/5</code> <code>gfx_pen_dict/2</code> <code>gfx_pen_handle/2</code></td>
</tr>
<tr>
<td>rops</td>
<td><code>gfx_rop_handle/2</code></td>
</tr>
</tbody>
</table>

Of these types of object, several are not true Windows objects. In particular, backgrounds and foregrounds are created by Windows on the fly, unlike brushes, but they have been presented as objects in GraFiX for the sakes of consistency and programming style.

All of the above are level 1 predicates, and should be considered alongside the window handling predicates (`wcreate/8`, `wclose/1`, etc.) and printer predicates (`prnini/4`, `prnend/1`, etc).

GraFiX Predicates: Device Context Management

Several predicates provide for the management of device contexts with respect to their physical devices. These are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal control</td>
<td><code>gfx_begin/1</code> <code>gfx_paint/1</code> <code>gfx_end/1</code></td>
</tr>
<tr>
<td>error handling</td>
<td><code>gfx_cleanup/0</code></td>
</tr>
</tbody>
</table>

The `gfx_begin/1` predicate commences graphics on a given window or printer, by obtaining and saving its device context. The `gfx_paint/1` predicate performs a similar operation, but for windows only, and applies automatic "dirty region" clipping to the window for use in response to "msg_paint" messages.
Both predicates should be mirrored by calls to \texttt{gfx\_end/1} upon completion of graphics; the latter predicate restores the device context before releasing it for other purposes.

The \texttt{gfx\_cleanup/0} predicate is used in response to errors when the nesting of \texttt{gfx\_begin/1} or \texttt{gfx\_paint/1} calls and their corresponding \texttt{gfx\_end/1} calls might have been violated, and should be called in any error handler prior to aborting a series of graphics operations. These predicates belong to level 2.

**GraFiX Predicates: Device Context Settings**

Whenever a device context is obtained through a call to \texttt{gfx\_begin/1} or \texttt{gfx\_paint/1}, it is initialised with a default set of object selections and mappings. These include:

<table>
<thead>
<tr>
<th>Object</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>stock(null_back)</td>
</tr>
<tr>
<td>brush</td>
<td>stock(ltgray_brush)</td>
</tr>
<tr>
<td>font</td>
<td>stock(prolog_fixed_font)</td>
</tr>
<tr>
<td>foreground</td>
<td>stock(white_fore)</td>
</tr>
<tr>
<td>mapping mode</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>origin</td>
<td>(0,0)</td>
</tr>
<tr>
<td>pen</td>
<td>stock(black_pen)</td>
</tr>
<tr>
<td>rop</td>
<td>stock(copypen_rop)</td>
</tr>
</tbody>
</table>

All of the above are defaults inherited by \texttt{gfx/1}, but each can be replaced by one of the following predicates:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>mappings</td>
<td>\texttt{gfx_clipping/4}</td>
</tr>
<tr>
<td>objects</td>
<td>\texttt{gfx_transform/4}</td>
</tr>
<tr>
<td></td>
<td>\texttt{gfx_select/1}</td>
</tr>
</tbody>
</table>

The \texttt{gfx\_resolution/4} predicate can only be used to check the resolution of the current device context. Its return values are used in conjunction with the predicates, \texttt{gfx\_clipping/4}, \texttt{gfx\_mapping/4} and \texttt{gfx\_origin/2} to change the mapping of logical to physical coordinates. The \texttt{gfx\_transform/4} predicate can be used to apply the current settings from \texttt{gfx\_mapping/4} and \texttt{gfx\_origin/2} to convert between logical to physical coordinates, and vice versa, and is useful when processing mouse input. The \texttt{gfx\_select/1} predicate applies objects specified as a series of "equality" expressions to the current device context. These predicates belong to level 3.
### GraFiX Predicates: Hit Tests

Three predicates provide support for "hit tests", facilitating the implementation of interactive graphics programs:

<table>
<thead>
<tr>
<th>Type</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests</td>
<td>gfx_begin/3</td>
</tr>
</tbody>
</table>

A program, typically in response to a mouse message, calls `gfx_begin/3` to initialise a device context for testing, before invoking the graphics being tested. Once the graphics sequence is underway, it calls `gfx_test/1` as often as required to check or return the hit count, which is the number of solid items in the given accumulated GraFiX calls which are directly beneath the mouse cursor. Upon completion of the test, `gfx_end/3` is called to restore the device context. These predicates belong to level 2.

### GraFiX Predicates: Grafix Window Control

Three predicates provide useful control of Grafix windows, providing for the setting of cursors and redrawing and scrolling of their contents:

<table>
<thead>
<tr>
<th>Type</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>gfx_window_cursor/2</td>
</tr>
</tbody>
</table>

These predicates are part of level 1, although they may be called independently of other graphics operations.

### GraFiX Predicates: GraFiX!

Finally, we come to the predicate which actually performs graphics once everything else is in place:

<table>
<thead>
<tr>
<th>Type</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphics</td>
<td>gfx/1</td>
</tr>
</tbody>
</table>

The single argument of this predicate contains a GraFiX procedure, which may consist of anything from a single graphics function to a deeply nested combination of conjunctions and implications. Some examples follow below:

```prolog
gfx( rectangle( 100, 100, 200, 200 ) )
gfx( ( rectangle( 100, 100, 200, 200 ), )
```
ellipse( 150, 150, 250, 250 )
)
)
gfx( (  pen = red
   -> ellipse( 100, 100, 200, 200 )
   )
)

gfx( (  (  brush = blue,
   pen = red
   -> ellipse( 200, 200, 300, 300 ),
   (  pen = stock(black_pen),
       @( 250, 250 ),
       #( 0, 0, 50, 50 )
   -> rectangle( 0, 0, 100, 100 )
   ),
   (  brush = stock(gray_brush)
   -> roundrect( 300, 300, 400, 400, 50, 50 )
   )
)
   polygon( 350, 350, 450, 450, 350, 550 )
)
)

This predicate is firmly part of level 3. Its single argument is comprised of a procedure written in the GraFiX language, and this in turn is discussed in the next few sections.

**The GraFiX Language: Structure**

The GraFiX "Language" is simply a syntactic notation for procedures comprising graphics object selections and primitive calls. The simplest form of procedure is a tuple, resembling a predicate call:

```
functor( arg1 ... argn )
```

Each such tuple calls a single graphics primitive, for example drawing a rectangle or an ellipse. More than one primitive may be called at a time, simply
by combining them into a conjunction:

\[
( \text{functor1( arg1 ... argn )}, \\
\text{...} \\
\text{functorn( ... )} \\
) 
\]

One or more objects may be selected into the device context to control the appearance of the primitive or primitives, using an implication:

\[
( \text{attribute1 = object1,} \\
\text{...} \\
\text{attributen = objectn} \\
\text{\rightarrow functor1( arg1 ... argn ),} \\
\text{...} \\
\text{functorn( ... )} \\
) 
\]

Additionally, either or both of two transformations may be applied, supporting relative offsets and clipping, using the implication:

\[
( \text{@( X, Y ),} \\
\text{#( L, T, R, B ),} \\
\text{...} \\
\text{\rightarrow functor1( arg1 ... argn ),} \\
\text{...} \\
\text{functorn( ... )} \\
) 
\]

Finally, the "then" part of any implication may itself be an implication which inherits its surrounding object selections and transformations, but which applies selected overrides:

\[
( \text{@( X, Y ),} \\
\text{#( L, T, R, B ),} \\
\text{...} \\
\text{\rightarrow functor1( arg1 ... argn ),} \\
\text{\text{( attribute1 = object1,} \\
\text{...} \\
) 
\text{}}} 
\]
attributen = objectn
  -> functor2( ... ),
  ...
),
  ...
  functorn( ... )
)

The GraFiX procedure may be nested arbitrarily deeply, with only Windows limits preventing infinite depth.

**The GraFiX Language: Selectable Objects**

Selectable objects are defined within the Windows GUI, and include such things as brushes and pens. In GraFiX, several pseudo-objects have been added for consistency, including backgrounds, foregrounds and "rop" modes. Objects are selected during the "if" part of an implication, using equality expressions, for example:

```prolog
  ( brush = yellow_stripes,
    pen = stock(white_pen)
  -> ... )
```

In each case, the right hand argument may be:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;atom&gt;</td>
<td>a user-defined named object</td>
</tr>
<tr>
<td>stock(&lt;atom&gt;)</td>
<td>a Windows-defined stock object</td>
</tr>
<tr>
<td>&lt;atom&gt;(&lt;integer&gt;)</td>
<td>any valid handle for an object of given type</td>
</tr>
</tbody>
</table>

The objects which may be selected include:

<table>
<thead>
<tr>
<th>Object</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td>background colour and mode</td>
</tr>
<tr>
<td>brush</td>
<td>brush for filling solid objects</td>
</tr>
<tr>
<td>font</td>
<td>font used for text output</td>
</tr>
</tbody>
</table>
In all cases but "rop", objects may be defined by the user or selected from a collection of "stock" objects. A user defined object is named by a single atom, while a stock object is named by a tuple of arity 1, with the principal functor "stock" and an atomic argument comprising the name of the stock object. It is also possible to use object handles which have been created elsewhere: see the individual object families for further information.

The "rop" case is unique in not allowing user-defined cases; instead, it is necessary to select from one of the 16 stock object handles.

**The GraFiX Language: Selectable Transformations**

Two transformation functions are provided and, like objects, these are selected during the "if" part of an implication, for example:

\[
( \begin{array}{c}
@ ( 100, 100 ) , \\
# ( 125, 125, 175, 175 ) \\
\rightarrow ... \\
\end{array}
) 
\]

Each of these functions works in logical coordinates, and is affected by the current mapping mode. The functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ (X,Y)</td>
<td>add offset (X,Y) to graphics</td>
</tr>
<tr>
<td>#(L,T,R,B)</td>
<td>clip to bounding box (L,T,R,B)</td>
</tr>
</tbody>
</table>

The effects of these functions are cumulative, so the call:

\[
( \begin{array}{c}
@ (10,10) , \\
@ (10,10) \\
\rightarrow ... \\
\end{array}
) 
\]

is equivalent to (but less efficient than):

\[
( \begin{array}{c}
@ (20,20) \\
\end{array}
) 
\]
The transformation functions may be mixed freely with object selections.

**The GraFiX Language: Graphics Primitives**

The main body of a GraFiX procedure, be it a single primitive function, a conjunction of primitive functions, or the "then" part of an implication, performs the actual drawing of graphics, for example:

```prolog
rectangle( 100, 100, 200, 200 )

( ... -> ellipse( 150, 150, 250, 250 )
)
```

Each primitive maps more or less directly onto a Windows GUI function of the same name, with only a couple of exceptions. It should be noted that none of the following primitives is available as a Prolog predicate: they only have meaning in the context of a call to gfx/1. The next several sections describe each primitive in turn:

**arc(x1,y1,x2,y2,xs,ys,xf,yf)**

Draw an arc, the ellipse of whose bounding rectangle has opposite corners at (x1,y1) and (x2,y2), and whose ends are defined by the intersection of that ellipse with straight lines passing through its centre and the points (xs,ys) and (xf,yf), using the current pen.

**bitmap(x1,y1,x2,y2,xo,yo,bitmap)**

Draw a bitmap with its opposite corners at (x1,y1) and (x2,y2), starting at the offset (xo,yo) within the bitmap. Note that the given size does not define the stretching or shrinking of the image size, but how much of the bitmap is displayed. The bitmap will be scaled in accordance with the mapping. The bitmap can be any valid bitmap handle.

**chord(x1,y1,x2,y2,xs,ys,xf,yf)**

Draw a filled chord, the ellipse of whose bounding rectangle has opposite corners at (x1,y1) and (x2,y2), and whose ends are defined by the intersection of that ellipse with straight lines passing through its centre and the points (xs,ys) and (xf,yf), using the current pen and brush.
ellipse(x1,y1,x2,y2)

Draw a filled ellipse, whose bounding rectangle has opposite corners at (x1,y1) and (x2,y2), using the current pen and brush.

icon(x1,y1,icon)

Draw an icon with its top left corner at (x1,y1). Note that icons have a default size (generally 32*32 units), but this will be scaled in accordance with the current mapping. The icon can be any valid icon handle.

metafile(x1,y1,x2,y2,meta)

Draw an enhanced metafile, scaled to a bounding rectangle with opposite corners at (x1,y1) and (x2,y2). Note that depending upon the enhanced metafile, it may not fill or may exceed the given bounding rectangle. The enhanced metafile be can any valid enhanced metafile handle.

pie(x1,y1,x2,y2,xs,ys,xf,yf)

Draw a filled pie, the ellipse of whose bounding rectangle has opposite corners at (x1,y1) and (x2,y2), and whose ends are defined by the intersection of that ellipse with straight lines passing through its centre and the points (xs,ys) and (xf,yf), using the current pen and brush.

polybezier(x1,y1,x11,y11,x12,y12,x2,y2,..xn,yn)

Draw a polybezier, using the current pen, starting at (x1,y1), through all the remaining points (x2,y2)..(xn,yn), using the control points (x11,y11) and (x12,y12) .. (xm1,ym1) and (xm2,ym2) for each successive segment. No line is drawn between (xn,yn) and (x1,y1), and the resulting "open" graphics figure is not filled.

polygon(x1,y1,x2,y2,..xn,yn)

Draw a polygon, using the current pen, starting at (x1,y1), through all the remaining points (x2,y2)..(xn,yn). The figure is "closed" by drawing a line between (xn,yn) and (x1,y1), and is filled with the current brush using an alternate filling algorithm.

polyline(x1,y1,x2,y2,..xn,yn)

Draw a polyline, using the current pen, starting at (x1,y1), through all the remaining points (x2,y2)..(xn,yn). No line is drawn between (xn,yn) and (x1,y1), and the resulting "open" graphics figure is not filled.
rectangle(x1,y1,x2,y2)

Draw a filled rectangle whose opposite corners are at (x1,y1) and (x2,y2), using the current pen and brush.

roundrect(x1,y1,x2,y2,xd,yd)

Draw a filled rectangle with rounded corners, whose bounding rectangle has opposite corners at (x1,y1) and (x2,y2), and whose corner diameters are specified by (xd,yd), using the current pen and brush.

text(x1,y1,string)

Draw a text string, starting at the point (x1,y1), using the current font, foreground colour, background colour and transparency mode.


The remainder of this document summaries GraFiX predicates in turn, including the enumeration of any associated stock objects or reserved values: more formal descriptions of each can be found in the main alphabetical reference section:

gfx/1

Perform a GraFiX procedure on the current device context (see gfx_begin/1, gfx_paint/1 and gfx_begin/3).

gfx_back_close/1

Close the named background object.

gfx_back_create/4

Create a named background object with the specified red, green and blue values. Each of the three colour channels consists of an integer in the range (0..255) corresponding to (dark..light). Background objects are selected into a device context to define the colour of the background of text and patterned brushes.

gfx_back_dict/2

Return a dictionary of named background objects as a list.
gfx_back_handle/2

Convert between a background object and its numerical handle. The object may be a named object, a handle of the form "back(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>white_back</td>
<td>white (0% black) background</td>
</tr>
<tr>
<td>ltgray_back</td>
<td>light grey (25% black) background</td>
</tr>
<tr>
<td>gray_back</td>
<td>mid grey (50% black) background</td>
</tr>
<tr>
<td>dkgray_back</td>
<td>dark grey (75% black) background</td>
</tr>
<tr>
<td>black_back</td>
<td>black (100% black) background</td>
</tr>
<tr>
<td>null_back</td>
<td>null (transparent) background</td>
</tr>
</tbody>
</table>

gfx_begin/1

Obtain and save the device context for the given window, enabling graphics to commence. Any existing device context is saved, localising the side effects of any settings. The new device context begins with default settings.

Provided it has been initialised with either prnbox/4 or prnini/4, followed by a call to prnpag/1, the printer may be specified by providing an empty list ("[]") in place of the window argument.

Note that all calls to gfx_begin/1 should be matched by calls to gfx_end/1 when graphics are complete, because device contexts are scarce resources.

gfx_begin/3

Obtain and save the device context for the given window, enabling hit testing on the given X and Y coordinates to commence. Any existing device context is saved, localising the side effects of any settings. The new device context begins with default settings.

During hit testing, graphics calls perform no output, but test the number of intersections between the given test point and any solid shapes defined in those calls. At any time during testing, the current hit count can be returned by calling gfx_test/1.

Note that all calls to gfx_begin/3 should be matched by calls to gfx_end/3 when hit testing is complete, because device contexts are scarce resources.
gfx_bitmap_close/1

Close the named bitmap object.

gfx_bitmap_dict/2

Return a dictionary of named bitmap objects as a list.

gfx_bitmap_handle/2

Convert between a bitmap object and its numerical handle. The object may be a named object, or a handle of the form "bitmap(INT)", where "INT" is the numerical handle. There are no stock bitmaps.

gfx_bitmap_load/2

Load a bitmap object from the named disk file.

gfx_brush_close/1

Close the named brush object.

gfx_brush_create/5

Create a named brush object with the specified red, green, blue and style parameters. Each of the three colour channels consists of an integer in the range (0..255) corresponding to (dark..light), and the style must be one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>solid</td>
<td>(   ) solid brush</td>
</tr>
<tr>
<td>horizontal</td>
<td>(-----) horizontal hatch</td>
</tr>
<tr>
<td>vertical</td>
<td>(     ) vertical hatch</td>
</tr>
<tr>
<td>fдиagonal</td>
<td>(:::\) forwards diagonal hatch</td>
</tr>
<tr>
<td>bдиagonal</td>
<td>(\\) backwards diagonal hatch</td>
</tr>
<tr>
<td>cross</td>
<td>(+ + + + +) cross hatch</td>
</tr>
<tr>
<td>diagcross</td>
<td>(XXXXX) diagonal cross hatch</td>
</tr>
</tbody>
</table>
Brush objects are selected into a device context to define the fill colour of enclosed shapes; with hatched brushes, the colour and mode of the gaps is defined by the currently selected background object.

### gfx_brush_dict/2

Return a dictionary of named brush objects as a list.

### gfx_brush_handle/2

Convert between a brush object and its numerical handle. The object may be a named object, a handle of the form "brush(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>white_brush</td>
<td>white (0% black) brush</td>
</tr>
<tr>
<td>ltgray_brush</td>
<td>light grey (25% black) brush</td>
</tr>
<tr>
<td>gray_brush</td>
<td>mid grey (50% black) brush</td>
</tr>
<tr>
<td>dkgray_brush</td>
<td>dark grey (75% black) brush</td>
</tr>
<tr>
<td>black_brush</td>
<td>black (100% black) brush</td>
</tr>
<tr>
<td>null_brush</td>
<td>null (transparent) brush</td>
</tr>
</tbody>
</table>

### gfx_cleanup/0

Explicitly restore all saved graphics device contexts, for example after an error has disrupted a graphics program. Device contexts are scarce resources, and should be restored when not in use. This predicate should only be called in an error handler that detects problems with graphics.

### gfx_clipping/4

Set a clipping rectangle for the current device context, using the given left, top, right and bottom device coordinates, or return the current clipping rectangle.

### gfx_cursor_handle/2

Convert between a cursor object and its numerical handle. The object may be a named object, a handle of the form "cursor(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:
### Name | Meaning
--- | ---
arrow_cursor | slanted arrow cursor
ibeam_cursor | i-beam text cursor
wait_cursor | hourglass cursor
cross_cursor | small cross cursor
uparrow_cursor | upward arrow cursor
size_cursor | four-headed arrow cursor
icon_cursor | small square cursor
sizenwse_cursor | nw/se two-headed arrow cursor
sizenesw_cursor | ne/sw two-headed arrow cursor
sizewe_cursor | w/e two-headed arrow cursor
sizes Cursor | n/s two-headed arrow cursor
sizeall_cursor | Windows size all cursor
no_cursor | Windows no cursor
appstarting_cursor | Windows application starting cursor

### gfx_end/1

Restore and release the device context for the given window, causing graphics to terminate. Any previous device context is reenabled, allowing its graphics to continue.

Provided it has been initialised with either `prnbox/4` or `prnini/4`, followed by a call to `prnpag/1`, the printer may be specified by providing an empty list ("[]") in place of the window argument.

Note that `gfx_end/1` can only be called after a matching call to `gfx_begin/1` or `gfx_paint/1`.

### gfx_end/3

Restore and release the device context for the given window, causing hit testing on the given X and Y coordinates to terminate. Any previous device context is reenabled, allowing its graphics to continue.

Note that `gfx_end/3` can only be called after a matching call to `gfx_begin/3`. 
gfx_font_close/1

Close the named font object.

gfx_font_create/4

Create a named font object with the typeface, point size and style parameters. The style must be one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>normal roman font</td>
</tr>
<tr>
<td>italic</td>
<td>normal italic font</td>
</tr>
<tr>
<td>bold</td>
<td>bold roman font</td>
</tr>
<tr>
<td>bolditalic</td>
<td>bold italic font</td>
</tr>
</tbody>
</table>

Font objects are selected into a device context to define the appearance of text, which is drawn with the fill colours defined by the currently selected foreground and background objects.

gfx_font_dict/2

Return a dictionary of named font objects as a list.

gfx_font_handle/2

Convert between a font object and its numerical handle. The object may be a named object, a handle of the form "font(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog_fixed_font</td>
<td>Prolog fixed font (Courier New 10pt)</td>
</tr>
<tr>
<td>oem_fixed_font</td>
<td>OEM fixed font (IBM-PC ROM BIOS char set)</td>
</tr>
<tr>
<td>ansi_fixed_font</td>
<td>ANSI fixed font (ANSI char set)</td>
</tr>
<tr>
<td>ansi_var_font</td>
<td>ANSI var font (ANSI char set)</td>
</tr>
<tr>
<td>system_font</td>
<td>system var font (ANSI char set)</td>
</tr>
<tr>
<td>device_default_font</td>
<td>default fixed font (ANSI char set)</td>
</tr>
</tbody>
</table>
**gfx_fore_close/1**

Close the named foreground object.

**gfx_fore_create/4**

Create a named foreground object with the specified red, green and blue parameters. Each of the three colour channels consists of an integer in the range (0..255) corresponding to (dark..light). Foreground objects are selected into a device context to define the colour of the foreground of text.

**gfx_fore_dict/2**

Return a dictionary of named foreground objects as a list.

**gfx_fore_handle/2**

Convert between a foreground object and its numerical handle. The object may be a named object, a handle of the form "fore(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>white_fore</td>
<td>white (0% black) foreground</td>
</tr>
<tr>
<td>ltgray_fore</td>
<td>light grey (25% black) foreground</td>
</tr>
<tr>
<td>gray_fore</td>
<td>mid grey (50% black) foreground</td>
</tr>
<tr>
<td>dkgray_fore</td>
<td>dark grey (75% black) foreground</td>
</tr>
<tr>
<td>black_fore</td>
<td>black (100% black) foreground</td>
</tr>
</tbody>
</table>

**gfx_icon_close/1**

Close the named icon object.

**gfx_icon_dict/2**

Return a dictionary of named icon objects as a list.
gfx_icon_handle/2

Convert between an icon object and its numerical handle. The object may be a named object, a handle of the form "icon(INT)" where "INT" is the numerical handle, or a stock object of the form "stock(NAME)" where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>application_icon</td>
<td>simple square icon</td>
</tr>
<tr>
<td>hand_icon</td>
<td>stop sign icon</td>
</tr>
<tr>
<td>question_icon</td>
<td>question mark icon</td>
</tr>
<tr>
<td>exclamation_icon</td>
<td>exclamation mark icon</td>
</tr>
<tr>
<td>asterisk_icon</td>
<td>information icon</td>
</tr>
</tbody>
</table>

gfx_icon_load/3

Load an icon object from the named disk file and size.

gfx_mapping/4

Set up the Xw and Yw window extent, and Xv and Yv viewport extent for the current device context, defining its logical to physical coordinate mapping, or return the current coordinate mapping.

gfx_metafile_close/1

Close the named enhanced metafile object.

gfx_metafile_dict/2

Return a dictionary of named enhanced metafile objects as a list.

gfx_metafile_handle/2

Convert between an enhanced metafile object and its numerical handle. The object may be a named object, or a handle of the form "metafile(INT)" where "INT" is the numerical handle. There are no stock enhanced metafiles.
**gfx_metafile_load/2**

Load an enhanced metafile object from the named disk file.

**gfx_origin/2**

Set up the Xo and Yo viewport origin for the current device context, defining its physical origin, or return the current origin.

**gfx_paint/1**

Obtain and save the clipped device context for the given window, enabling painting to commence. Any existing device context is saved, localising the side effects of any settings. The new device context begins with default settings.

Each "Grafix" window maintains a region which automatically accumulates "dirty" areas whenever WM_PAINT message are received by WIN-PROLOG, and this predicate selects the accumulated region into the returned device context as its clipping region, before nullifying the stored region. This means that any graphics applied to the returned device context are automatically clipped to the invalid portions of the given window.

Note that all calls to `gfx_paint/1` should be matched by calls to `gfx_end/1` when graphics are complete, because device contexts are scarce resources.

**gfx_pen_close/1**

Close the named pen object.

**gfx_pen_create/5**

Create a named pen object with the specified red, green, blue and style parameters. Each of the three colour channels consists of an integer in the range (0..255) corresponding to (dark..light), and the style must either be an integer specifying the thickness of the pen, or one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>solid</td>
<td>(    ) solid pen</td>
</tr>
<tr>
<td>dash</td>
<td>(-----) dashed pen</td>
</tr>
<tr>
<td>dot</td>
<td>(......) dotted pen</td>
</tr>
<tr>
<td>dashdot</td>
<td>(<em>.</em>._) dashed/single dotted pen</td>
</tr>
<tr>
<td>dashdotdot</td>
<td>(<em>.</em>._) dashed/double dotted pen</td>
</tr>
</tbody>
</table>

WIN-PROLOG 7.0 - Technical Reference
Pen objects are selected into a device context to define the outline colour of shapes; with dashed and dotted pens, the colour and mode of the gaps is defined by the currently selected background object.

### gfx_pen_dict/2

Return a dictionary of named pen objects as a list.

### gfx_pen_handle/2

Convert between a pen object and its numerical handle. The object may be a named object, a handle of the form "pen(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>white_pen</td>
<td>white (0% black) pen</td>
</tr>
<tr>
<td>black_pen</td>
<td>black (100% black) pen</td>
</tr>
<tr>
<td>null_pen</td>
<td>null (transparent) pen</td>
</tr>
</tbody>
</table>

### gfx_resolution/4

Get or check the physical width, height, horizontal resolution and vertical resolution of the current device context. Used together with gfx_mapping/4 and gfx_origin/4, this predicate allows real-world measurements to be applied to arbitrary devices and scaling of graphics to fit display or printer device contexts.

### gfx_rop_handle/2

Convert between a rop object and its numerical handle. The object may be a handle of the form "rop(INT)", where "INT" is the numerical handle, or a stock object of the form "stock(NAME)", where "NAME" is one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>black_rop</td>
<td>black raster operation mode</td>
</tr>
<tr>
<td>notmergepen_rop</td>
<td>not merge pen raster operation mode</td>
</tr>
<tr>
<td>masknotpen_rop</td>
<td>mask not pen raster operation mode</td>
</tr>
<tr>
<td>notcopypen_rop</td>
<td>not copy pen raster operation mode</td>
</tr>
</tbody>
</table>
### gfx_select/1

Select one or more objects or transformations into the current device context (see `gfx_begin/1`, `gfx_paint/1` and `gfx_begin/3`). The object or conjunction of objects is in the same format as that of the "if" part of implication expressions in GraFiX procedures, and allows default selections to be made which persist in a device context. The items which may be selected include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td>background colour and mode</td>
</tr>
<tr>
<td>brush</td>
<td>brush for filling solid objects</td>
</tr>
<tr>
<td>font</td>
<td>font used for text output</td>
</tr>
<tr>
<td>fore</td>
<td>foreground colour</td>
</tr>
<tr>
<td>pen</td>
<td>pen for outlining objects</td>
</tr>
<tr>
<td>rop</td>
<td>raster operator combination mode</td>
</tr>
<tr>
<td>@(X,Y)</td>
<td>add offset (X,Y) to graphics</td>
</tr>
<tr>
<td>#(L,T,R,B)</td>
<td>clip to bounding box (L,T,R,B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>maskpennot_rop</td>
<td>mask pen not raster operation mode</td>
</tr>
<tr>
<td>not_rop</td>
<td>not raster operation mode</td>
</tr>
<tr>
<td>xorpen_rop</td>
<td>xor pen raster operation mode</td>
</tr>
<tr>
<td>notmaskpen_rop</td>
<td>not mask pen raster operation mode</td>
</tr>
<tr>
<td>maskpen_rop</td>
<td>mask pen raster operation mode</td>
</tr>
<tr>
<td>notxorpen_rop</td>
<td>not xor pen raster operation mode</td>
</tr>
<tr>
<td>nop_rop</td>
<td>nop raster operation mode</td>
</tr>
<tr>
<td>mergenotpen_rop</td>
<td>merge not pen raster operation mode</td>
</tr>
<tr>
<td>copypen_rop</td>
<td>copy pen raster operation mode</td>
</tr>
<tr>
<td>mergepennot_rop</td>
<td>merge pen not raster operation mode</td>
</tr>
<tr>
<td>mergepen_rop</td>
<td>merge pen raster operation mode</td>
</tr>
<tr>
<td>white_rop</td>
<td>white raster operation mode</td>
</tr>
</tbody>
</table>
gfx_test/1
Get or check the number of hits encountered since the start of the current hit test (see gfx_begin/3). This predicate may be called any number of times during a test, allowing tests to be exhaustive or to end after a given number of hits.

gfx_transform/4
Convert between the logical coordinates, Xl and Yl, and their physical equivalents, Xd and Yd, or vice versa, using the current mapping and origin (see gfx_mapping/4 and gfx_origin/2).

gfx_window_cursor/2
Set the cursor handle in the given "Grafix" window, causing the given cursor to be displayed whenever the cursor is over the window's client area.

gfx_window_redraw/5
Invalidate the specified portion of the client area of the given "Grafix" window, causing a "msg_paint" message to be sent to signal a redraw of the contents.

gfx_window_scroll/3
Scroll the client area of the given "Grafix" window by the given number of horizontal and vertical pixels, causing a "msg_paint" message to be sent to signal a redraw of the newly exposed contents.
Appendix F - The Development Environment

The WIN-PROLOG "development environment" provides a rich selection of functions through a series of "pull-down" menus, and most of these can be programmed and automated using special development-time predicate, system_menu/3. As an example, the following call performs the same function as selecting the "File/New" menu option by hand:

?- system_menu( _, file, new ).
  yes

For those menu options that bring up subsidiary dialogs, different variants of calls to system_menu/3 allow the programmer to choose whether to allow these dialogs to be invoked, or to supply the relevant information directly. For example, the following call performs a "File/Save As" operation on the "untitled" window that was created by the previous command, showing the "save as" dialog; for the sake of illustration, we will assume the user responded to the dialog by clicking the "Cancel" button in this instance:

?- system_menu( 'Untitled-1', file, save_as ).
  no

The same "File/Save As" operation could have been completely automated, without user intervention, by including a file name as an argument to "save_as"; in this case, the "save as" dialog is not shown, and the user has no chance to cancel the operation:

?- system_menu( 'Untitled-1', file, save_as(foo) ).
  yes

The "Untitled-1" window created earlier has now been saved as "foo.pl", in the current working directory.

Most functions work specifically on particular "classes" of windows, for example "prolog" or "text" windows. For this reason, an existing window handle should always be passed in as an argument: in cases such as "File/New", where the operation is only loosely related to an existing window, it is permitted to use the console window, "1", as the handle.

The File Menu: system_menu( W, file, F )

The following table describes the functions available from the "file" menu; each of these is invoked by specifying the atom, "file", as the second argument in the call to system_menu/3:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>Create a new window</td>
</tr>
<tr>
<td>open</td>
<td>Open an existing window</td>
</tr>
<tr>
<td>close</td>
<td>Close a window</td>
</tr>
<tr>
<td>save</td>
<td>Save the window contents</td>
</tr>
<tr>
<td>save_as</td>
<td>Save the window contents as a file</td>
</tr>
<tr>
<td>print</td>
<td>Print the contents of the window</td>
</tr>
<tr>
<td>print_as</td>
<td>Print the contents of the window as a file</td>
</tr>
<tr>
<td>select</td>
<td>Select an existing window</td>
</tr>
<tr>
<td>select_as</td>
<td>Select a file to open or save</td>
</tr>
<tr>
<td>kill</td>
<td>Kill the window</td>
</tr>
<tr>
<td>quit</td>
<td>Quit the application</td>
</tr>
</tbody>
</table>
### Function Meanings

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>use the “new” dialog to allow the user to create a new, untitled program window</td>
</tr>
<tr>
<td>new(Class)</td>
<td>create a new, untitled program window, using the specified window class, such as &quot;prolog&quot;</td>
</tr>
<tr>
<td>open</td>
<td>use the “open” dialog to allow the user to choose one or more files to open as program windows</td>
</tr>
<tr>
<td>open(Files)</td>
<td>open the given files into program windows: &quot;files&quot; may be a single logical file name or a list of names</td>
</tr>
<tr>
<td>save</td>
<td>save the given program window; use the &quot;save as&quot; dialog if the specified window is &quot;untitled&quot;</td>
</tr>
<tr>
<td>save_as</td>
<td>save the given program window as a different name, using the &quot;save as&quot; dialog</td>
</tr>
<tr>
<td>save_as(File)</td>
<td>save the given program window using the given logical file name</td>
</tr>
<tr>
<td>save_all</td>
<td>display the &quot;save all&quot; dialog to allow the user to save one or more files</td>
</tr>
<tr>
<td>close</td>
<td>close the given program window; offer to save if it has been modified since last being saved</td>
</tr>
<tr>
<td>close_all</td>
<td>close all program windows; offer to save any that have been modified since last being saved</td>
</tr>
<tr>
<td>import</td>
<td>use the “import” dialog to allow the user to import data into an existing window</td>
</tr>
<tr>
<td>import(File)</td>
<td>import the given file into an existing window using the given logical file name</td>
</tr>
<tr>
<td>export</td>
<td>use the “export” dialog to allow the user to export data from an existing window</td>
</tr>
<tr>
<td>export(File)</td>
<td>export the given file from an existing window using the given logical file name</td>
</tr>
<tr>
<td>load</td>
<td>use the &quot;load&quot; dialog to allow the user to choose one or more files to load directly into memory</td>
</tr>
<tr>
<td>load(Files)</td>
<td>load the given files directly into memory: &quot;files&quot; may be a single logical file name or a list of names</td>
</tr>
<tr>
<td>project</td>
<td>create a project using the &quot;project&quot; dialog</td>
</tr>
<tr>
<td>project(File)</td>
<td>create a project using the given logical file name</td>
</tr>
<tr>
<td>print</td>
<td>show the print dialog; offer to print the contents of the given window</td>
</tr>
<tr>
<td>print_setup</td>
<td>show the &quot;print setup&quot; dialog</td>
</tr>
<tr>
<td>exit</td>
<td>exit from Prolog; offer to save any program windows that have been modified since last being saved</td>
</tr>
</tbody>
</table>

### The Edit Menu: system_menu( W, edit, F )

The following table describes the functions available from the "edit" menu; each of these is invoked by specifying the atom, "edit", as the second argument in the call to system_menu/3:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>undo</td>
<td>undo the most recent edit on the given window</td>
</tr>
<tr>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>redo</td>
<td>redo the most recent edit on the given window</td>
</tr>
<tr>
<td>cut</td>
<td>cut from the given window into the clipboard</td>
</tr>
<tr>
<td>copy</td>
<td>copy from the given window into the clipboard</td>
</tr>
<tr>
<td>paste</td>
<td>paste from the clipboard into the given window</td>
</tr>
<tr>
<td>clear</td>
<td>delete from the given window</td>
</tr>
<tr>
<td>select_all</td>
<td>select all text in the given window</td>
</tr>
<tr>
<td>select_query</td>
<td>select query text in the given window</td>
</tr>
<tr>
<td>empty_console</td>
<td>empty the console if the user agrees to a prompt</td>
</tr>
</tbody>
</table>

The Search Menu: `system_menu( W, search, F )`

The following table describes the functions available from the "search" menu; each of these is invoked by specifying the atom, "search", as the second argument in the call to `system_menu/3`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>show the &quot;find&quot; dialog</td>
</tr>
<tr>
<td>change</td>
<td>show the &quot;change&quot; dialog</td>
</tr>
<tr>
<td>goto_definition</td>
<td>show the &quot;goto definition&quot; dialog; allow the user to choose a predicate to find, starting from the given program window</td>
</tr>
<tr>
<td>goto_definition(Prd/Ary)</td>
<td>go to the definition of a predicate, &quot;Prd/Ary&quot;, starting from the given program window</td>
</tr>
<tr>
<td>goto_next_clause</td>
<td>go to the next clause in a found definition</td>
</tr>
<tr>
<td>locate_files</td>
<td>show the &quot;locate files&quot; dialog, allowing the user to initiate external file searches</td>
</tr>
<tr>
<td>compare_files</td>
<td>show the &quot;compare files&quot; dialog, allowing the user to initiate external file comparisons</td>
</tr>
</tbody>
</table>

The Run Menu: `system_menu( W, query, F )`

The following table describes the functions available from the "run" menu; each of these is invoked by specifying the atom, "run", as the second argument in the call to `system_menu/3`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_syntax</td>
<td>perform a syntax check on the given program window</td>
</tr>
<tr>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cross_reference</td>
<td>show the &quot;cross reference&quot; dialog, allowing the user to choose one or more program windows and to perform a cross reference</td>
</tr>
<tr>
<td>cross_reference(Wins)</td>
<td>perform a cross reference on the given list of program windows</td>
</tr>
<tr>
<td>compile</td>
<td>compile the given program window, using the incremental compiler</td>
</tr>
<tr>
<td>compile_all</td>
<td>compile all program windows that have been modified since last being compiled, using the incremental compiler</td>
</tr>
<tr>
<td>optimize</td>
<td>compile the given program window, using the optimising compiler</td>
</tr>
<tr>
<td>optimize_all</td>
<td>compile all program windows that have been modified since last being compiled, using the optimising compiler</td>
</tr>
<tr>
<td>abolish</td>
<td>abolish programs that have been compiled or optimised from the given program window</td>
</tr>
<tr>
<td>abolish_all</td>
<td>abolish programs that have been compiled or optimised from all program windows</td>
</tr>
<tr>
<td>application</td>
<td>show the &quot;application&quot; dialog; allow the user to fill its fields and test or create an application</td>
</tr>
<tr>
<td>application(I,M,A,B,E,T,N)</td>
<td>test or create an application directory from the given field entries, where &quot;I&quot; is either the atom, &quot;test&quot;, or the term, &quot;save(File)&quot;, where &quot;File&quot; is an atomic file name and &quot;M&quot;, &quot;A&quot;, &quot;B&quot;, &quot;E&quot;, &quot;T&quot; and &quot;N&quot; are the names of the main, abort, break, error, timer and message hooks respectively</td>
</tr>
</tbody>
</table>

**The Options Menu: system_menu( W, options, F )**

The following table describes the functions available from the "options" menu; each of these is invoked by specifying the atom, "options", as the second argument in the call to system_menu/3:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace</td>
<td>toggle debugging mode between &quot;trace&quot; and &quot;off&quot; on successive calls</td>
</tr>
<tr>
<td>debug</td>
<td>toggle debugging mode between &quot;debug&quot; and &quot;off&quot; on successive calls</td>
</tr>
<tr>
<td>spypoints</td>
<td>show the &quot;spypoints&quot; dialog; allow the user to select and deselect predicates to be spied</td>
</tr>
<tr>
<td>prolog_flags</td>
<td>show the &quot;prolog_flags&quot; dialog; allow the user to choose preferences by adjusting these flags</td>
</tr>
<tr>
<td>prolog_flags(P1..P11)</td>
<td>set eleven preferences, which relate to a variety of prolog flags</td>
</tr>
<tr>
<td>font</td>
<td>show the &quot;font&quot; dialog; allow the user to choose a font for use in the console window and all program windows</td>
</tr>
<tr>
<td>font(Face,Size,Style)</td>
<td>create a font from the given face, size and style parameters and select it for use in the console window and all program windows</td>
</tr>
</tbody>
</table>
### Console History
- `console_history` show the "console history" dialog, allowing the user to set save setting and history depth
- `console_history(Sve,Dpt)` set the given save setting and depth for the console history

### Status Bars
- `status_bars` show the "status bars" dialog, allowing the user to set the mode for status bars
- `status_bars(Mode)` set the given mode for status bars

### Syntax Colouring
- `syntaxColouring` show the "syntax colouring" dialog, allowing the user to set mode and delay in milliseconds for rich syntax colouring
- `syntaxColouring(Md,Dly)` set the given mode and delay in milliseconds for rich syntax colouring

### Colour Settings
- `colour_settings` show the "colour settings" dialog, allowing the user to choose rich syntax colours for all windows of the same class as the given window

### Save Settings on Exit
- `save_settings_on_exit` toggle the "save_settings" Prolog flag between "on" and "off" on successive calls

### Upgrade
- `upgrade` show the "upgrade" dialog box, allowing the user to register a new key code

### Purchase
- `purchase` show the "purchase" dialog box, advising the user how to purchase a licensed copy

## The Window Menu

The "window" menu is handled directly by the WIN-PROLOG "kernel", and not the development environment, and cannot be accessed by `system_menu/3`.

## The Help Menu: `system_menu(W, help, F)`

The following table describes the functions available from the "help" menu; each of these is invoked by specifying the atom, "help", as the second argument in the call to `system_menu/3`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>win_prolog_help</td>
<td>show the WIN-PROLOG helpfile contents page</td>
</tr>
<tr>
<td>about_win_prolog</td>
<td>show the WIN-PROLOG about box</td>
</tr>
</tbody>
</table>
Appendix G - Errors, Names and Numbers

The **WIN-PROLOG** "kernel" performs extensive run-time error checking to ensure the integrity of applications, and this is echoed throughout the rest of the system. Errors can be handled in two primary ways in user applications: the '?ERROR?/2 hook predicate provides a mechanism for intercepting and possibly correcting errors, while catch/2 and throw/2 supplement this with a powerful "catch and throw" error handling model. Whichever approach is used for error handling, errors are reported using their "error number"; these are described in the following sections.

Success and Failure

When using catch/2 and throw/2, two "error" numbers are used to denote simple success or failure. These are not true errors, and will not be reported by the error hook, '?ERROR?/2: zero ("0") is returned by catch/2 when its goal succeeds, and minus one ("-1") is returned when its goal fails; throw/2 simply succeeds if called with an error number of zero ("0"), and fails when called with minus one ("-1"). All other numbers are treated as errors, as described below.

Keyboard Break Error

The keyboard break error occurs when the user presses the <ctrl-del> key combination. Note that this error bypasses the catch and throw error mechanism (see catch/2 and throw/2), as well as '?ERROR?/2: instead, it can enabled or disabled with the break/1 predicate, and handled by '?BREAK?/1.

<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Keyboard Break</td>
<td>This signals that the user has typed &lt;ctrl-del&gt; in order to interrupt program execution. As noted above, this error is routed to the '?BREAK?/1 hook, rather than either '?ERROR?/2 or catch/2, and can be toggled with break/1.</td>
</tr>
</tbody>
</table>

Memory Errors

Memory errors occur when one of **WIN-PROLOG**'s nine configurable memory areas becomes full during program execution. Often this will be as the result of a programming bug, but may be due to legitimate requirements (see Appendix H for a detailed discussion of memory management). The default settings for memory allocation can be changed using "command line switches", either on the **WIN-PROLOG** command line itself, or in its "initialisation" (.INI) file (see Appendix B for more information about initialisation files and switches). The two predicates, free/9 and total/9, can be used to enquire of memory usage, while xinit/9 can be used to perform runtime modifications. The gc/1 predicate can be used to force a garbage collection, although this is never required, because the garbage collector is called automatically whenever necessary.
<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Backtrack Stack Full</td>
<td>This occurs when the &quot;backtrack stack&quot; is full. A program has created too many choice points, perhaps because of a missing cut (I/O) (see Appendix H). This error can be &quot;caught&quot;, but not &quot;handled&quot;, in the latter case resulting in execution being aborted.</td>
</tr>
<tr>
<td>2</td>
<td>Local Stack Full</td>
<td>This occurs when the &quot;local stack&quot; is full. A program has performed a very deep, non-tail recursion, perhaps because of a trailing cut (I/O); it is almost always possible to rewrite such programs to require no local stack space (see Appendix H). This error can be &quot;caught&quot;, but not &quot;handled&quot;, in the latter case resulting in execution being aborted.</td>
</tr>
<tr>
<td>3</td>
<td>Reset Stack Full</td>
<td>This occurs when the &quot;reset stack&quot; is full. This is an unusual error, and is somewhat related to the &quot;Backtrack Stack Full&quot; error: a program has unified too many &quot;old&quot; variables, perhaps because of a missing cut (I/O) (see Appendix H). This error can be &quot;caught&quot;, but not &quot;handled&quot;, in the latter case resulting in execution being aborted.</td>
</tr>
<tr>
<td>4</td>
<td>Heap Space Full</td>
<td>This occurs when the &quot;term heap&quot; is full. A program has created too many lists or other compound data structures, or possibly a single data structure that is simply too large. After careful consideration (see Appendix H), it is worth restarting WIN-PROLOG with a larger term heap setting (/H - see Appendix B). This error can be &quot;caught&quot;, but not &quot;handled&quot;, in the latter case resulting in execution being aborted.</td>
</tr>
<tr>
<td>5</td>
<td>Text Space Full</td>
<td>This occurs when the &quot;text heap&quot; is full. A program has created too many atoms and strings, or too large a program has been loaded. After careful consideration (see Appendix H), it is worth restarting WIN-PROLOG with a larger text heap setting (/T - see Appendix B). Although this error can be &quot;caught&quot; or &quot;handled&quot;, care should be taken not to generate atoms or strings while handling it: since there is no free text space, such actions will result in another &quot;Text Space Full&quot; error, and a looping error condition.</td>
</tr>
<tr>
<td>Error</td>
<td>Text</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>6</td>
<td>Program Space Full</td>
<td>This occurs when the &quot;program heap&quot; is full. A program has asserted too many clauses, or too large a program has been loaded. After careful consideration (see Appendix H), it is worth restarting WIN-PROLOG with a larger program heap setting (&quot;/P&quot; - see Appendix B). Although this error can be &quot;caught&quot; or &quot;handled&quot;, care should be taken not to assert clauses or load further files while handling it: since there is no free program space, such actions will result in another &quot;Program Space Full&quot; error, and a looping error condition.</td>
</tr>
<tr>
<td>7</td>
<td>System Stack Full</td>
<td>This occurs when the &quot;system stack&quot; is full. A program has created a very deeply nested term, and possibly one which is self-referencing. Although usually the result of a programming bug (see Appendix H), it is sometimes worth restarting WIN-PROLOG with a larger system stack setting (&quot;/S&quot; - see Appendix B). This error can be &quot;caught&quot;, but not &quot;handled&quot;, in the latter case resulting in execution being aborted.</td>
</tr>
<tr>
<td>8</td>
<td>Input Space Full</td>
<td>This occurs when the &quot;string input buffer&quot; is full. A program has attempted to perform input from a string (see <code>&lt;~/2</code>) which is larger than the amount of space allocated for this purpose. Consider using memory files instead (see fcreate/5), or restart WIN-PROLOG with a larger input space setting (&quot;/I&quot; - see Appendix B). This error can be &quot;caught&quot; or &quot;handled&quot; safely.</td>
</tr>
<tr>
<td>9</td>
<td>Output Space Full</td>
<td>This occurs when the &quot;string output buffer&quot; is full. A program has attempted to perform output to a string (see <code>~/2</code>), but the amount of data written is larger than the amount of space allocated for this purpose. Consider using memory files instead (see fcreate/5), or restart WIN-PROLOG with a larger output space setting (&quot;/O&quot; - see Appendix B). This error can be &quot;caught&quot; or &quot;handled&quot; safely.</td>
</tr>
</tbody>
</table>

**Console I/O Errors**

The console I/O errors comprise a small group of exceptions that can occur during console (user) input and output, including the handling of windows, graphics, the mouse and the keyboard.
<table>
<thead>
<tr>
<th></th>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Window Handling Error</td>
<td>This is something of a &quot;catch all&quot;, being the response to virtually any Windows API failure (see <code>winapi/4</code>). It simply means, &quot;this Windows function did not work&quot;: the usual response is to examine the errant call, correct it, and try again.</td>
</tr>
<tr>
<td>11</td>
<td>Graphics Handling Error</td>
<td>Like the &quot;Window Handling Error&quot;, this is something of a &quot;catch all&quot;: in this case, it simply means, &quot;this GrafXiX function did not work&quot;: the usual response is to examine the errant call, correct it, and try again.</td>
</tr>
<tr>
<td>12</td>
<td>Mouse Handling Error</td>
<td>This error is not currently used in WIN-PROLOG.</td>
</tr>
<tr>
<td>13</td>
<td>Console Buffer Full</td>
<td>A special &quot;console buffer&quot; is maintained internally by WIN-PROLOG to double-buffer text entered at the &quot;console window&quot; (user device): this is required in order to allow term input to be interrupted by window messages and timers (see <code>window_handler/2</code> and <code>timer_create/2</code> respectively), even though some lines of text have been entered. This buffer effectively places a 4kb limit on the size of any one term entered at the console, although it has no effect on disk file, memory file or string input. This error signals that this buffer has been exceeded: the usual response is to place the desired input into a file and read it from there.</td>
</tr>
<tr>
<td>14</td>
<td>Timer Handling Error</td>
<td>This error occurs when an attempt is made to reprogram a timer which is already active, for example by calling the <code>timer_set/2</code> predicate twice in succession with the same first argument and non-zero second argument: timers can only be cancelled by calling <code>timer_get/2</code> with a variable second argument (see <code>timer_create/2</code> for more information about timers).</td>
</tr>
<tr>
<td>15</td>
<td>Socket Handling Error</td>
<td>An error has occurred during a Windows Sockets (Winsock) predicate, perhaps because a reference has been made to a nonexistent socket, or because an inappropriate use of an existing socket has been attempted (see Appendix P for more information about sockets).</td>
</tr>
<tr>
<td>16</td>
<td>Midi Handling Error</td>
<td>An error has occurred during a Musical Instrument Digital Interface (MIDI) predicate, perhaps because a reference has been made to a nonexistent MIDI device, or because an inappropriate use of an existing MIDI device has been attempted (see Appendix Q for more information about MIDI).</td>
</tr>
</tbody>
</table>
### Predicate Errors

The predicate errors describe a collection of conditions directly related to the process of calling a predicate, either because the predicate itself does not exist, or because one or more of its arguments, or even the entire goal, is incorrect. With the exception of a specific type of "Control Error", all these errors can be "caught" or "handled" safely.

<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Predicate Not Defined</td>
<td>This occurs when a completely nonexistent predicate is called. One of the Prolog flags (&quot;unknown&quot;: see <code>prolog_flag/3</code>) can be used to replace instances of this predicate by failure, unless it occurs within a call to <code>catch/2</code>: in this instance, the value &quot;20&quot; will still be returned, and not &quot;-1&quot; as might be expected. Predicates declared dynamic (see <code>dynamic/1</code>), whether or not they contain clauses, technically &quot;exist&quot;, and simply fail if called while they are empty.</td>
</tr>
<tr>
<td>21</td>
<td>Control Error</td>
<td>This occurs when a &quot;call&quot; is made to an invalid structure, such as an integer or a variable (see <code>callable/1</code>). It most often results when a &quot;metacall&quot; in a program has not been bound by the time it is called. Depending upon the data type involved, this error may not be passable to the <code>?ERROR?/2</code> hook: it is, however, always &quot;catchable&quot;.</td>
</tr>
<tr>
<td>22</td>
<td>Instantiation Error</td>
<td>This is similar to the &quot;Control Error&quot;, but occurs when a specific argument is or contains an unbound variable where some ground data type was expected. The most common cause is a mis-spelled variable name.</td>
</tr>
<tr>
<td>23</td>
<td>Type Error</td>
<td>This is similar to the &quot;Instantiation Error&quot;, but occurs when a specific argument is or contains a data type which is different from the data type that was expected: the usual response is to examine the errant call, correct it, and try again.</td>
</tr>
<tr>
<td>24</td>
<td>Domain Error</td>
<td>This is similar to the &quot;Type Error&quot;, but occurs when a specific argument is or contains the correct data type, but a value which is out of the expected range: the usual response is to examine the errant call, correct it, and try again.</td>
</tr>
<tr>
<td>25</td>
<td>Too Many Arguments</td>
<td>An attempt has been made to define or call a predicate with more than the permitted number of arguments, currently 128 (see <code>arity/1</code>). The simplest response is to group sets of arguments together as lists, tuples or other compound data structures, and try again.</td>
</tr>
</tbody>
</table>
Term Too Big

This is almost always a response to a program bug in which an "infinite" (self-referencing) term has been created by mistake. Unlike the "Machine Stack Full" error, which normally results from creating an infinitely deep term, this error normally reflects an infinitely long term. Because long terms require no stack space for analysis, it is always possible to "catch" or "handle" the present error.

File Handling Errors

These errors report a handful of common exceptions reported by Windows when problems occur handling or performing input and output to and from files. Some of the more esoteric errors are reported as "Unknown Errors" (see below). All these errors can be "caught" or "handled" safely.

<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>File Handling Error</td>
<td>This is something of a &quot;catch all&quot;, being the response to a variety of file exceptions. Most frequently, it indicates an attempt to switch input from or output to (see input/1 and output/1 respectively) a file which has not been opened appropriately (see fcreate/5). Correct the program and try again.</td>
</tr>
<tr>
<td>31</td>
<td>File Not Found</td>
<td>A file required by a predicate does not exist. Check the errant call, as well as the current working directory and its contents, and make appropriate changes.</td>
</tr>
<tr>
<td>32</td>
<td>Path Not Found</td>
<td>This is very similar to &quot;File Not Found&quot;, the only difference being that part of a path required by a predicate does not exist. Check the errant call, as well as the current working directory and its contents.</td>
</tr>
<tr>
<td>33</td>
<td>Too Many Files Open</td>
<td>Although WIN-PROLOG allows up to 64 files to be open at any one time, this might be more than is supported by a given installation of Windows. In any event, an attempt has been made to open one too many files simultaneously: the simple response is to close one or more dormant files and try again.</td>
</tr>
</tbody>
</table>
File Access Denied

An attempt has been made to open, rename or delete a file for which the appropriate permission has not been granted (see `fcreate/5`, `ren/2` and `del/1`). It can also result from trying to write to a file that has been opened in read-only mode or vice versa: check file permissions, other processes that might be accessing the file, and input/output calls accordingly, and try again.

Disk Full

Whoops: this probably indicates that Microsoft Office and Internet Explorer have been installed on the hard disk! An attempt to write to a file has failed through lack of disk space. This is also signalled when trying to write beyond the end of a memory file (see `fcreate/5`). Solutions include uninstalling unwanted software, deleting old backup and temporary files, or buying a bigger hard disk.

Memory Full

A combination of WIN-PROLOG and all other running processes have managed to exceed the amount of real and virtual memory available. Solutions include trying to run Prolog with smaller memory settings (see Appendix H), or closing down one or more other applications before trying again.

### Term I/O Errors

These errors typically result from problems during input and output, other than those caused by operating system and hardware exceptions (see above). All these errors can be "caught" or "handled" safely.

<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Format Not Defined</td>
<td>An invalid format letter has been specified as the first argument in a call to one of the formatted input or output predicates (see <code>fread/4</code> and <code>fwrite/4</code> respectively). Correct the call, and try again.</td>
</tr>
<tr>
<td>41</td>
<td>Format Field Overflow</td>
<td>A data item has exceeded the specified field width in a call to the formatted input predicate (see <code>fread/4</code>). Solutions include increasing the field width, or specifying a truncated field: correct the call, and try again.</td>
</tr>
<tr>
<td>Code</td>
<td>Error Type</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>42</td>
<td>Syntax Error</td>
<td>An attempt has been made to read a Prolog term which contains one or more syntax errors. In keeping with Quintus Prolog, the <code>read/1</code> predicate attempts to handle this particular error &quot;intelligently&quot;, depending upon the setting of the Prolog flag, &quot;syntax_errors&quot; (see <code>prolog_flag/3</code>). The <code>eread/1</code> or <code>eread/2</code> report all syntax errors, irrespective of the setting of Prolog flags. The solution is to correct the errant term, and try again.</td>
</tr>
<tr>
<td>43</td>
<td>End Of File</td>
<td>The end of a file has been encountered while reading a term or other data item; this error is also signalled when trying to read at the end of a file after a previous read has indicated the end of file through simple failure or equivalent signal. If the file was not exhausted prior to the input (see <code>eof/0</code>), this usually indicates an error in the file (an incomplete term, for example); otherwise, it suggests an error in the program which was performing input. Correct the file and/or the program, and try again.</td>
</tr>
<tr>
<td>44</td>
<td>Binary Format Error</td>
<td>An attempt to process one of the WIN-PROLOG binary file or stream formats (encrypted, compressed or object) has encountered an error (see <code>decode/2, fluff/3 and load_files/1</code>). Check that the file is uncorrupted and of the appropriate type, and try again.</td>
</tr>
<tr>
<td>45</td>
<td>Checksum Error</td>
<td>An attempt to process an encrypted or compressed file has encountered an inconsistency in the data (see <code>decode/2</code> and <code>fluff/3</code>). In the former case, check the password that was used; otherwise, check that the file is uncorrupted and of the appropriate type, and try again.</td>
</tr>
<tr>
<td>46</td>
<td>String Too Long</td>
<td>There is effectively no limit on the length of strings in WIN-PROLOG; however, some operations require that their text is placed in a temporary buffer. An example of this is the &quot;immediate&quot; string arguments offered to <code>winapi/4</code>. This error occurs when too much data is passed to such predicates: consider using memory files when handling large amounts of text (see <code>fcreate/5</code>).</td>
</tr>
<tr>
<td>47</td>
<td>Atom Too Long</td>
<td>Atoms are limited to a maximum length of 8192 characters in WIN-PROLOG; this error can occur either when reading terms which contain longer atoms, or when converting strings to atoms (see <code>stratm/2</code>). Correct the source file or program, and try again.</td>
</tr>
</tbody>
</table>
Character Error

An attempt has been made to read or write a character that is not valid for the current input or output stream, such as when to send the Unicode Trademark symbol (character 2122h) to a file currently open in ISO mode (see fcreate/5).

Arithmetic Handling Errors

This group of errors is largely confined to the arithmetic handling predicates, including is/2, fpn/2 and rpn/2, as well as the group of predicates that compares expressions. All these errors can be “caught” or “handled” safely.

<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Function Not Defined</td>
<td>An attempt has been made to evaluate an arithmetic expression which contains a nonexistent function (see is/2 for a list of supported floating point functions): locate the errant function, correct the call and try again.</td>
</tr>
<tr>
<td>51</td>
<td>Arithmetic Underflow</td>
<td>An attempt has been made to evaluate an arithmetic expression to a non-zero value smaller than about +/- 2.2e-308: correct the call and try again.</td>
</tr>
<tr>
<td>52</td>
<td>Arithmetic Overflow</td>
<td>An attempt has been made to evaluate an arithmetic expression to a value larger than about +/- 1.8e308; this error can also occur in several input predicates when attempting to read numbers: correct the call or input data and try again.</td>
</tr>
<tr>
<td>53</td>
<td>Arithmetic Error</td>
<td>An attempt has been made to evaluate an arithmetic expression which contains an invalid data for a function (for example, trying to compute the arcsine of a number greater than +/- 1.0): correct the call and try again.</td>
</tr>
</tbody>
</table>

Assembly and Compilation Errors

This group consists of errors relating to the compiling and decompiling of programs. Several of them should not occur under normal circumstances, being directly related to errors in output from the optimising compiler. All these errors can be “caught” or “handled” safely.

<table>
<thead>
<tr>
<th>Error</th>
<th>Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>60</td>
<td>Instruction Not Defined</td>
<td>An optimised code sequence contains an invalid Sigma2 instruction: this is a bug which should</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not occur in the release system, and should be reported to technical support.</td>
</tr>
<tr>
<td>61</td>
<td>Bad Number Of Arguments</td>
<td>An optimised code sequence contains a Sigma2 instruction with the wrong number of arguments:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this is a bug which should not occur in the release system, and should be reported to technical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>support.</td>
</tr>
<tr>
<td>62</td>
<td>Bad Argument Type</td>
<td>An optimised code sequence contains a Sigma2 instruction with the wrong type of argument:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this is a bug which should not occur in the release system, and should be reported to technical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>support.</td>
</tr>
<tr>
<td>63</td>
<td>Bad Register Number</td>
<td>An optimised code sequence contains a Sigma2 instruction with an invalid register number:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this is a bug which should not occur in the release system, and should be reported to technical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>support.</td>
</tr>
<tr>
<td>64</td>
<td>Label Not Defined</td>
<td>An optimised code sequence contains a Sigma2 instruction which references a nonexistent label:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this is a bug which should not occur in the release system, and should be reported to technical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>support.</td>
</tr>
<tr>
<td>65</td>
<td>Label Already Defined</td>
<td>An optimised code sequence contains more than one instance of a given label: this is a bug</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which should not occur in the release system, and should be reported to technical support.</td>
</tr>
<tr>
<td>66</td>
<td>Bad Assembler Module</td>
<td>This error is not currently used in WIN-PROLOG.</td>
</tr>
<tr>
<td>67</td>
<td>Predicate Protected</td>
<td>An attempt has been made to redefine, abolish or otherwise modify a built-in predicate, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to assert to or retract from a &quot;static&quot; predicate (see dynamic/1). In the former case, check</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the names of any predicates in files being loaded, as well as calls to abolish/1 and other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>program manipulation predicates; in the latter case, make sure all predicates that are being</td>
</tr>
<tr>
<td></td>
<td></td>
<td>asserted to and retracted from have been declared &quot;dynamic&quot; (see assert/1, retract/1 and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dynamic/1)</td>
</tr>
</tbody>
</table>

**DOS Errors and Unknown Errors**

All error numbers other than those listed in the above sections have the same name, "Unknown Error". Mostly the errors can only be generated by throw/2; however, a special subset is known as the "DOS Errors": if a file or other operating system call generates an error that cannot be classified into one of the "File Errors", a value of 1000 is added to the error code and this is reported as an "Unknown Error". Check Windows documentation to determine the cause (remembering to subtract 1000 from the reported number), correct the program and try again. All these errors can be "caught" or "handled" safely.
Appendix H - Prolog Memory Management

One of the black arts of Prolog programming is the management of memory. By its very nature, Prolog hides the low-level allocation and deallocation of memory from the user: this is a feature to be expected of such a high-level language; on the other hand, it can be a cause of confusion and program inefficiency if not reasonably well understood. This appendix attempts to shed light on the matter in a practical way, while touching on some theoretical aspects of this complex subject.

Memory: Stacks, Heaps and Buffers

The WIN-PROLOG system provides sophisticated, automatic memory management, with the added flexibility that the user can change default memory settings when required for any particular application. One of the problems with this, however, is that it is not always apparent what such settings should be used!

Three types of memory area are used by WIN-PROLOG, and are generally known as "stacks", "heaps" and "buffers". We will look at these in more detail in the following sections.

Memory Stacks

Stacks are used to handle the flow of control during program execution, and are common to all programming languages which support functions, procedures or subroutines. In C, for example, the function call:

```c
foo(a);
```

uses a stack, first to save the parameter "a", and then to save the address of the next thing to do after calling "foo" (the "return address"). Then it jumps to the code for "foo" itself. Upon completion, "foo" picks up the return address from the stack, and jumps to that location to continue processing; the "a" parameter is also "popped" off the stack to restore its status to that immediately before the function call.

Prolog's execution model is more complex than that of C or most other languages; in particular, there are effectively two return addresses for each "function" (or rather, "predicate") call: where to continue after success and where to continue after failure. This suggests that at least two independent stacks must exist, and this is indeed the case: the "local" stack is used to store "on success" addresses and associated data, while the "backtrack" stack is used to store "on failure" addresses and associated data. For example, consider the program:

```prolog
foo :-
    bar,
    sux.
```
foo.

At the point at which "bar" is being called by "foo", there are two things that might need to be done subsequently. If "bar" succeeds, the next thing that needs to be called is "sux". If it fails, the next thing to be called is the second clause of "foo". The address of "sux" is therefore stored on the local stack, and that of the second clause of "foo" on the backtrack stack.

But even this is not quite sufficient. As well as knowing where to go on failure, Prolog must also know how to "undo" any variable bindings that were made before a given predicate call failed. The information needed to do this is stored on yet another stack, the "reset" stack. Entries on this stack do not occur every time a variable is bound, because many variables simply cease to exist on failure. Again, consider the program:

```prolog
foo :-
    X = 123,
    bar(X).

foo.
```

Should "bar(X)" fail, the whole first clause of "foo" will fail, and the variable "X", which only occurs in that clause, will vanish. As such, there is no need for Prolog to create a "reset" entry for it. However, consider a variation of the above program:

```prolog
foo :-
    ( X = 123
      ; X = 456
    ),
    bar(X).

foo.
```

This time, if "bar(X)" were to fail, an attempt is going to be made to bind "X" to "456" before trying "bar(X)" again. Patently, if "X" still had the value "123", any attempt to say "123 = 456" would fail at once, so Prolog has to convert "123" back to the variable "X" before continuing. It prepares for this eventuality by creating an entry on the reset stack for the variable "X" before binding it to "123". Notice that if "bar(X)" does fail, and "X" is reset, it is not necessary to create a new reset entry before binding the variable to "456"; if "bar(X)" were to fail a second time, the whole first clause for "foo" would fail, and "X" would vanish.

In brief, "local" stack entries are used to record where to continue in the current clause after a successful predicate call, "backtrack" stack entries are used to record where to continue earlier in the current clause, or elsewhere in the current predicate, after a failed predicate call, and "reset" entries are used to
store the previous values of any variables bound, if those variables will still be visible after a failed predicate call. All three of these stacks, together with a fourth one (the "system" stack), will be discussed in greater depth below.

Memory Heaps

Unlike stacks, which are ordered memory areas onto which data and addresses can be "pushed", and later "popped", a "heap" is a disordered repository of memory units, called "cells". In WIN-PROLOG, heaps are used to store the individual elements of compound terms and lists, as well as strings and atoms, and even segments of compiled Prolog code.

Stacks do not need special memory management, just so long as pushes and pops are performed symmetrically; heaps, on the other hand, require additional handling if disused memory is to be recycled. Consider the program:

```
foo :-
    bar( X ),
    foo.

bar( [1,2,3,4,5,6,7,8,9,0] ).
```

Each time "foo" calls "bar(X)", the latter program creates a ten-element list structure by allocating and linking ten cells from the data heap. A recursive call is then made to "foo", requiring no use of any stacks because there is no "next thing to do" either for success or failure. Moreover, the list that has been created can no longer be seen, because neither was it passed out of "foo" as a parameter, no is there any way back into "foo" through failure such that it might need to be seen again. The ten cells of heap memory that the list occupied is simply severed from the computation, becoming ten cells of "garbage".

Each new recursive call uses up a further ten cells of heap space, and eventually the entire heap will be full of garbage. At this point, a special internal routine, the "garbage collector", is invoked. It works by checking which bits of the heap are still in use, and then collecting up all the remaining cells into a new allocation list. The garbage collector is called automatically whenever the heap fills with garbage, and is extremely quick, so in effect the heap behaves as an endless source of memory cells. It is, of course, possible to exhaust this "endless" supply, simply by creating a massive data structure; for example, consider:

```
foo( List ) :-
    bar( X ),
    foo( [List|X] ).

bar( [1,2,3,4,5,6,7,8,9,0] ).
```

This variation of the program not only uses ten cells per iteration for the list returned by "bar(X)", but also an eleventh of linking to the existing passed-in
list. The main point, however, is that the data is not "lost" on the recursive call, and so does not become garbage, cannot be collected, and will quickly fill the heap to capacity.

The space used to store strings and atoms, the "text" heap, is handled in exactly the same way as data heap, except that the cells are somewhat larger and designed for storing text rather than Prolog data. The third, or "program" heap, is used to store the clauses and predicates of compiled Prolog code. While still a true heap, with specially designed cells, it does not need to be garbage collected in the conventional sense. The reason is that its data items (pieces of Prolog code) are persistent, and have to be explicitly removed by programs (using retract/1, abolish/1, etc). Whenever these predicates delete code, they explicitly return the freed memory to the program heap.

**Memory Buffers**

Only a few words need to be said about buffers. These are simply flat memory blocks used for special purposes. WIN-PROLOG features a powerful "string" data type, which can be used among other things as an input or output "device". For example, the call:

```
?- listing ~> String.
```

creates a string called "String" containing the text of a program listing. Two buffers, the "input" and "output" buffers, are used to support this feature. The above example requires output buffer space, while:

```
?- read( Term ) <~ `foo(X). `.
```

reads from a string, requiring input space. WIN-PROLOG additionally supports memory files, which can be created and deleted at will, and can be used both as input/output streams and for interfacing to the Windows API (see fcreate/5, winapi/4 and wintxt/4).

**Controlling Stack Usage**

In virtually all cases, the default size settings for the four stacks enable programs to run comfortably, and with ample "spare" space. It is almost always true that "Backtrack Stack Full", "Local Stack Full", "Reset Stack Full" and "System Stack Full" errors are due to a bug in an application's code, and not because the program concerned genuinely requires more resources. The next few sections show some examples of how to fill - or avoid filling - each of the four stacks.

**The Backtrack Stack**

As noted above, this stack is used to store "on failure" addresses introduced by choice points in an evaluation. The simplest way to fill it up, whatever its size, is to write a programs such as:
foo :-
    foo.

foo.

bar :-
    sux,
    bar.

sux.

sux.

On entry to the first clause for "foo", a backtrack stack entry is made pointing to the second clause, because this would need to be called in the event of failure. In the case of "bar", there is just one clause, but a backtrack entry is made at the point "sux" is called, because this has two clauses.

The only way to avoid filling the backtrack stack is to ensure that, at the point of recursion, there are no choice points left in the current clause or relation. The following programs do not use up space on the backtrack stack:

foo :-
    foo.

bar :-
    sux,
    bar.

sux.

The "foo" case only has one clause, so no choice points are introduced, and the backtrack stack is not used; similarly, the "bar" case calls a version of "sux" with only one clause, and again, no backtrack entries are needed.

It is possible to force "determinism" on programs which otherwise would have choice points, by using the "cut" predicate, \(/\). This has the effect of removing backtrack stack entries for any choice points earlier in the current clause and elsewhere in the current predicate. The following programs run forever without filling the backtrack stack:

foo :-
The cut in "foo" removes the backtrack stack entry pointing to the second clause, while the one in "bar" removes the choice point introduced by the call to "sux". Similarly, the following program will run indefinitely:

```prolog
bar :-
    sux,
    !,
    bar.

sux.
sux.
```

In this case, the "sux" program is itself deterministic thanks to its use of cut.

Another feature of **WIN-PROLOG** which helps create determinism is "first argument indexing". This enables the execution mechanism to "ignore" clauses whose first arguments do not match the type, or in many cases, the value of a call. This subject is discussed at length elsewhere, but for present purposes, consider the following programs:

```prolog
foo :-
    sux( 1 ),
    foo.

bar :-
```
The "foo" program calls "sux(1)", specifying the first argument: because there are no other clauses of "sux(...)" which can match the type or value "1", no choice points are created, and the program will run forever without using the backtrack stack. The "bar" program, however, calls "sux(.)" with a variable, which means that it cannot use first argument indexing to isolate just one of the clauses of "sux(...)", so choice points are created on the backtrack stack. Now consider:

```prolog
foo :-
    sux( 1 ),
    foo.

sux( 1 ).
sux( _ ).
```

Here, even though the call in "foo" is to "sux(1)", because the second clause for "sux(...)" contains a variable, which could also match the value "1", choice points are once again needed and the backtrack stack will be quickly used up. Finally, consider:

```prolog
foo :-
    sux( 1, _ ),
    foo.

bar :-
    sux( _, 1 ),
    bar.

sux( 1, 1 ).
sux( 2, 2 ).
```

Even though both "foo" and "bar" specify one value of "1" and one variable ("_") in their respective calls to "sux(...)", only "foo" is deterministic; "bar" creates
choice points and requires backtrack stack space. The reason is that "foo" specifies the first argument as "1", in the call "sux(1,_)", while "bar" specifies the second argument, in the call "sux(_,1)". As its name implies, first argument indexing only works when the first argument is specified!

The Local Stack

As already described, this stack is used to store "on success" addresses. In theory, every predicate call needs to know where to continue upon successful completion, but in practice, the final predicate call in any relation saves stack space by not creating an entry: upon completion of such a call, the next thing to do is whatever the "parent" program is supposed to do next. Consider the program:

```prolog
foo :-
    bar,
    sux.

bar :-
    you,
    too.
```

When "foo" calls "bar", a local stack entry is made pointing at "sux". Now "bar" calls "you", and another local stack entry is made, this time pointing at "too". When "you" succeeds (it might also have made local stack entries, as might any programs it called), the topmost entry is popped from the local stack, telling Prolog to continue execution at "too". This time, because "too" is the last call, no local stack entry needs to be made. Upon success, the topmost local stack entry is popped, which is our original one, pointing at "sux", so this is where execution continues.

It follows that a "tail recursive" program, whereby the recursive call is the final one in the clause, does not use local stack:

```prolog
foo :-
    bar,
    foo.
```

If the recursive call is not the last call, we have a non-tail recursive (or "head recursive") program, such as:

```prolog
foo :-
    foo,
    bar.
```

Such a program will use up local stack very quickly. Unlike the backtrack stack, it is not possible to use "cut" to save local stack space:
foo :-
    !,
    foo,
    bar.

requires just as much local space as the previous example. The reason is that cut is designed for controlling backtracking on failure, and has no power to control success. Note that the cut itself is treated as a predicate call, and a common mistake is to place one at the end of every clause in a program; this converts tail recursion into head recursion:

foo :-
    bar,
    foo,
    !.

will use up the local stack, because Prolog must store the call to "!" as the next thing to do after calling "foo".

The Reset Stack

This stack is used to store variables that need resetting during failure. Only variables "older" than the current choice point, as stored on the backtrack stack, need resets to be created during binding. This is because "newer" ones simply vanish on failure. Consider the program:

foo :-
    X = 123,
    bar( X, Y ),
    Y = 456.

bar( X, Y ).

bar( X, Y ).

Both variables, "X" and "Y", are "declared" at the point "foo" begins to execute. They have the same "age" as whichever choice point (backtrack stack entry) is most recent. As "X" is bound to "123", a test is made to see if the variable is older than the current choice point: it is not, so no reset needs to be made. A call is then made to "bar(X,Y)", which has two clauses, and therefore introduces a choice point, with its entry on the backtrack stack. Next, "Y" is to be bound to "456", and a test is once again made on the variable's age. This time, because a new choice point has been introduced since the variable was declared, a reset stack entry must be made before the variable can be bound.
If a failure subsequently occurs, the most recent choice point is popped off the backtrack stack, returning control to the second clause for \( \text{bar}(X,Y) \). Before execution proceeds, however, the variable \( Y \) must be reset to its unbound state, by popping the reset stack entry. This way, \( \text{bar}(X,Y) \) will see, as expected, \( X \) still bound to the value \( "123" \), and \( Y \) as an unbound variable. Because the backtrack stack has been popped, and there are no further choice points in \( \text{bar}(X,Y) \). when control returns to \( \text{foo} \), and \( Y \) is once again to be bound to \( "456" \), the age test will now show that \( Y \) is the same age as the current choice point, and a reset will this time not need to be made. This is because should a second failure occur, both \( X \) and \( Y \) will cease to exist entirely, because \( \text{foo} \) will have failed, and there will have been no point in trying to reset these variables.

In practise, the reset stack can be thought of as vaguely linked to the backtrack stack. If the latter does not grow, then neither will the former; however, it is quite possible to grow the latter without the former.

The System Stack

A few brief words should be said about the system stack. This is the direct equivalent of the stack used by \( \text{"C"} \), and is used internally by \text{WIN-PROLOG}'s engine to recurse into structures and call functions. The only time that it is ever an issue is when excessively deep Prolog structures are passed to built-in predicates. For example, the call:

\[
?- X = [X], assert( \text{foo}(X) ).
\]

creates an infinitely deep list structure, and then tries to assert it. In attempting to compile this structure, \( \text{assert/1} \) runs out of system stack space. The other time system stack space can be seen to run out is during term output: when items being written are too deep, the output routine "cheats" by writing three dots and skipping to the tail. Thus the call:

\[
?- X = [X].
\]

will not generate an error, but will output something like:

\[
X = [[[[[[[[[[[[[...]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
\]

Several thousand brackets, rather than just sixteen, normally print before the ellipsis.

Setting Stack Sizes

As mentioned above, the default sizes of the four stacks, at 64kb each, are virtually always sufficient for any non-buggy Prolog program. As a rough guide, 64kb on the backtrack stack allows about 1400 choice points to be active at any one time; 64kb on the local stack allows about 2100 head recursions before filling up. In the former case, any program with 1400 outstanding choice points is going to take a vast amount of time to complete: each choice point represents at the very least a binary choice, meaning this program is non-deterministically searching a solution space of \( 2^1400 \) or greater com-
plexity! It is essential to add cuts where necessary to cut down the search space to manageable proportions, and in the process, free up the backtrack (and reset) stacks.

As for the latter case, it is always possible to write programs to avoid incrementally creating local stack entries. For example, consider the "traditional" implementation of the factorial function:

```
fact( 0, 1 ).

fact(Number, Fact) :-
    Less is Number - 1,
    fact(Less, Partial),
    Fact is Number * Partial.
```

While perfectly "correct", this implementation requires local stack space because the recursive call to "fact(Less,Partial)" is followed by a call to `is/2`. An alternative implementation uses an additional argument to allow the last two calls to be reversed:

```
fact(Number, Fact) :-
    fact2(Number, 1, Fact).

fact2(0, Fact, Fact).

fact2(Number, Sofar, Fact) :-
    Less is Number - 1,
    Next is Number * Sofar,
    fact2(Less, Next, Fact).
```

The trick here is to pass an initial value ("1") into the program at the start, and to perform the computation on this, simply echoing back the final result. Because the recursive call to "fact2(Less,Next,Fact)" is the last call in the second clause for "fact2", no local stack is used.

It is tempting to make sure that "fact" (or "fact2") is deterministic, by adding a cut. This will prevent endless backtracking after failure, but it is essential to place the cut correctly. In the above program, you could write:

```
fact(Number, Fact) :-
    fact2(Number, 1, Fact).

fact2(0, Fact, Fact) :-
```

```
fact2( Number, Sofar, Fact ) :-
    Less is Number - 1,
    Next is Number * Sofar,
    fact2( Less, Next, Fact ).

This would avoid backtracking by ensuring that once the "0" end case was found, no alternatives would be tried. It is not helpful in this case to place a cut in the second clause of "fact2":

fact( Number, Fact ) :-
    fact2( Number, 1, Fact ).

fact2( 0, Fact, Fact ).

fact2( Number, Sofar, Fact ) :-
    Less is Number - 1,
    Next is Number * Sofar,
    !,
    fact2( Less, Next, Fact ).

There are no non-deterministic calls in this clause, and no subsequent clauses, so this use of cut is a waste of time (but no worse than that). It is, however, distinctly wrong to write:

fact( Number, Fact ) :-
    fact2( Number, 1, Fact ).

fact2( 0, Fact, Fact ).

fact2( Number, Sofar, Fact ) :-
    Less is Number - 1,
    Next is Number * Sofar,
    !,
    fact2( Less, Next, Fact ),
    !.

Not only does this do nothing to help avoid backtracking, but it also makes the recursive call to "fact2(LESS,NEXT,FACT)" head recursive, once again using
one local stack frame per iteration.

Ironically, factorial is a bad example for demonstrating local stack usage, because it will cause an "Arithmetic Overflow" error for any values of "Number" greater than 170, and so cannot get near to using up the 2100 or so available stack frames! Other programs however, such as those which add up list elements, can benefit from rewriting with the extra argument:

```prolog
total( [], 0 ).

total( [Head|Tail], Total ) :-
    total( Tail, Rest ),
    Total is Rest + Head.
```

adds up a list such as "[1,2,3]" to return the value "6", but needs one local stack frame for each list element, limiting it to lists of less than about 2100 entries. The same job can be attained without any such limits by adding the extra argument:

```prolog
total( List, Total ) :-
    total2( List, 0, Total ).

total2( [], Total, Total ).

total2( [Head|Tail], Sofar, Total ) :-
    Next is Head + Sofar,
    total2( Tail, Next, Total ).
```

This program runs without using local stack space, and its only limit is the size of list that can be squeezed into the current heap. Other examples include "naive reverse", which is traditionally written:

```prolog
reverse( [], [] ).

reverse( [Head|Tail], List ) :-
    reverse( Tail, Partial ),
    append( Partial, [Head], List ).
```

As well as being inefficient because of the use of append/3, this is not tail recursive, and therefore uses local stack. The three-argument version of "reverse" is preferred:
reverse( Data, List ) :-
  reverse2( Data, [], List ).

reverse2( [], List, List ).
reverse2( [Head|Tail], Sofar, List ) :-
  reverse2( Tail, [Head|Sofar], List ).

Here, the extra argument is initialised with an empty list ("[]"); this program is not only tail recursive, requiring no local stack space, but is also linear in performance with respect to the length of the input list, compared with "naive reverse", whose performance relates to the square of the length of the input list. Cuts are completely unnecessary here, thanks to first argument indexing which can distinguish the two cases of "reverse2". Please note that both these examples are for illustration only: reverse/2 is built into WIN-PROLOG (using the second, efficient form, of course!).

The summary of this section is that, other than in completely exceptional circumstances, it is never necessary to increase the sizes of the backtrack, local, reset or system stacks from their default settings of 64kb each.

### Setting Heap Sizes

Unlike the stacks, which should never usually overflow except when application programs contain bugs, heaps are used to store real data that can legitimately grow in size. The main data heap, or just plain "heap", has a default size of 256kb. This is sufficient to hold a single list with about 26100 elements, or ten each with 2610 elements, and so forth. In other words, it can hold a lot of data. While most programs never even get close to requiring this much dynamic term storage, there are valid exceptions. In natural language programs, compilers, chess algorithms and similar, parse trees can be of a considerable size; while portions of these are passed around, extended, manipulated and so on, the overall data burden can grow rapidly. Similarly, predicates such as findall/3, setof/3 and bagof/3, when applied to queries with large solution sets, can use up considerable quantities of heap space.

If a "Heap Space Full" error occurs, the first question that should be considered is, "does this application really need over twenty-six thousand list cells in order to operate?" The answer might be "yes", in which case WIN-PROLOG should be run with a higher heap setting, but often the answer will be "no", and the search for "heap traps" should commence. Consider the program:

```prolog
foo :-
  bar( X ),
  sux( X ),
  foo.

bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).

sux( X ).
```
At each iteration of "foo", a call is made to "bar(X)", which generates a list of twenty elements, requiring twenty cells, or two hundred bytes, or heap space. A call is then made to "sux(X)", which here simply ignores the data. As the recursive call is made to "foo", all references to the twenty-element list are lost, since there are no choice points, and the list is not being passed as a parameter. The list therefore becomes garbage, and can be collected and recycled in due course as needed. Now consider the program:

```prolog
foo :-
    bar( X ),
    sux( X ),
    foo.
```

```prolog
bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).
```

```prolog
sux( X ).
```

```prolog
sux( X ).
```

The only difference is an extra clause for "sux(X)". As we saw earlier, this means that "foo" now uses up backtrack stack entries, as well as local stack entries. But it also uses up heap cells, twenty at a time, because even though the list is not passed as a parameter to the recursive call to "foo", it needs to be kept in case of a subsequent failure, which will cause backtracking into the second clause of "sux(X)". Here is another case where a simple placement of a cut saves on memory usage, including both the above-mentioned stacks and the heap:

```prolog
foo :-
    bar( X ),
    sux( X ),
    !,
    foo.
```

```prolog
bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).
```

```prolog
sux( X ).
```

```prolog
sux( X ).
```

Here, the cut before the recursive call to "foo" removes the backtrack stack entry pointing to the second clause of "sux(X)". Alternatively, the cut could be placed in the first clause of "sux(X)".
foo :-
  bar( X ),
  sux( X ),
  foo.

bar( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20] ).

sux( X ) :-
  !.

sux( X ).

This time, "sux(X)" itself prevents backtracking to subsequent clauses. Placing a cut in the clause for "bar([...])" would have no effect, and neither would a cut in the second clause for "sux(X)". Placing one after the recursive call to "foo" would both fail to help backtracking, and require use of the local stack by making this call head recursive.

The text heap is used to store atoms and strings. In the former case, each atom's text is only stored once, and all subsequent occurrences of the atom refer to this same piece of stored text. The first eight characters of a new atom require two cells, or 32 bytes of storage in the text heap, the remaining space after these characters being used for system housekeeping purposes. Each successive 12 bytes require another cell, or 16 bytes of storage. As atoms get longer, up to their limit of 8192 characters, their storage requirement tends towards 4/3 of their length.

Unlike atoms, strings are stored as text each time they occur. The first eight bytes of a string require one text heap cell, or 16 bytes, and then successive characters are stored, as with atoms, at 12 characters per 16-byte text heap cell. Again, as their length increases, the space required to store strings is approximately 4/3 of their length.

Because atoms are stored once each, programs with large numbers of long, but re-used atoms, take up less text heap space than might be imagined. The same programs, using strings in place of atoms, would require more text space. If the majority of atoms are unique, then the same program would be more space-efficient if it used strings rather than atoms. The real reasons for choosing between atoms and strings should be governed by their intended uses, rather than their storage requirements. Atoms are slower to create than strings, but once created, are more efficiently indexed and stored, and can be used as predicate names, file names, and so on. Strings are very fast to create, but are not used in first-argument indexing; their intended uses are for input and output streams, storage of text messages and prompts, and for handling the contents of files.

As with the (data) heap, unreferenced atoms and strings in the text heap become garbage, and can be collected automatically for recycling of their memory. The only difference between these two heaps in this respect is that atoms and strings may be referenced in programs as well as in dynamic data, and their memory can only be recycled if the respective program clauses or relations are first deleted.
The final heap is for program code itself, and this one is easy to understand. Each clause that is added to a dynamic program, and each entire optimised predicate, occupies one or more 128-byte cells in the program heap. This heap is not "garbage collected" as such, because when code is deleted, its memory is directly recycled there and then.

In general, text and program heap space needs to be allocated in step: larger programs tend to contain larger numbers of unique atoms and strings. The default settings of 512kb and 2048kb respectively for these two heaps allows for some fairly large programs to be loaded and run, but these allocations are by no means excessive, and should be freely increased as needed. To see how much space a program requires, a predicate such as the following can be written to consult a file and report its memory requirements:

```
consult( File, Text, Prog ) :-
  gc( 1 ),
  free(_, _, _, _, T0, P0, _, _, _),
  consult( File ),
  gc( 1 ),
  free(_, _, _, _, T1, P1, _, _, _),
  Text is T0 - T1,
  Prog is P0 - P1.
```

The calls to "gc(1)" invoke the garbage collector manually, simply to ensure that the returned values of free text space in the calls to "free(...)" are maximized for measurement purposes. The amount of text and program space available both before and after loading the program are returned, and their differences computed. Whatever results are returned for "TextUsed" and "ProgUsed" reflect the minimum text and program space required by the file "foo" itself, not taking into account the requirements of any other modules that might have been loaded, or of the system itself. The predicate `free/9` is matched with another, called `total/9`, that returns the total settings, so an overall text and program heap memory requirements can be computed with a program such as:

```
in_use( Text, Prog ) :-
  total(_, _, _, _, T0, P0, _, _, _),
  free(_, _, _, _, T1, P1, _, _, _),
  Text is T0 - T1,
  Prog is P0 - P1.
```

The returned values here are the absolute minima required to load all current program modules and the system itself. Setting the text and program heap size to about 256kb and 1024kb above these values respectively should give most applications ample working space.
Command Line Switches

Having discussed the stacks and heaps in WIN-PROLOG, and explained why most of them should be left at their default sizes, it is time to describe how their memory allocations are changed. A series of nine command line switches can be used to change the sizes, given in kilobytes, of each of the memory areas:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Bnnn</td>
<td>64</td>
<td>Backtrack Stack: normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug</td>
</tr>
<tr>
<td>/Lnnn</td>
<td>64</td>
<td>Local Stack: normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug</td>
</tr>
<tr>
<td>/Rnnn</td>
<td>64</td>
<td>Reset Stack: this should be left at its default setting; filling this stack is nearly always an indication of an application bug</td>
</tr>
<tr>
<td>/Hnnn</td>
<td>256</td>
<td>Term Heap: this can be increased modestly, but seldom needs to exceed this size (most applications can run with a far smaller heap); the apparent need for a huge (&gt;1024kb) heap should be considered with suspicion as an indication of possible application bugs</td>
</tr>
<tr>
<td>/Tnnn</td>
<td>2048</td>
<td>Text Heap: this should be increased in line with the size of the application and its data set, and as a rough guide should be about 25% of the size of the program heap</td>
</tr>
<tr>
<td>/Pnnn</td>
<td>8192</td>
<td>Program Heap: this should be increased in line with the size of the application and its data set, and as a rough guide should be about 400% of the size of the text heap</td>
</tr>
<tr>
<td>/Snnn</td>
<td>64</td>
<td>System Stack: normally this should be left at its default setting; filling this stack is nearly always an indication of an application bug</td>
</tr>
<tr>
<td>/Innn</td>
<td>256</td>
<td>String Input Buffer: this is the maximum size of string that can be used as an input stream, and can be adjusted to suit the application</td>
</tr>
<tr>
<td>/Onnn</td>
<td>256</td>
<td>String Output Buffer: this is the maximum size of string that can be used as an output stream, and can be adjusted to suit the application</td>
</tr>
</tbody>
</table>

After carefully considering the points made in this appendix, please refer to Appendix B for further information about using command line switches and modifying their default values.

As a footnote, it should be commented that when WIN-PROLOG starts up, the current settings (in kb) for these nine memory areas can be viewed on
Run-time Adjustments

Starting in version 4.500 of WIN-PROLOG, it has become possible to adjust the nine memory areas dynamically at runtime. A special predicate, xinit/9, can be used to set new values for some or all of the memory areas in a single call.

In order perform this memory reallocation, xinit/9 performs a full "compacting" garbage collection on all stacks and heaps, before writing their contents together other data buffers to a temporary file in the current working directory. Memory is then reallocated according to the user's supplied parameters, before the temporary file is reloaded and then deleted.

Here is an illustrated transcript to show xinit/9 in action; note that parameters are specified in kilobytes, with the two special cases of zero ("0"), which causes the given data area to remain unchanged, and minus one ("-1"), which resets a memory area to its default size; suppose WIN-PROLOG had been started with the command line switches, "/b96 /r96"; the following sequence uses ver/1 to show how the memory settings can be changed by calling xinit/9:

```prolog
| ?- ver( 0 ).
BDS WIN-PROLOG 7.000 X86 S/N 0020596320 28 Feb 2019
Copyright 1989-2019 Brian D Steel (www.solanum.org)
Licensed To: LPA Development and Documentation Team
B=96 L=64 R=96 H=256 T=2048 P=8192 S=64 I=256 O=256
yes
| ?- xinit( -1, 0, 0, 512, 0, 0, 0, 0, 0 ).
yes
| ?- ver( 0 ).
BDS WIN-PROLOG 7.000 X86 S/N 0020596320 28 Feb 2019
Copyright 1989-2019 Brian D Steel (www.solanum.org)
Licensed To: LPA Development and Documentation Team
```


In the above transcript, the changed settings have been shown in bold.

Summary

In this appendix, we have looked at a number of aspects of the "black art" of Prolog memory management, and discovered that, apart from two of the three heaps, in most cases, default settings for stack and heap sizes should be left unchanged. Before increasing stack or data heap sizes, application should be checked for inefficient structures and other bugs. For each of the following errors, the corresponding checks should be made:

<table>
<thead>
<tr>
<th>Error</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backtrack Stack Full</td>
<td>make sure there are no unnecessary choice points being left: use the cut to commit to current results wherever possible before making recursive calls</td>
</tr>
<tr>
<td>Local Stack Full</td>
<td>make sure there are no unwanted head recursions: search for any recursive calls which are followed by further calls, and rearrange code as necessary</td>
</tr>
<tr>
<td>Reset Stack Full</td>
<td>make sure there are no unnecessary choice points being left: use the cut to commit to current results wherever possible before making recursive calls, and if still a problem, try to bind variables before the introduction of choice points in a given clause</td>
</tr>
<tr>
<td>Heap Space Full</td>
<td>decide whether the application really needs tens of thousands of data elements: if not, make sure there are no unnecessary choice points being left: use the cut to commit to current results wherever possible before making recursive calls, and avoid passing large list parameters around if not necessary</td>
</tr>
<tr>
<td>Text Space Full</td>
<td>after checking that this is not simply a heap related problem, restart Prolog with an increased text heap in line with the complexity of the application, maybe using the free/9 and total/9 predicates to obtain a measure of how much space is genuinely required</td>
</tr>
<tr>
<td>Program Space Full</td>
<td>restart Prolog with an increased program heap in line with the complexity of the application, maybe using the free/9 and total/9 predicates to obtain a measure of how much space is genuinely required</td>
</tr>
<tr>
<td>System Stack Full</td>
<td>check the application for incorrect unifications which result in infinitely deep structures, and correct them</td>
</tr>
<tr>
<td>Input Space Full</td>
<td>too long a string has been used as an input stream: modify the application to use memory buffers (see fcreate/5) or restart Prolog with an input buffer which reflects the requirements</td>
</tr>
<tr>
<td>Output Space Full</td>
<td>too long a string is being created as an output stream: modify the application to use memory buffers (see fcreate/5) or restart Prolog with an output buffer which reflects the requirements</td>
</tr>
</tbody>
</table>
Hopefully, the explanations, examples and “cookbook” solutions above will help demystify memory management under WIN-PROLOG, and therefore help readers write code which runs comfortably without the need for huge and wasteful memory resources.
Appendix I - Time and File Handling

Inspired, once upon a time, by the then imminence of a new Millennium, and with the resultant worldwide focus that this placed upon date/time related computing, the opportunity was taken back in 1998 to make a number of extensions and modifications to the way dates and times were handled in WIN-PROLOG. Some of these issues related to file handling (notably directory listings and timestamp management), which is why both time and file handling is discussed in this document. The main issues which were addressed are described in the following sections.

The Midnight Rollover Bug

With all the historical hype around "Y2K", or the "Millennium Bug", it was and still is worrying just how many program contained a more insidious bug, the "Midnight Rollover Bug", which could strike any day of any year. The problem arose in pre-4.0 versions of WIN-PROLOG because, thanks to the MS-DOS heritage, system date and time were returned by separate predicates (date/3 and time/4 respectively). Any program which wanted both the date and time, but which simply called the two predicates in turn, was at risk:

```prolog
date_time( Y, M, D, H, N, S, F ) :-
    date( D, M, Y ),
    time( H, N, S, F ).
```

If such a predicate is called at around midnight, give or take about 55ms, the resulting date/time could be wrong by 24 hours. For safety, code should have been written to check for date rollover, with programs such as the following:

```prolog
date_time( Y, M, D, H, N, S, F ) :-
    date( D1, M1, Y1 ),
    time( H1, N1, S1, F1 ),
    date( D2, M2, Y2 ),
    ( D1 = D2
    -> (Y,M,D,H,N,S,F) = (Y1,M1,D1,H1,N1,S1,F1)
    ; (Y,M,D,H,N,S,F) = (Y2,M2,D2,0,0,0,0)
    ).
```

Here is another program which performs a midnight rollover check:

```prolog
date_time( Y, M, D, H, N, S, F ) :-
    repeat,
    date( D, M, Y ),
```

time( H, N, S, F ),
date( C, _, _ ),
C = D,
!.

The irony was that Windows provided both date and time, simultaneously, in a single system call, and the previous date/3 and time/4 predicates each called this function, but to only return their relevant components. Had a single predicate existed which returned all the data elements, code such as the above would have been unnecessary.

The Comparison of Times and Dates

The previous date/3 predicate returned its arguments as "Day, Month, Year", which is back-to-front for comparisons or sorting; furthermore, date/3 was "English" in layout: Americans would have expected "Month, Day, Year", which is also wrong for sorting. The previous time/4 predicate returned its data as "Hour, Minute, Second, Hundredth", which is correct for sorting, but inconsistent with the old date/3.

Curiously, the old dir/3 predicate returned file times as a list in the sequence: [Year,Month,Day,Hour,Minute,Second], which was ideal for sorting.

Elapsed Times and Arithmetic

Various predicates, including the old ticks/1, together with stats/4 and others returned or used times as simple 32-bit integer millisecond counts. These values would overflow after just over 24.85 days, becoming negative: the "Ticker Overflow Bug". After about another 24.85 days, they would overflow a second time and start counting again. While integer counts were convenient for arithmetic purposes, with the increased use of WIN-PROLOG as a server technology, 24.85 days was felt to be too short a time for the expiry of timers.

File Timestamps and Other Data

The old dir/3 predicate returned a subset of the data available for one or more files, massaged into a format that coincided with the information available to MS-DOS programs. Only the Windows "last write" timestamp was returned, and even then with its resolution limited to one second. The old stamp/2 predicate used DOS-style integer timestamps, which were very difficult to use apart from when simply preserving an existing file time around a write operation.

A Universal Format for Time

The issues discussed above, among others, lead to the adoption of a new standard format for date/time throughout the WIN-PROLOG kernel. All times are now represented as a conjunction of two integers which correspond to the day and millisecond counts respectively: this is known as a "<date_time>" data structure. For example, the following structure indicates a period of 123 days and 456.789 seconds:
The reason for this structure is that, even with the limits imposed by the use of 32-bit integers, it permits all later time values to compare as "greater" than all earlier ones. Had a simple millisecond count been returned, every 24.85 days or so the low-order 32-bit integer would switch from a positive to a negative value, or vice versa. Because there are only 86400000 milliseconds in a day, the chosen scheme means that the low order word will always be a value between 0 and 86399999. It will take a very long time (a few thousand years) before the high order 32-bit integer (day count) overflows.

Any pair of times can be processed arithmetically. For example, to find the difference (elapsed time) between an initial time (DY1,MS1) and a final time (DY2,MS2), all that is needed is the expression:

\[
\text{Milliseconds is } (\text{DY2-} \text{DY1}) \times 86400000 + (\text{MS2-MS1}).
\]

This is far simpler than handling years, months, days, hours, minutes, seconds etc. Two new predicates have been provided to convert between day counts and year/month/day values, and between millisecond counts and hour/minute/second/millisecond values.

### Old, New and Changed Predicates

In order to support the new `<date_time>` structure throughout the system, a number of predicates have been removed, modified or added between versions 3.6 and 4.0 of WIN-PROLOG; here is a brief summary:

<table>
<thead>
<tr>
<th>Status</th>
<th>Predicate</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;old&gt;</td>
<td>date/3</td>
<td>use time/2 or time/7 instead</td>
</tr>
<tr>
<td>&lt;old&gt;</td>
<td>date/4</td>
<td>use time/4 instead</td>
</tr>
<tr>
<td>&lt;old&gt;</td>
<td>ticks/1</td>
<td>use time/2 instead</td>
</tr>
<tr>
<td>&lt;old&gt;</td>
<td>time/4</td>
<td>use time/2 or time/7 instead</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>file/3</td>
<td>return arbitrary file data</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>time/2</td>
<td>return elapsed or local time</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>time/4</td>
<td>convert between days and yr/mt/dy</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>time/5</td>
<td>convert between ticks and hr/mn/sc/ms</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>time/7</td>
<td>return time as yr/mt/dy/hr/mn/sc/ms</td>
</tr>
<tr>
<td>&lt;mod&gt;</td>
<td>dir/3</td>
<td>new attribute handling, filenames only</td>
</tr>
<tr>
<td>&lt;mod&gt;</td>
<td>getflg/3</td>
<td>uses new dir/3 style attributes</td>
</tr>
<tr>
<td>&lt;mod&gt;</td>
<td>pdict/4</td>
<td>uses new dir/3 style attributes</td>
</tr>
</tbody>
</table>
The time/n Predicates

Because of the importance of Y2K (and midnight rollover) issues, it was considered desirable that all code using dates and times be checked carefully. It was thus a calculated “nag” to re-use the time/4 predicate, in the hope that it would encourage an early review of date/time software for version 4.0!

Any code which called one of the old date/3 or time/4 can simply be modified to call the new time/7, noting only that the seventh argument now returns milliseconds, rather than hundredths. Any code which called both date/3 and time/4 should now make a single call to time/7, eliminating the risk of midnight rollover: furthermore, any rollover detection code can now be removed.

The Time Emulation Library

A library file, "TIME.PL", has been implemented to provide direct emulations of the old date and time predicates. Because of its use of hide/2, this library should only be used for development-time emulation, providing cover while code is modified to use the new time/n predicates. This library should be loaded before any user code which wishes to use the old definitions. This library implements the old predicates, time/4, date/3, date/4 and ticks/1.

The dir/3 Predicate

An exhaustive evaluation of available system code suggested that the existing dir/3 predicate was used in two primary ways: first, to return a list of file names for further processing, and second, to obtain a specific piece of information about a single file. Because of its quirky (inclusive or) attribute handling, its returned information being incomplete (in Windows terms), and the wish to use the new time format, this predicate has been reimplemented.

The new dir/3 predicate uses a file pattern and attribute integer as before, but the attributes are interpreted differently. In short, the fewer the attributes that are specified, the more files that may be returned. Setting any of the lower 16 bits of the attributes integer specifies attributes that must be present; setting the upper 16 bits specifies attributes that must be absent. Some examples are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>16’000000000</td>
<td>all files</td>
</tr>
<tr>
<td>16’000000001</td>
<td>files which are read-only</td>
</tr>
<tr>
<td>16’000010000</td>
<td>files which are not read-only</td>
</tr>
<tr>
<td>16’000000002</td>
<td>files which are hidden</td>
</tr>
<tr>
<td>16’000000003</td>
<td>files which are hidden and read-only</td>
</tr>
</tbody>
</table>
Unlike its predecessor, the new `dir/3` only ever returns filenames: in order to find out information about a file, a new predicate, `file/3` predicate has been added. This can return any piece of Windows information about a file, and if not available, substitutes the appropriate DOS or Win16 information.

**The stamp/2 Predicate**

Because of the relative obscurity of its previous date/time format (a 32-bit integer encoding a DOS timestamp), most uses of the `stamp/2` predicate involved bracketing a write operation with two calls, the first to read the timestamp, and the second to write it back. Such code will work without modification, although the timestamp will now be represented in the new time format. This will lend additional uses to this predicate. For example, the following program can be used to implement a Unix-like "touch" utility to give a file a timestamp corresponding to the current local time:

```prolog
touch( File ) :-
    time( 1, Stamp ),
    stamp( File, Stamp ).
```

**The stats/4 Predicate**

This predicate returns counts of total elapsed running time, together with counts of idle time (time spent waiting for user input) and garbage collection time, as well as a garbage collection count. All three times are now returned in the new time format, rather than as simple millisecond counts. Any code which uses `stats/4` directly will need appropriate modifications.

**Timers and Beeps**

Two groups of time-related predicates remain unchanged: the named timers, (see `timer_create/2`) together with the noise-making `beep/2` predicate, still work on millisecond delays. It is not envisaged that either of these need to support durations in the order of days, let alone 49.7 days, and so these predicates retain integer milliseconds as their time arguments. The `ms/2` predicate, which is used for benchmarking goals, returns a simple millisecond count, but as it now uses `time/2` to obtain its information, the arithmetic applied to its data means that once 24.85 days have passed, a positive, integer-precise value will still be returned, albeit represented as a 64-bit floating point number. Not until something of the order of $10^{14}$ milliseconds have passed (over 3000 years...), will this count lose millisecond-level resolution!
Appendix J - LZSS Compression

As its name suggests, "LZSS Compression" is based on Lempel-Ziv compression, and is a modified version of the original LZ77 algorithm. In this algorithm, a sliding window is maintained over recently output data, while a look-ahead 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. Depending upon the sizes of sliding window and look-ahead buffer, compression ratios of up to 64:1 are theoretically possible for highly patterned data; in practice, ratios of 2:1 to 4:1 are more usual.

The original LZ77 algorithm had two major faults. Firstly, because for every symbol being encoded, the entire sliding window had to be searched, position by position, compression was very slow: furthermore, doubling to the size of the window to achieve better compression also doubled the time taken to perform that compression. Secondly, the method used to encode single (unmatched) characters was very inefficient. LZSS employs fundamental improvements over LZ77 in both these areas, in that it uses a binary tree for string searching, and a more efficient code stream for representing matches and data bytes. The LZSS implementation built into WIN-PROLOG is fast, and largely unaffected by sliding window or look-ahead buffer size.

Sliding Window and Look-Ahead Sizes

For maximum encoding efficiency, the combination of window address and match length used by LZSS must fit into an exact number of bytes. In the implementation used here, two bytes (16 bits) are used. This constraint results in a play-off between sliding window size (how many bytes of recently processed data are kept for matching) and look-ahead size (how long a match is possible). If 12 bits are used for the sliding window, then matches are possible with anything in the most recently processed 4096 bytes of data; however, only 4 bits can be used to denote match length, so the best possible match is just 16 bytes. Because each match requires 2 bytes of output, the best compression ratio possible is 8:1, where every look-ahead string finds a perfect match. By using 11 bits for the sliding window, and 5 for the look-ahead, only 2048 bytes are kept for matching at any one time, but the longest possible match is 32 bytes, resulting in a theoretical compression ratio of 16:1 in a file consisting solely of an initial string of the right length and repeated subsequent perfect matches.

The implementation of LZSS used in WIN-PROLOG allows anything between 9/7 (512 byte sliding window, 128 byte look-ahead) and 13/3 (8192 byte sliding window, 8 byte look-ahead) to be selected. Files with large amounts of repetitive data in localised areas, such as uncompressed bitmaps, compress better with 9/7 or 10/6 LZSS; program source files compress best at around 11/5, and ordinary text at around 12/4. The greater the amount of self-similarity within smaller regions, the better the file will compress with a smaller sliding window and bigger look-ahead. The only overhead in compressed output is a 4-byte header at the start and 32-bit checksum at the end of the data stream, (8 bytes total).
The stuff/4 and fluff/3 Predicates

The predicates which implement LZSS compression and decompression are called stuff/4 and fluff/3 respectively. Both of these take their input from the current input stream, and emit their output to the current output stream. Their three arguments specify the size of sliding window (look-ahead is computed from this), the total number of raw (uncompressed) bytes processed, and the total number of compressed bytes processed. By experimenting with the sliding window size and comparing the last two values, it should be easy to determine the optimum setting for whichever type of data you want to compress.

Applications of LZSS Compression

There are many potential applications of LZSS compression: these include the creation of stuffed archive files, which contain an accumulation of compressed files with their names, creation dates/times and attributes, the simple stuffing of individual files, such as bitmaps, which normally use large amounts of disk space, or the compaction of data which needs to be transmitted by modem or some other slow device. More advanced uses might include the creation of mixed database files consisting of uncompressed index information, and compressed data records.
Appendix K - MZSS Encryption

As its name suggests, "MZSS Encryption" makes use of a Marsaglia/Zaman "pseudo random number generator" ("PRANG"), which has the primary benefit of offering a very large key size (1185 bits, compared with just 64 bits in WIN-PROLOG's other, "linear congruential" PRANG). The "M2/PRANG" is seeded by a user-specified password of up to 148 characters, and successive numbers are then combined (xor) with the plaintext in order to encrypt it, or are combined with the cyphertext in order to decrypt it. Several special features of MZSS encryption make it especially secure, and these will be described in the following sections.

Comb Filtering

The M2/PRANG used in MZSS encryption compute successive 32-bit numbers using the formula:

\[ r_n = (r_{n-24} - r_{n-37} - b) \mod 2^{32} \]

While having excellent random properties, and a massively long cycle (there are well over \(2^{1180}\) numbers in the sequence), there is a weakness in the M2/PRANG. If a history of the previous 37 numbers is visible, it is possible to guess the next one +/- 1 bit: this is quite sufficient for an intelligent attack to decrypt a document, if the first 37 characters of the document's plaintext are known.

MZSS encryption prevents such an attack by only using a random sample of numbers from the M2/PRANG sequence. Prior to using the modified generator, MZSS/PRANG, a 32-bit shift register is initialised to zero. This register is used to control which M2/PRANG numbers are returned to the user using the following (pseudocode) algorithm:

```
MZSS_PRANG:
call MZ_PRANG to obtain NUMBER
shift SHIFT right one bit into CARRY
if   SHIFT is zero
then copy NUMBER into SHIFT
   set topmost bit of SHIFT
   discard NUMBER and start over
else if CARRY is one
then discard NUMBER and start over
else return NUMBER
```

The effect of this "comb filtering" is to reveal only randomly-selected numbers from the random sequence: this prevents a knowledge of the generator's history from being of any use in predicting future numbers. Because of this feature, it is not possible to decode into a file even if the first 37 or more
Sequence Variation

A common attack on simple cyphertext is to find some other document to which both the encrypted and decrypted version are available. Suppose that a document, "Vulnerable", is to be attacked, and the attacker has access to a pair of documents, "Plaintext" and "Cyphertext". On the (fairly reasonable) assumption that the same password has been used to encode both Vulnerable and Cyphertext, it is possible to decode the former, at least as far as length of the Plaintext document, as follows:

XOR PLAINTEXT with CYPHERTEXT to generate KEYSEQUENCE, then
XOR VULNERABLE with KEYSEQUENCE to generate DISCOVERED

Only by using a different password on every occasion, can this attack be avoided: MZSS encryption prevents such an attack by adding a "public" variation to the user's password key. In the current implementation, this is simply the SHA-256 hash of the combined system time and performance counter at the moment of encryption. This key is combined with the user's password before seeding the MZSS/PRANG, but is also stored in the cyphertext stream. During decryption, this stored hash field is combined with the user password.

This "sequence variation" ensures that, even if exactly the same plaintext file is encrypted with exactly the same password more than once, a unique cyphertext file will be created every time unless the system clock is artificially reset. Because of this feature, having access to the plaintext and cyphertext versions of any one or more documents will be of no help in decrypting further documents, even if they were encrypted with the identical password.

The encode/3 and decode/2 Predicates

The predicates which implement MZSS encryption and decryption are called encode/3 and decode/2 respectively. Both of these take their input from the current input stream, and emit their output to the current output stream. Their two arguments specify the password key and the total number of raw (unencrypted) bytes processed.

Applications of MZSS Encryption

There are many potential applications of MZSS encryption: these include the creation of encrypted archive files, which contain an accumulation of encoded files with their names, creation dates/times and attributes, the simple protecting of individual files, or the encryption of data which needs to be transmitted by modem or some insecure device. More advanced uses might include the creation of mixed database files consisting of unencrypted index information, and encrypted data records.
Appendix L - ASCII, Unicode, and All That

A major new feature of WIN-PROLOG 4.100 is full support for Unicode, a standard for encoding many thousands of characters, which suffices to display virtually any written text in any language in the world. This contrasts with ASCII, which only supports 128 characters geared towards the English language, and its various OEM, ANSI and other extensions which support up to 256 characters to help cover other alphabet-based languages.

Unicode Encodings in Memory

When it was first mooted in the early 1990s, Unicode was conceived of as a simple, fixed-width 16-bit encoding that would permit over 65,000 characters each to be assigned a unique number, or "code point". It was thought that this would be sufficient to encode every character in every language in the entire world, with space to spare. Adding Unicode support to WIN-PROLOG would simply have been a matter of assigning two bytes of storage to each character in an atom or string, rather than one as before.

Unfortunately, by the beginning of the year 2000, over 48,000 of the 65,000 characters were already used up, and it was expected to run out of the original range of code points before or during 2001. As a result, the Unicode 3 standard, published in February 2000, defined over 1,000,000 possible code points. Patently, this meant that a simple linear 16-bit encoding was no longer enough: with new $2^{20}$ code points being added to the original 65535, a space of some 20.1 bits was now required to encode each character.

The simplest Unicode encoding, called UTF-32, simply assigns a single 4-byte integer to each possible code point, and is the natural successor to the original 16-bit encoding. With just over 20 bits used by Unicode, a 32-bit storage mechanism is very wasteful of memory, and (compared with ASCII) would make Western text four times its previous size, requiring four bytes per character.

Two other Unicode encodings are defined: the first, called UTF-8, uses 8-bit codes for the original ASCII range (00h..7Fh), and 16, 24 or 32-bit codes for the remaining characters. It enables plain English to be represented, on average, at about 1.1 bytes per character, and averages out at about 3 bytes per character for average "World Text". It is, however, a cumbersome encoding that requires considerable "bit twiddling" to extract or encode higher code points, and so although compact, it is expensive in terms of processing.

The second short Unicode encoding is called UTF-16: this uses 16-bit encodings for all characters defined up to the year 2000 (as with the original Unicode), and 32-bit "surrogate pairs" for the additional characters that were expected to be added from about mid-2000 onwards. Plain English requires 2 bytes per character, as does all current "World Text". It is envisaged that even after the addition of many surrogate pairs, the latter average will still be only slightly above 2, because the additional characters will be extremely rare.
Unicode Encodings in Files

There are three Unicode basic memory encodings: UTF-32, UTF-16 and UTF-8; unfortunately, these require up to ten different file formats for disk storage, and for serial transmission. The reason is that storage devices and serial protocols are byte-oriented, and different processors and operating systems represent multi-byte integers in one of two ways, called "big endian" and "little endian". To store a number, say 1024, as a two-byte integer, an operating system on the Intel 80x86 processor would output the little endian byte pair:

```
00 04
```

while an operating system on a Macintosh would output the big endian pair:

```
04 00
```

Because Unicode files can come from any source, both numeric formats need to be supported, for each of the UTF-16 and UTF-32 encodings. UTF-8 stores bytes, and so does not need big and little endian file variants.

In order to help deduce which of five possible file or serialisation encodings are used, Unicode optionally allows a special lead character, the "Zero Width No Break Space" (ZWNBSP), to begin the file. When used in this mode, it is treated as a "Byte Order Mark" (BOM), a sort of mini-header that identifies the encoding, hence the eight variations. The ZWNBSP character itself has a code point of FEFFh; the byte-reversed equivalent character, "Byte Swapped Byte Order Mark" (FFFEh) is defined as illegal in Unicode, so the ordering and encoding of the bytes "FE" and "FF" provide all the clues necessary to decode the rest of the file or serialised stream:

<table>
<thead>
<tr>
<th>Name</th>
<th>BOM</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8*</td>
<td>EF BB BF</td>
<td>The BOM is discarded, and the rest of the file assumed to be in UTF-8 format. Any further EF BB BF is a ZWNBSP, and EF BF BE (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-8</td>
<td>&lt;none&gt;</td>
<td>The file known to be in UTF-8 format in advance. Its first character cannot be ZWNBSP unless preceded by BOM. Any other EF BB BF is a ZWNBSP, and EF BF BE (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-16BE*</td>
<td>FE FF</td>
<td>The BOM is discarded, and the rest of the file assumed to be in UTF-16 big-endian format. Any further FE FF is a ZWNBSP, and FF FE (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-16LE*</td>
<td>FF FE</td>
<td>The BOM is discarded, and the rest of the file assumed to be in UTF-16 little-endian format. Any further FE FF is a ZWNBSP, and FE FF (BSBOM) is an error.</td>
</tr>
<tr>
<td>Encoding</td>
<td>BOM</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>&lt;none&gt;</td>
<td>The file known to be in UTF-16 big-endian format in advance. Its first character cannot be ZWNBSP unless preceded by BOM. Any other FE FF is a ZWNBSP, and FF FE (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-16LE</td>
<td>&lt;none&gt;</td>
<td>The file known to be in UTF-16 little-endian format in advance. Its first character cannot be ZWNBSP unless preceded by BOM. Any other FF FE is a ZWNBSP, and FE FF (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-32BE*</td>
<td>00 00 FE FF</td>
<td>The BOM is discarded, and the rest of the file assumed to be in UTF-32 big-endian format. Any further 00 00 FE FF is a ZWNBSP, and FF FE 00 00 (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-32LE*</td>
<td>FF FE 00 00</td>
<td>The BOM is discarded, and the rest of the file assumed to be in UTF-32 little-endian format. Any further FE FF 00 00 is a ZWNBSP, and 00 00 FE FF (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-32BE</td>
<td>&lt;none&gt;</td>
<td>The file known to be in UTF-32 big-endian format in advance. Its first character cannot be ZWNBSP unless preceded by BOM. Any other 00 00 FE FF is a ZWNBSP, and FF FE 00 00 (BSBOM) is an error.</td>
</tr>
<tr>
<td>UTF-32LE</td>
<td>&lt;none&gt;</td>
<td>The file known to be in UTF-32 little-endian format in advance. Its first character cannot be ZWNBSP unless preceded by BOM. Any other FF FE 00 00 is a ZWNBSP, and 00 00 FE FF (BSBOM) is an error.</td>
</tr>
</tbody>
</table>

Note: the UTF-32 BSBOM encodings shown above are taken from an IBM-hosted webpage written by Mark Davis, President of the Unicode Consortium, although a careful reading of other text at www.unicode.org suggests that 00 00 FF FE should be the UTF-32 big-endian BSBOM, and FE FF 00 00 the UTF-little-endian BSBOM.

**Multiple Encodings**

Even once character encoding techniques are settled, Unicode has further complications. Many characters can be encoded in more than one way; a simple example is the "Angstrom" character, Å. Character code 212Bh is described in the Unicode Standard 3.0 as the "ANGSTROM SIGN", but its definition says that it is identical to another character, 00C5h which is described as "LATIN CAPITAL A WITH RING ABOVE". This, in turn, is defined as being identical to the pair of characters, 0041h and 030Ah ("LATIN CAPITAL LETTER A" together with "COMBINING RING ABOVE"). So here is one character which can be represented in three ways: there are many more complicated examples, where three or more characters can be considered to combine to produce single characters, either directly, or via other intermediate characters.

The multiple encodings posed a problem for embedding Unicode within WIN-PROLOG: if the code points 00C5h and 212Bh are to be treated as identical, not only to each other, but also to the pair of characters 0041h and 030Ah, "simple" operations like unification of atoms and strings becomes hugely more complex, requiring recursive table look-up character by character, rather than simple code comparisons.
The problem extends to sorting: depending upon language, different rules apply to alphabetic ordering. In Finnish, for example, the character "A" (00C4h) is listed at the end of the alphabet, after "Z"; in German, it is treated as "AE", and sorted accordingly; in English, accents are ignored when sorting. As with unification, culturally-sensitive sorting would require continual table look-ups, character by character.

**Windows Support**

When it comes to the Windows operating system, Unicode support becomes complicated, and even muddled. Versions of Windows based on DOS/Win32s (Windows 3.1, 95, 98 and even ME) have no native support for Unicode, but emulate it with the use of locale-specific "code pages" which map between Unicode and single- or multi-byte encodings based on 8-bit characters. Later versions of Windows based on NT (Windows NT4, 2000) contain complete Unicode support, as well as support for code pages.

A simple way more-or-less to support Unicode would be to write for the Win3x/9x/ME model, which would also work under WinNT/2k; however, the round-trip between Unicode -> CodePage -> Unicode loses information, so WinNT/2k users would be penalised by the necessity of supporting the more primitive versions of Windows. Alternatively, **WIN-PROLOG** code could be written directly for WinNT/2k, but in the process, it would no longer be able to run under Win3x/9x/ME. Two versions of **WIN-PROLOG** could be produced, one dumbed down for all versions of Windows, and the other spiced up for NT/2k, but the inevitable differences between the two systems would increase the potential for confusion, application bugs, and so forth, as well as complicating user choice.

**Unicode in WIN-PROLOG**

The previous sections give a very brief overview of some of the problems and constraints of supporting Unicode in **WIN-PROLOG**; there were, in fact, many more quirks and considerations involved with embedding wide character character support, not least in the areas of Operating System and Windows API interfacing. In brief, the following requirements emerged:

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compactness</td>
<td>Applications using western text should not be penalised by excessive memory requirements, so a compact storage mechanism should be used</td>
</tr>
<tr>
<td>Ordering</td>
<td>Existing sorting, comparing and unification routines should run as before, avoiding processing overheads</td>
</tr>
<tr>
<td>Encoding</td>
<td>All possible ASCII, Unicode and future character encodings should be supported fully</td>
</tr>
<tr>
<td>Expedience</td>
<td>Minimal source code changes be required to existing applications</td>
</tr>
<tr>
<td>Automation</td>
<td>All ASCII, 8-bit and Unicode file formats should be supported, as far as possible automatically</td>
</tr>
<tr>
<td>Universality</td>
<td>All versions of Windows should be supported by one version of <strong>WIN-PROLOG</strong></td>
</tr>
</tbody>
</table>
The result of these requirements, together with a lot of thought and planning, resulted in the implementation described in the remainder of this appendix.

**Character Sets and Definitions**

Before leaping into the nitty-gritty, it would be well to firm up the definitions of some terms which have been used rather freely in the past, relating to character sets. Previous WIN-PROLOG documentation has described "character sets" and even "fonts" using three terms: "ASCII", "OEM" and "ANSI". The first of these was almost always misused to describe arbitrary 8-bit characters; the terms "OEM" and "ANSI" were used to describe variations of an 8-bit character set which respectively mapped onto the IBM-PC ROM BIOS characters (OEM) or the Windows ANSI character set. One font, the "OEM Fixed Font", contained characters which conformed to the former set, while most other Windows fonts mapped onto the latter.

In the present documentation, character sets are described in the following terms:

<table>
<thead>
<tr>
<th>Character Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>This is a 7-bit character set, which encodes 128 characters in the range 0000h..007Fh inclusive; it is a pure subset of and comprises the first 128 characters of:</td>
</tr>
<tr>
<td>ISO/IEC 8859-1</td>
<td>This is an 8-bit character set, which encodes 256 characters in the range 0000h..00FFh inclusive; it is a pure subset of and comprises the first 256 characters of:</td>
</tr>
<tr>
<td>ISO</td>
<td>To avoid quoting ISO/IEC 8859-1 in full each time, WIN-PROLOG documentation will usually refer to this standard simply as &quot;ISO&quot;.</td>
</tr>
<tr>
<td>Unicode</td>
<td>This is a 20.1-bit character set, which encodes somewhat over 1,000,000 characters in the non-contiguous range 0000h..10FFFFh</td>
</tr>
<tr>
<td>RAW</td>
<td>This term describes 16-bit and 32-bit character sets, which notionally map onto the equivalent code points in Unicode, and therefore also in ISO and ASCII.</td>
</tr>
<tr>
<td>ANSI</td>
<td>This is the default character set used by 8-bit Windows API functions, and supported by most Windows fonts. It is very similar to ISO, except that it uses a number of code points (0080h..009fh) for printable characters (for example, the Euro currency glyph (€) and trademark symbol (™), while ISO and Unicode reserve these code points for control characters.</td>
</tr>
<tr>
<td>OEM</td>
<td>This is the default character set of English versions of the IBM PC ROM, and was used to define the lexical syntax of earlier versions of WIN-PROLOG, as well as its DOS-PROLOG, BAT-PROLOG and CON-PROLOG siblings.</td>
</tr>
</tbody>
</table>
The Lexical Syntax of WIN-PROLOG

All versions of WIN-PROLOG up to 4.040 used a 256-character lexical table to define characters into a range of types, based upon their meanings in the OEM (IBM-PC BIOS) character set. This table classified characters into a number of types, such as "letter" (a, B, Ç), "digit" (0..9), "graphic" (#, $, £) and so forth. Atoms can be written as any contiguous sequence of letters or graphics, but mixtures must be quoted: by extending the lexical table above the 7-bit ASCII range into the extended characters, 80h..FFh, the old WIN-PROLOG syntax worked well in the native OEM character set.

With Windows' ANSI character set (which is similar to ISO), the extended characters are assigned differently than in the OEM character set, and type mismatches occur among the extended characters: for example, code 00A3h is a pound sign in ANSI (and ISO and Unicode), but a lowercase u-acute in the OEM character set. WIN-PROLOG therefore treated this character as a graphic, where more correctly (in Windows and in Unicode) it should be a lowercase letter.

There were two approaches to modifying the lexical rules to account for Unicode: extend the table to over 1,000,000 entries or truncate it to the 7-bit ASCII range, 00h..7Fh, which happily corresponds exactly to the first 128 characters of the Unicode standard. The former approach was obviously inappropriate, and so the latter was chosen. All characters with special syntactic meanings (letters, graphics, digits, brackets, variable prefixes, and so on) are now contained within the ASCII character set, and all characters above this (80h upwards) are now treated as lowercase letters.

Direct Character Encoding Syntax

When processing quoted text (strings, atoms and char lists), previous versions of the WIN-PROLOG parser recognised characters for the form "~nnn", reading the integer, "nnn" and storing its (8-bit) character code:

?- X = `foo~13~10`.
X = `foo~M~J`

While it would have been simple to extend the "~nnn" syntax to allow unsigned 32-bit numbers, this would have become clumsy with larger character codes. There was also a problem with this syntax: it was not possible to juxtapose direct character syntax with literal digits. For example, trying to express a string with the character code 123 followed by the literal digit "4" would result in:

?- X = `~1234`.

which refers, instead, to character code 1234, which would have resulted in an Arithmetic Overflow Error in the parser.

The Unicode 3.0 standard uses hexadecimal in its character tables, and therefore it was decided to replace the old "~nnn" decimal syntax with a hexadecimal equivalent. This could have resulted in its own problems; for example, would:
refer to the character code, FFh, or to the pair of characters, <ctrl-F> and the letter "F"? The solution was to introduce compulsory parentheses into the syntax:

`~FF`  
`~{F}F`  
`~(FF)`

The first case describes a string containing two characters, one being <ctrl-F> and the other, the letter "F". The second case also contains two characters, one being character code 0Fh, and the other, the letter "F". The third case simply contains a single character, with the code FFh. This model allows even 7-bit ASCII text to specify 32-bit characters, as with the following example, which terminates with the character 2122h ("TRADE MARK SIGN"):

?- X = `This is a Unicode trade mark sign: ~(2122)`.
X = `This is a Unicode trade mark sign: ™`.

Previous versions of **WIN-PROLOG** system never wrote the "~nnn" syntax on output, but the new Unicode-enabled version will use the "~(xxx)" syntax whenever trying to display quoted text containing characters which are not supported directly by the current output device, such as when writing large Unicode characters to a file that has been opened in 7-bit ASCII mode.

**Full 32-bit Character Support**

After much consideration, it was decide to add full 32-bit character support to **WIN-PROLOG**: the cost, compared with limiting text characters to the 20.1 or so bits of Unicode, were minimal (or even negative, as it removed the need for range tests), and the benefits would be automatic future-proofing against new, extended character sets. For simplicity, to avoid the need to encode and access vast tables, each code point is assumed to be a distinct character; all apart from the first 128 (ASCII) characters are further thought of as lowercase letters. This means that, for example, the three valid encodings for Å (Angstrom), namely 212Bh, 00C5h, and (0041h,030Ah) are considered to be different characters within **WIN-PROLOG** itself (the last case is two characters).

Sorting, comparing, unification and other internal processes treat characters and their code points as being one and the same thing, so a call such as the following will fail:

?- atom_chars( A, 16'212b ), atom_chars( B, 16'00c5 ), A = B.
no

It would be supremely weird were the above call to succeed! Windows itself provides APIs which can convert between different encodings of the same
Unicode character, where this is required by an application.

There is one apparent quirk of which the reader should be aware: when comparing or sorting strings, code points are treated as unsigned 32-bit integers, but when comparing or sorting integers themselves, they are treated as signed integers in the ranges. Thus the atom, `~(ffffffff)` and string, `~(ffffffff)` are each thought of as containing a single character with a (decimal) value of 4294967295, the greatest possible value, but the char list: "~(ffffffff)" is actually a list containing the single (decimal) value of -1, which is one less even than the smallest ASCII, ISO or Unicode character! This can be seen in the calls:

```
?- cmp( C, '~(0)', '~(ffffffff)').
C = -1
?- cmp( C, `~(0)`, `~(ffffffff)`).
C = -1
?- cmp( C, "~(0)", "~(ffffffff)").
C = 1
```

The important point to remember is that the char list is not really a data type in its own right, but is simply an interpretation sometimes put on a list of integers. The third example above is simply a syntactic variation of the call:

```
?- cmp( C, [0], [-1] ).
C = 1
```

One final consideration applies to certain predicates, such as get/1 and get0/1: these return "-1" to indicate that the end of file has been reached. For files opened in ASCII, ISO and Unicode modes, this signal is still perfectly reasonable, but for any file opened in a 32-bit RAW mode, it can potentially lead to confusion, since FFFFFFFFh is a valid character, with the (signed) integer code point of -1. If a file has been opened in a 32-bit RAW mode, and is being read by one of the classic single-character input routines, any instance of "-1" should be followed by a call to at_end_of_file/0 to determine whether it is a real character, or genuinely the end of file.

**UTF-BS - A Unicode Format for WIN-PROLOG**

Because of the desire to support 32-bit characters, none of the existing Unicode encodings were suitable for internal user by WIN-PROLOG: even if characters were to be limited to the Unicode set, these encodings would be inefficient either in terms of memory usage or processing - or both! The various UTF schemes are robust, being designed for real-world access where files might be incorrectly indexed, or data lost in transmission, enabling the reading program to deduce that its data stream was out of synchronisation. Within WIN-PROLOG's internal data structures, such access was not an issue, and a more efficient encoding, dubbed "UTF-BS", was devised, which stores ASCII at 1 byte per character, ISO at 1.016 bytes per character (UTF-8 requires 1.5 bytes per character), 16-bit Unicode at 3 bytes per character (about the same as UTF-8) and all other characters at 5 bytes per character. This format is summarised below:
<table>
<thead>
<tr>
<th>Character Range</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>16'00000000 .. 16'000000FD</td>
<td>The first 253 characters, in the range 00h-FDh, are stored as normal 8-bit text, 1 byte at a time</td>
</tr>
<tr>
<td>16'000000FE .. 16'000FFFFF</td>
<td>The next $(2^16)-253$ characters are stored as an escape byte, FEh, followed by the 16-bit character value as 2 bytes, in big-endian ordering</td>
</tr>
<tr>
<td>16'00010000 .. 16'FFFFFFFF</td>
<td>The remaining $(2^32)-(2^16)$ characters are stored as an escape byte, FFh, followed by the 32-bit character value in 4 bytes, in big-endian ordering</td>
</tr>
</tbody>
</table>

The use of big-endian format (most significant byte first) will seem strange, because on the Intel 80x86 processor, integers are stored in little-endian format (least significant byte first); however, the choice of big-endian allows existing string-based sort and compare algorithms to work unchanged. Consider the old "~nnn" decimal syntax `WIN-PROLOG` string, written:

```
`ABC~255`
```

Previously this would have been parsed into and stored as the four bytes:

```
41 42 43 FF
```

Now consider the equivalent new "~(xxx)" hexadecimal syntax `WIN-PROLOG` string, written:

```
`ABC~(FF)`
```

In the new UTF-BS format, it will be stored as:

```
41 42 43 FE 00 FF
```

Now consider "~(xxx)" hexadecimal syntax `WIN-PROLOG` string:

```
`ABC~(100)`
```

In the new UTF-BS format, this will be stored as:

```
41 42 42 FE 01 00
```

When performing the existing compare algorithm, the first string would be compared, byte for byte, with the second, with the comparison stopping at the first different byte (the penultimate one, "00" versus "01"), and the second string would be correctly identified as "greater" than the first. However, if we
used little-endian encoding, the strings would have been stored:

    41 42 43 FE FF 00
    41 42 42 FE 00 01

and the byte-wise comparison would result in the first string being thought as "greater" than the second ("FF" versus "00" in the pre-penultimate byte).

By using "FE" and "FF" an escape bytes, and encoding the character in big-endian format in the ensuing 2 or 4 bytes, all existing byte-oriented sort and compare algorithms will correctly sequence a 32-bit character set, without having to "decode" them in any way. This would not be the case with UTF-8 or UTF-16 formats.

**Changes to Built-in Predicates**

So much for the background and details: what does it all mean? In short, **WIN-PROLOG 4.100** supports a 32-bit character set, where **WIN-PROLOG 4.040** and earlier were limited to using 8-bit characters. Because of the UTF-BS format used to store text in atoms and strings, the "hit" on existing applications is minimal. Existing ASCII applications use precisely the same amount of memory for storing text; even 8-bit binary applications use an average of 1.016 bytes to store each character. Only those applications using Unicode (or greater) characters require the bulk of the additional storage routines.

The term, "byte list", was previous used to describe lists of integers in the range 0..255, traditionally used to represent text in simple versions of Prolog:

```
?- X = "byte list".
```

This term has been renamed, "char list", since character are no longer limited to one byte in size:

```
?- atom_chars( X, [12345] ).
X = '~(3039)'
```

Notice the use of quoted output in hexadecimal notation: this will occur on output devices that cannot display character code 3039h ("HANGZHOU NUMERAL TWENTY"). On suitable versions of Windows, this character will display directly.

Because of the change in nomenclature, and to underscore the switch from "bytes" to "chars" as the basic unit of text, two name changes have taken place, and some predicates have subtle changes in their behaviour; for example:

<table>
<thead>
<tr>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
</table>

---

**Example**

```
?- X = "byte list".
```

**Comments**

Notice the use of quoted output in hexadecimal notation: this will occur on output devices that cannot display character code 3039h ("HANGZHOU NUMERAL TWENTY"). On suitable versions of Windows, this character will display directly.
formatted I/O
fread/4 and fwrite/4 now use a “c” format, rather than “b”, to specify that they are to read/write a list of chars (as opposed to bytes).

Text Data Type Conversion
atmbyt/2 and strbyt/2 have been replaced with atmchr/2 and strchr/2

<char> Data Type
char/1 is now true of any 32-bit integer, rather than just 8-bit integers; likewise, chars/1 is true of any list of 32-bit integers.

The main changes, however, are in File Input/Output and Windows API interfaces, as described in the next two sections.

File Input and Output

The input and output subsystem is one of two places where WIN-PROLOG has to present Unicode (or ASCII, ISO, RAW…) data to the outside world, and it is here that filters and checks determine which of its more than 4,000,000,000 32-bit characters are valid and how they will be encoded. At the lowest level, the old fcreate/4 and fdata/4 predicate has been replaced with fcreate/5 and fdata/5 respectively: the new fifth argument specifies which of eleven file formats will be used for the file during its read/write session:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>32-bit RAW-32LE</td>
</tr>
<tr>
<td>-4</td>
<td>16-bit RAW-16LE</td>
</tr>
<tr>
<td>-3</td>
<td>Unicode UTF-32LE</td>
</tr>
<tr>
<td>-2</td>
<td>Unicode UTF-16LE</td>
</tr>
<tr>
<td>-1</td>
<td>7-bit ASCII</td>
</tr>
<tr>
<td>0</td>
<td>8-bit ISO/IEC 8859-1</td>
</tr>
<tr>
<td>1</td>
<td>Unicode UTF-8</td>
</tr>
<tr>
<td>2</td>
<td>Unicode UTF-16BE</td>
</tr>
<tr>
<td>3</td>
<td>Unicode UTF-32BE</td>
</tr>
<tr>
<td>4</td>
<td>16-bit RAW-16BE</td>
</tr>
<tr>
<td>5</td>
<td>32-bit RAW-32BE</td>
</tr>
</tbody>
</table>

None of these formats expects or recognises a byte order mark (BOM): it is assumed that the application knows what the file contains. One level above fcreate/5 is the new open/3 predicate, which allows file formats to be specified directly:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Notice that, unlike with *fcreate/5*, all the *open/3* Unicode formats include the byte order mark (BOM): an error will be flagged if an attempt is made to open a Unicode file that omits this mark, or which is in the wrong Unicode format. The "automatic" mode works with read-only or read-write access to existing files, and parses the first four bytes of the file to search for a BOM in each of its five possible encodings: if found, the file is opened in the appropriate Unicode UTF mode; otherwise, it is opened in ISO mode. Predicates like *consult/1*, *compile/1* and *load_files/1* use this flag to automatically detect and process ISO and Unicode files.

**Random File Access and Indexing**

Compared with 8-bit files, Unicode's numerous file formats pose problems to any application when it comes to random access and indexing. If a file contains a series of fixed-width "lines", each of 72 characters followed by a 2-byte `<CR>`<LF> pair, it is possible to jump to (say) the 123rd line simply by multiplying:

\[
\text{Pos is 123} \times (72+2)
\]

and then directly indexing to the resulting file offset (9102). In all Unicode encodings except for the UTF-32 formats, indexing like this is impossible. In most formats, the only way to reach the 9102nd character would be to retreat to the top of the file, and start reading, counting and discarding the first 9101 characters. The variants of UTF-8 and UTF-16 are therefore efficient only for sequential access files. The two tagged UTF-32 ((a) and (b)) formats are just as efficient as the two untagged ones (BE and LE), although the programmer must remember to add 4 to the computed offset in the former cases to account for the BOM. Care should therefore be taken with *inpos/1* and *outpos/1*, and in general, they should only be used to record file offsets and return to these later.
The Windows API

Two special predicates, \texttt{winapi/4} and \texttt{wintxt/4}, replace their \texttt{winapi/3} and \texttt{wintxt/3} predecessors, with an additional argument specifying the text width. Unlike the \texttt{WIN-PROLOG}'s internal text processing and file I/O system, which supports Unicode 3.0 and beyond, the Windows API is limited to "ANSI" (8-bit) and "Wide" (16-bit) characters. The "A" at the end of almost every text-based function seen in earlier \texttt{WIN-PROLOG} examples stood for "ANSI", so a \texttt{WIN-PROLOG} 4.040 call such as:

\begin{verbatim}
?- \texttt{winapi( (kernel32,'MessageBoxA'), [0,`world`,`hello`,0], R ).}
\end{verbatim}

would call the ANSI version of "MessageBox", passing it the null-terminated ANSI strings "world" and "hello", alongside two zero parameters. In \texttt{WIN-PROLOG} 4.040, it would not have been possible to call the "Wide" version of this function, "MessageBoxW". The additional (third) argument of \texttt{winapi/4} and \texttt{wintxt/4} specifies the format of text arguments. Four formats are supported:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pass 8-bit &quot;Multibyte&quot; text</td>
</tr>
<tr>
<td>1</td>
<td>Pass 16-bit &quot;Wide Char&quot; as &quot;Multibyte&quot; text</td>
</tr>
<tr>
<td>2</td>
<td>Pass 16-bit &quot;Wide Char&quot; text</td>
</tr>
<tr>
<td>3</td>
<td>Pass 32-bit &quot;Raw Char&quot; text</td>
</tr>
</tbody>
</table>

These need further explanation. \texttt{WIN-PROLOG} now stores all text as 32-bit characters (albeit efficiently, thanks to the UTF-BS encoding technique). All characters are assumed to map to ASCII, ISO and Unicode as appropriate, so (for example), the trademark symbol (™) is stored as code 2122h. There is no such symbol in ISO/IEC 8859-1, but there is in Window's variant, ANSI: character number 99h. Calling \texttt{winapi/4} with the string:

```
`Trademark is ™`
```

therefore poses a problem. If the characters are copied directly from the string to the Windows API (as previously), an error will occur when the ™ symbol is encountered: it's character code of 2122h will not fit in a single byte. There are two ways it can be passed to Windows, however: either it can be converted from its Unicode value of 2122h to its ANSI code, 99h, which is an 8-bit value which can be passed to an "ANSI" function, or it can be passed directly as Unicode to the equivalent "Wide" function. The latter (preferable) approach only works in WinNT/2k, while the former (standby) approach might fail in some circumstances (such as whenever their is no ANSI encoding for some particular character). Which of these approaches is to be used is specified by that third argument, so:

\begin{verbatim}
?- \texttt{winapi( (kernel32,'MessageBoxA'), [0,`(2122)`,'`,0], 1, R ).}
\end{verbatim}

converts the Unicode value 2122h to ANSI 99h, before calling the ANSI text function, "MessageBoxA": this will work on all versions of Windows. Mean-
while:

?- winapi( (kernel32,'MessageBoxW'), [0,\~(2122)\~,\~,0], 2, R ).

will directly pass a Unicode string containing the character 2122h, because calling the Wide text function, "MessageBoxW": this will only work in Windows NT or 2k.

Given that the first call works in all versions of Windows, but the second only works in WinNT/2k, the question might well be asked, "why bother with the second case". The answer is twofold: first, conversion between Unicode and ANSI takes time, and second (more importantly) is not necessarily reversible. This can be seen more clearly when using wintxt/4; the call:

?- fcreate( buffer, [], 0, 0, 0 ).
   yes

will create a memory file which we can use to experiment. The next call attempts to load two characters, 2122h (TRADE MARK SIGN) and 2123h (VERSICLE) into the buffer, but gives an error because the third argument, "0", requests 8-bit text transfers with no conversion:

?- wintxt( buffer, 0, 0, \~(2122)\~(2123) ).
   Error 48, Character Error, Trying wintxt/4

Using a third argument of "1", we can try to convert the string to ANSI before storing it:

?- wintxt( buffer, 0, 1, \~(2122)\~(2123) ).
   yes

Apparently that's worked; but when we try to retrieve the text, it can be seen that something's gone wrong:

?- wintxt( buffer, 0, 1, T ), string_chars( T, C ).
   T = \"™\"
   C = [8482,63]

Clearly, although the Unicode trademark symbol has made the round trip to and from ANSI, the versicle has been replaced by a question mark. This is because versicle is not part of Windows' ANSI character set. Now consider:

?- wintxt( buffer, 0, 2, \~(2122)\~(2123) ).
   yes
This seems to have worked as before, but the third argument of "2" requests that 16-bit text is passed directly, without conversion. When we try to retrieve this text, we see that both characters have survived:

?- wintxt( buffer, 0, 2, T ), string_chars( T, C ).
T = `™~(2123)`
C = [8482,8483]

Even so, the versicle cannot be displayed in all versions of Windows, and might appear in the new "---(xxx)" hexadecimal syntax, as illustrated above.

The bottom line is that when the sole purpose of a winapi/4 or wintxt/4 call is to display something on the screen, it makes little difference whether a generic ANSI function is called, with a third argument of "1", or whether an WinNT/2k-specific Wide function is called, with a third argument of "2". When the data needs to remain undamaged, and possibly has to survive a round-trip, the latter is the only option.

Many of WIN-PROLOG's predicates make run-time decisions about which version of a function to call, so as to preserve maximum functionality together with general portability. This involves making a version check, and then calling one of two winapi/4 cases accordingly, effectively:

foo(...) :-
    (  windows_nt_test
        -> winapi( (...,'FooW'), [...], 2, R )
        ;  winapi( (...,'FooA'), [...], 1, R )
    ).

As was said in a previous section, when it comes to the Windows operating system, Unicode support becomes complicated!

Summary

By employing a custom text encoding format, dubbed UTF-BS, it has been possible to add 32-bit character support, and hence Unicode, to WIN-PROLOG, with minimal space or processing overheads compared to previous 8-bit character support. Character-level filtering of input and output has been supported by an additional arguments in each of fcreate/5, fdata/5, winapi/4, wintxt/4 and open/3. The lexical tables have been reduced from 8-bit to 7-bit, and all characters above 7F will be treated as lowercase letters. Sorting, comparing, unification and other critical routines have remained entirely unchanged, thanks to UTF-BS encoding.
Appendix M - Types of Compilation

This appendix discusses the differences between the three types of compilation, namely incremental, hashed and optimised compilation in WIN-PROLOG, and indicates why and when each of these types of compilation should be used.

Incremental Compilation: Clause by Clause

The term "incremental compilation" is used to describe a process where each individual program clause is compiled independently, without reference to other clauses. In WIN-PROLOG, incremental compilation is performed by handwritten 80386 assembler code, and is extremely fast. First argument indexing and unification instructions are highly optimised, even in the incremental compiler, making it ideal for use in small to medium sized data relations (collections of facts), whether or not these need to be modified dynamically at run time. Program relations (collections of rules, with or without recursion) are less efficiently compiled, but still run considerably faster than would be possible using an interpreter.

A special feature of incrementally compiled programs is that they can be incrementally decompiled, back into their original source form; this allows full support not only of assert/1 (compilation), but also clause/2, retract/1, and so forth. Apart from this flexible run-time support, incremental compilation also fully supports the WIN-PROLOG debuggers.

Hashed Compilation: Instant Access

The term "hashed compilation" is a slight misnomer, as hashing is actually a reversible post-process applied to incrementally compiled programs. Introduced in version 4.200 of WIN-PROLOG, this feature allows the creation of a highly optimised hash table for any incrementally compiled predicate that has no variable or variable-headed structure cases in its first argument. The benefit of hashing on small to medium sized relations is relatively slight, but it really comes into its own on large relations containing thousands, or even millions, of clauses.

When an incrementally compiled predicate is hashed, each of its unique first arguments is counted and built into a table with a user-defined amount of headroom, specified by a "fudge factor", or percentage excess. The greater the fudge factor, the fewer the number of mishits are likely to occur during hashing, resulting in a play-off between efficiency and memory requirements. A default setting of 100% gives excellent performance in most circumstances. Because hashing is reversible, it is possible to try different settings on any given relation quickly to find out an optimal fudge factor. Although a hashed relation cannot be modified with assert/1 or retract/1, it is still fully accessible through clause/2, and so forth; furthermore, because hashing can be removed as easily as it has been added, a relation revert to its incrementally compiled status for any such modifications, before (optionally) rehashing. Hashed compilation also fully supports the WIN-PROLOG debuggers.
Optimised Compilation: Relation by Relation

The term "optimised compilation" is used to describe a process where each entire relation (collection of clauses for a given predicate and arity) is compiled into a single piece of code. In WIN-PROLOG, optimised compilation performed by a program is written in Prolog which, although somewhat slower at compiling than the incremental compiler, carries out sophisticated analysis on data and control flow within the program, eliminating redundant data transfers and optimising data traffic in general. The optimising compiler can perform multiple argument indexing, and special high speed jump instructions can convert tail recursion into a conventional program loop. Data relations do not normally benefit from being optimised, unless multiple argument indexing is used, but program relations are invariably faster and more space efficient.

Because the optimised compiler converts an entire program relation into a single, monolithic piece of executable code, it is not able to support assert/1, clause/2, retract/1, or the WIN-PROLOG debuggers. On the other hand, since optimised code cannot be decompiled, it provides a level of security for program source code.

First Argument Indexing

The argument indexing used in WIN-PROLOG is fairly extensive, but there are some minor differences between the incremental, hashed and optimised compilers in this respect. Firstly, the incremental compiler handles a few more indexing cases than the hashed compiler, which in turn handles a few more than the optimising compiler; secondly, on the other hand, the optimising compiler can index on multiple arguments, which the incremental and hashed ones cannot.

All three compilers index on the type of the first argument; further, if the argument is an atom or an integer, they index on the value. When the first argument is any kind of compound term (tuple, list, conjunction or disjunction), the incremental compiler indexes on the type of its first element, and if this is an atom, on the value. The optimising compiler only performs indexing on the value of an atom in the first element of a compound term, and does not index on the other types when at the head of a term. The following table summarises the indexing capabilities:

<table>
<thead>
<tr>
<th>Indexing Type</th>
<th>Inc</th>
<th>Hsh</th>
<th>Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument type</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>argument atom value</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>argument integer value</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>tuple head type</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>tuple head atom value</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>list head type</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>list head atom value</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>conjunction head type</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
The Comparison: Head to Head

The three compilers in WIN-PROLOG provide complementary, rather than competing services; in an application, it will typically be desirable to use a mix of all types of compiled code. A key consideration is that both incremental and optimising compilers implement indexing through a form of case statement, where successive table entries are checked sequentially, while the hashed compiler implements a true hash table that can give direct access to the clause required. In small or medium relations, the performance improvement is fairly small, and may be offset by other considerations (such as the desire to modify the data relation dynamically, or to disguise the source code by optimising it); in large data relations, the improvement offered by hashing can be very dramatic. The following table summarises the main features and properties of the three compilers:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Inc</th>
<th>Hsh</th>
<th>Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>suitable for small data relations</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>suitable for medium data relations</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>suitable for large data relations</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>suitable for fast program relations</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>supports assert/retract</td>
<td>yes</td>
<td>*</td>
<td>no</td>
</tr>
<tr>
<td>supports clause/decompile/listing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>supports debuggers</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>built into run time kernel</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>provides source code security</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

* because it is possible to remove and reapply hashing at any time, a hashed relation can be modified simply by converting it back into an incremental relation, performing the desired updates, and then rehashing it.
Appendix N - All About Hashing

This appendix discusses the details of hashed compilation in WIN-PROLOG, and explains how it enables large Prolog database relations to be indexed and accessed with extreme speed and efficiency.

Incremental and Optimised Compilation

Internally, WIN-PROLOG has traditionally supported two types of compilation, namely "incremental" (clause by clause) and "optimised" (predicate by predicate). The former type has the benefit of interpreter-like flexibility, fully supporting assert/1, retract/1, clause/2, listing/1 and the debugger; the latter type has the edge on speed, with "logical" code running some 2.5-3 times faster than its incremental equivalent, but at the cost of being out of reach of assert/1, retract/1, clause/2, listing/1 and the debugger. Both the incremental and optimising compilers support a form of "first argument indexing", in which a case statement is used to locate clauses that match on type, and in some cases the value, of a call's first argument. The relative virtues of these forms of compilation are discussed more fully in Appendix M.

Hashed Compilation

New to version 4.200 of WIN-PROLOG is a third form of compilation, in which an incrementally compiled relation is subsequently "hashed", providing direct access to clauses based on true first argument indexing. This indexing is not to be confused with the "case analysis" supported by the incremental and optimising compilers, but rather is a direct "calculate and hit" mechanism. The main benefit of the hashed approach is potentially increased speed in applications containing in-line databases (large relations composed, typically, of a series of Prolog facts), and hashing does not interfere with clause/2, listing/1 and the debugger, though for reasons that shall become obvious, it puts a relation out of reach of assert/1 and retract/1.

Hashing provides direct access to the first (or subsequent) matching clause, independent of the size of the clause database, so it can find the one-millionth clause just as quickly as the first. However, its benefits only really come into play when there is a significant number of clauses in any given relation. Moreover, although it is equally useful for indexing "facts" and "rules", it is unable to process relations where one or more clauses contains a variable, or a structure whose head is a variable, in its first argument.

All of the above suggests that hashing is ideal for providing fast access to large database relations, such as name/address information, or the huge lexicons of an application such as WordNet, but is not a general purpose compiler. In particular, its benefits are small or even nonexistent when applied to relations containing fewer than, say, about 100 clauses, and even if the first arguments are hashable, it will not make traditional "logical" code run any faster than the incremental compiler.
First Clause Fast, Next Clause Instant

There is a subtle consideration to content with examining the benefits of hashing over incremental compilation: as well as finding the first clause very quickly, any "next" clause (to be used in backtracking) can be found instantaneously. With the incremental compiler, case analysis is used to find the first matching clause, but it is also used to find any next clause during the decision about whether or not to create a backtrack stack entry. In the worst case, this can mean searching an entire data relation for a nonexistent second case, even when the required case was found quickly. With the hashed compiler, all clauses for any given first argument are directly linked.

Hashing Considerations

For a program to be hashed, it must conform to certain requirements. Any data relation, from a single clause to many millions of clauses, can be hashed, provided none of its clauses contains a variable, or a tuple, list, conjunction or disjunction whose head is variable, as the first argument. This immediately rules out one common form of program structure, where a "catch-all" clause is added to the end of a relation, for example:

\[
\text{double}(\text{X}, \text{Y}) :- \\
\text{Y} \text{ is } \text{X} \times 2.
\]

In order to support hashing, programs of this kind should be rewritten as something like:

\[
\text{double}(\text{X}, \text{Y}) :- \\
\text{double_data}(\text{X}, \text{Y}).
\]

\[
\text{double}(\text{X}, \text{Y}) :- \\
\text{Y} \text{ is } \text{X} \times 2.
\]

\[
\text{double_data}(\text{1}, \text{2}).
\]

\[
\text{double_data}(\text{2}, \text{4}).
\]

\[
\text{double_data}(\text{3}, \text{6}).
\]

...
Dynamic, Multifile and Volatile Predicates

Although riding on the back of the incremental compiler, hashing a predicate forces that predicate to become static: otherwise it would be necessary to recompute and possibly reallocate the hash table each time a clause were added or removed. Three declared classes of predicate, namely those which are dynamic, multifile, and volatile are therefore unsuitable for hashing. This means, in effect, that the only predicates that can be hashed are those which are static, single-file non-volatile definitions, which also contain no clauses with variables or variable-headed structures in their first arguments.

Automatic Hashing - Prolog Flags

Under normal circumstances, version 4.200 of WIN-PROLOG automatically hashes suitable predicates after consulting a source file, provided they obey all the rules described above. This new behaviour can be turned on or off, and also "tuned", using three new Prolog flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_files</td>
<td>sets automatic hashing on or off</td>
</tr>
<tr>
<td>hash_quota</td>
<td>defines the hash table quota</td>
</tr>
<tr>
<td>hash_limit</td>
<td>defines the minimum clause limit</td>
</tr>
</tbody>
</table>

The first of these flags is a simple toggle, which can take the values "on" or "off". When set to "on" (the WIN-PROLOG default), then an attempt will be made to hash any predicates loaded from a source file immediately after the file is consulted; when set to "off", automatic hashing is not attempted. The other two flags define integers (stored as Prolog atoms) which control the "quota" and "limit".

The quota is a percentage which defines how much space is allocated in the hash table: its default, of '100', allows 100% more space in the table than is strictly necessary. Smaller quotas result in smaller tables, but at the expense of more hash "mishits" (where a computed hash offset finds a slot already occupied by another value), which can slow down performance; large quotas result in larger tables, where performance might be optimal, but at the expense of wasted memory. This will be discussed in greater detail below.

The limit is simply a clause count below which it is deemed not to be worth hashing a relation: its default, again of '100', is a reasonable setting, but this can be changed as required to adjust the partition between those data predicates which want to be hashed and those that don't. Typically, smaller relations benefit less from hashing than larger ones, and the limit value can be used to define the "sweet spot".

Manual Hashing - hash/n and nohash/n

Whether or not automatic hashing is enabled (controlled by the Prolog flags, hash_files, hash_quota and hash_limit), it is also possible to hash predicates directly, using the hash/1 and hash/3 predicates. The former predicate simply picks up the existing hash_quota and hash_limit values, before calling hash/3: this latter predicate allows the user to specify quota and limit settings directly, and independently of their equivalent Prolog flags. Two corresponding pred-
icates, nohash/1 and nohash/3 allow hashed predicates to be de-hashed, returning, in the latter case, their previous quota and clause count values.

The main purpose of manual hashing is to allow specific predicates to be hashed with different quota and/or limit values to the general default, for reasons that will be explained later. De-hashing must be carried out before a relation is hashed again with or without different quota and limit parameters, in order to avoid an Error 67, Predicate Protected error. Please note that if a relation contains one or more clauses with a variable or variable-headed structure as its first argument, an Error 22, Instantiation Error will be reported by hash/1 and hash/3.

**Hashed Predicates - Understanding the Statistics**

When any predicate is hashed, with a given quota, its clauses are first counted, followed by its unique first argument cases. A hash table is then allocated, with its number of slots defined by the case count and the given quota, which is treated as a "percentage overhead". In theory, a quota of zero could be used, in which case the hash table would contain exactly one slot per unique first argument, and in certain cases, this would work extremely well. More often than not, however, there would be clashes, where one or more unique first arguments map to the same slot in the table. When one of these "mis-hits" happens, an algorithm is called to rehash to a next-choice slot. If this is also occupied, the algorithm is applied again, as often as is needed to find a spare slot. In a worst-case scenario, it might be necessary to try all but one of the table's slots before finding the remaining spare one.

By allowing the hash table to be larger than strictly necessary, two things happen: firstly, because of the increased number of slots, fewer hash clashes will occur, and secondly, when the "try next" algorithm is applied to a mishit, there is a better chance of finding a spare slot sooner rather than later. The more spare space that is in the table, the fewer are the total mishits that should occur.

Even with the best hashing algorithm in the world, and with a generous quota of extra space, there will be times when individual data sets hash "badly". This is an inevitable result of the modulo arithmetic used to assign hash values to a table, and the only way round this is to adjust the table size. Because the quota can be defined directly in a call to hash/3, it is easy to experiment to find the best setting for any given relation.

If the "print_message" Prolog flag is set to "informational" or "help", then when a relation is hashed, a simple message will be displayed, as in the following case, in which one of the example programs, "salesman.pl", is consulted:

```
| ?- consult( examples(salesman) ).
<enter>
# 0.000 seconds to consult salesman.pl [c:\tempwin\examples]
% 0.000 seconds to hash predicate dist / 3
% Quota = 100, Count = 300, Cases = 24, Total = 6, Worst = 3
yes
```

The "Quota" field here says that the table was allocated with 100% extra space, while "Count" shows that there were 300 clauses in the dist/3 relation. The
next item, "Cases", shows that there were 24 unique first arguments (in this case, the names of the 24 towns used in the example program). The "Total" value shows that, in allocating the 24 unique cases, there was a total of 6 mishits, giving an average of 6/24 (ie 0.25) attempts to find the correct slot, while 'Worst' shows that, in the worst case, it took 3 attempts to find a slot.

Ideally, Total and Worst should both be zero, but small values don't really matter. In particular, if the "Total/Count" ratio is small, hashing can be considered to have been successful: remembering that there are 300 clauses in this example program, a worst case using the incremental compiler's case-based "indexing" could take 299 attempts to find the correct clause, with a mean of about 150 attempts, which is a lot worse than the values of 3 and 0.25 achieved with hashing.

**Memory Requirements**

The memory requirements for hashing are really quite small: minimally, a 4-byte slot is required for each unique first argument case. Things are never quite that simple, of course, and things such as the Quota field and the need for working space during the hashing process itself, should be taken into account.

Hashing is a three-pass process: the first pass is simple, as all it does is count the total number of clauses in a relation, and check that each one has a first argument that is neither a variable nor a structure whose head is a variable, which are the two cases which cannot be hashed uniquely. Once the predicate has been vetted, and its clause "Count" returned, memory must be allocated for a temporary hash table used by the second pass, in which unique first argument "Cases" are counted. In a worst case, where every clause has a distinct first argument, the minimum size for this table would be one slot per clause, but to improve efficiency, the user-definable Quota is applied to add some headroom, using the following formula:

\[
\text{Slots} = (\text{ip}(\text{Count} + (\text{Count} \times \text{Quota} / 100)) + 1) \lor 1.
\]

The "+1" part of the equation ensures that, even with a Quota of zero, there is always at least one spare slot, and the "\lor 1" (inclusive or) part ensures that the result is odd, which helps distribute hashed values during modulo arithmetic.

The clauses of the predicate are scanned a second time, applying a hash function to each one's first argument and "marking" the appropriate entry in the table. If a new first argument is discovered, a count is incremented; if there is already a clause with that argument, nothing is done. At the end of this second pass, the unique "Cases" value is known, and the memory occupied by the temporary hash table is released, before another, usually smaller table is created, this time with its size dependent on the Cases value, again using the Quota:

\[
\text{Slots} = (\text{ip}(\text{Cases} + (\text{Cases} \times \text{Quota} / 100)) + 1) \lor 1.
\]

The third pass resembles the second, only this time the table is loaded with the addresses of each first case of a clause matching a given first argument, and the clauses themselves are chained to point directly to the next case, if any, for the given argument.
In the example above, in which the dist/3 predicate was hashed, the memory requirements (in slots) for the three passes were as follows:

<table>
<thead>
<tr>
<th>Pass</th>
<th>Formula</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;none&gt;</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(ip(300+(300*100/100))+1)/1</td>
<td>601</td>
</tr>
<tr>
<td>3</td>
<td>(ip(24+(24*100/100))+1)/1</td>
<td>49</td>
</tr>
</tbody>
</table>

Because each table slot requires 4 bytes, this example required 4*601, or 2404 bytes during processing, but ended up using just 4*49, or 196 bytes.
Appendix 0 - The Console Input Model

This appendix explains the Console Input Model, first implemented in version 4.200 of WIN-PROLOG, in conjunction with the adoption of Rich Edit 3.0.

Introduction

In versions of WIN-PROLOG prior to 4.300, the whole of the Console window was editable at all times. This allowed the user to modify not just previous input, but also output, and had the overall result that at any given time, the console probably did not contain a true record of what had happened.

In WIN-PROLOG 4.300, editing is confined to a special region, called the "Input Zone", which effectively consists of the portion of the console window that follows the most recent prompt or program output. Once entered, keyboard input becomes locked along with any subsequent output, as the Input Zone is re-established at the very end of this output.

While any part of the console can be copied into the input zone for re-editing and re-entry, a much more useful mechanism for recalling commands is available. WIN-PROLOG has a powerful command history mechanism, that automatically stores up to 255 previously entered commands, even between sessions, which can be recalled and edited simply by using the <ctrl-up> and <ctrl-down> key combinations.

Some Basic Terminology and Concepts

The WIN-PROLOG 4.300 console can be thought of as being divided, invisibly, into two zones:

<table>
<thead>
<tr>
<th>Zone</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Zone</td>
<td>contains all console (&quot;user&quot;) output, as well as previously entered commands, and cannot be edited</td>
</tr>
<tr>
<td>Input Zone</td>
<td>is the bottom part of the console, into which new commands and other input can be typed, edited and entered</td>
</tr>
</tbody>
</table>

The two above two zones are best explained graphically; in Figure 1, the area of the console window with a dark grey background corresponds to the "Fixed Zone", while the area with the white background shows the "Input Zone".

Figure 1: The Initial Fixed and Input Zones
As a command is being typed into the Input Zone, it can be edited and modified at will; once <enter> is pressed to submit the command, the Fixed Zone expands to include the command, as well as any subsequent output, and once WIN-PROLOG is ready for more input, a fresh Input Zone is defined, as shown in Figure 2.

Please note that in both Figure 1 and Figure 2, the dark grey used to indicate the Fixed Zone is illustrative only: in WIN-PROLOG itself, the Fixed and Input Zones are not distinguished by any visible means.

Three Ways to Re-enter Commands

There are three ways in which you can re-enter previous commands, without or without edits and modifications, using the "Console History", "Highlight and Enter" or simply "Copy and Paste" features.

Console History Mechanism

By far the simplest and most powerful is to use the "Console History" mechanism, which records up to 255 of your commands in a special buffer. You can review these commands with two key combinations:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ctrl-up&gt;</td>
<td>Copy the previous (next, earlier) command into the Input Zone</td>
</tr>
<tr>
<td>&lt;ctrl-down&gt;</td>
<td>Copy the following (next, later) command into the Input Zone</td>
</tr>
</tbody>
</table>

When you’ve found the command you want to re-enter, just press <enter>, or optionally make modifications first. You can review the console history at any time you are performing input: you can always return to the last line you were actually editing simply by pressing <ctrl-down> repeatedly until it appears.

A key feature of the Console History mechanism is that your commands can be saved between WIN-PROLOG sessions, in a file in the "cache directory", which defaults to "prolog(cache)". When you start a future WIN-PROLOG session, <ctrl-up> and <ctrl-down> are ready immediately to review and re-enter your commands from the previous session.
Highlight and Enter

The second way in which to re-enter a command is simply to highlight its text using the mouse, and then press <enter>. At any time text is highlighted, pressing <enter> simply copies that text directly into the Input Zone, ready for editing, and eventual re-entry. The rules about the <enter> key are simple:

<table>
<thead>
<tr>
<th>Text selection</th>
<th>Action of &lt;enter&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 chars</td>
<td>submits the entire Input Zone to WIN-PROLOG</td>
</tr>
<tr>
<td>1 or more chars</td>
<td>copies highlighted text to the Input Zone, replacing its previous contents</td>
</tr>
</tbody>
</table>

Note that the Highlight and Enter model allows any console text, not just previous commands, to be selected and copied to the Input Zone. Furthermore, the area highlighted is not limited to the Fixed Zone: it can be entirely Fixed Zone, part Fixed and part Input Zone, or even just Input Zone. The last case is useful if, for example, you want to trim a command before entering it: simply highlight the bit you want to keep, and press <enter> twice - the first <enter> will replace the overall Input Zone with the stuff you highlighted (second case above); the second <enter> will submit it to WIN-PROLOG (first case).

Copy and Paste

The third, and most familiar way in which to re-enter text is simply to use "Copy" and "Paste". These are available either through the "Edit" menu, or the usual Windows shortcut keys (<ctrl-C> for "Copy", <ctrl-V> for "Paste". You can also use "Cut" (<ctrl-X>) on the Input Zone, but not on the Fixed Zone (that's why it's called the "Fixed" zone!). Remember also that you can only "Paste" into the Input Zone (for the same reason).

The only advantage of Copy/Paste is that you are not limited to the Console window for your source material: you can grab example queries from WIN-PROLOG program windows, or even PDF documentation files, for pasting into the Input Zone.

Related Keys and Menu Options

You can quickly identify and highlight the Input Zone by selecting the "Edit/Select Query" menu option; this has a matching keyboard shortcut:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ctrl-Q&gt;</td>
<td>Select (highlight) the entire Input Zone</td>
</tr>
</tbody>
</table>

Because the Fixed Zone is "Fixed", you cannot modify anything within it: moreover, neither can you delete text from it, with the result that the contents of the Console continue to grow throughout a WIN-PROLOG session. A special menu option, "Edit/Empty Console", allows you clear everything from the console and start over.
While emptying the console means that you will lose your ability to "Highlight and Enter" or even "Copy and Paste" previous commands, it does not affect the "Console History" mechanism: `<ctrl-up>` and `<ctrl-down>` can still recall up to 255 of your previous commands.

**Changing the Console History Depth**

Although the Console History can store up to 255 of your previous commands, this would require a fairly large chunk of memory (each command is stored as 16-bit Unicode, and could be up to 4096 characters long, meaning that 8Kb of RAM are reserved per desired command. By default, WIN-PROLOG allocates space for 64 saved commands, using 512kb of memory. If you want to keep fewer, or more, commands, you can change the "depth" of the Console History using the menu option, "Options/Console History", to any value between 1 and 255.

The same menu option lets you choose whether or not to save the Console History between sessions: you might decide not to save commands in this way if, for example, you have been entering passwords or encryption keys, and don't want to risk someone else finding them subsequently.

**Programmable Control of Console and History**

The console itself, together with its console history mechanism, can be controlled programmatically using a variety of predicates; this section refers the reader to the most important of these.

The entire console can be read (as text) using `wtext/2`, and also set (as text) with the same predicate. Thus the simplest way to clear the console programmatically is to make the call:

```
| ?- wtext( (1,1), ` ` ).  <enter>
<console is cleared>
yes
```

The Console can also be read and written using the `wrchtxt/3` predicate, though it should be noted that setting of text is possible only within the Input Zone, or for the entire console window.

The `history/1` predicate provides a means both to set the desired history depth, as well as to read or write the entire history as a Prolog structure. The following call, for example, sets the history depth to 32:

```
| ?- history( 32 ).  <enter>
yes
```

One of the "Prolog Flags" (see `prolog_flag/2`) governs an important feature of the console history mechanism: the "save_history" flag determines whether or not the console history is saved between sessions. Its default setting is "on", meaning that up to 255 of your commands (the exact number depends
on the setting from the most recent call to history/1) will be saved in the file, PRO386W.HST, between WIN-PROLOG sessions, which might constitute a security risk if you have been typing in commands containing passwords or other secret information. To disable the saving of commands between sessions, you can make the following call:

| ?- prolog_flag( save_history, _, off ). <enter>  
yes
Appendix P - Windows Sockets and TCP/IP

This appendix explains the built-in handling of Windows Sockets (Winsock), first implemented in version 4.600 of WIN-PROLOG, which provides a simple and powerful way to program TCP/IP network and Internet applications.

Introduction

Originally, "Sockets" was a concept developed in 1983 by the University of California at Berkeley, as a licensed API which gave Unix programmers the ability to program for early networks. Simple function calls would allow programs to set up and maintain connections across networks, and transmit and receive data.

The functions contained in this early system, know usually referred to as "Berkeley Sockets", still persist today in modern Windows operating systems, but have been extended and refined in a number of important ways, resulting in a new moniker, "Windows Sockets", or simply, "Winsock".

As with any system which emerged in one environment (BSD Unix 4.2), before migrating into and evolving in another (Windows), Winsock has become increasingly complex, with numerous ways of achieving any one goal, as well as equally many ways of getting thoroughly stuck, and crashing applications or even the operating system. In WIN-PROLOG, a lot of effort has been taken to simplify Winsock and render it "safe", with a minimal loss of functionality.

What is a Socket?

Before getting any further into this discussion, it will be useful to answer one simple question: "What is a socket?". Happily, the answer is simple. Effectively, a socket is nothing more than a local data structure which contains the information required to "connect" to another socket on some other computer across a network. So, for example, machine "A" creates a socket, which initially does nothing. Elsewhere, machine "B" does the same. Two machines, each with a data structure that is sitting somewhere in memory.

In order to get the two machines to communicate, one or other of the machines attempts to connect its socket to the other. So, for example, "A" says, "connect my socket to B". Machine "B" is notified of this request, and is invited to "accept" (or reject) the connection. Assuming "B" accepts, then the two sockets, one on "A" and the other on "B", are connected. From now on, "A" can write ("send") data to its socket, and "B" can read ("recv") the same data from its socket; likewise, anything "B" sends to its socket can be "recv"d by "A". Eventually, when enough data is exchanged, "A" or "B" can elect to close the connection, and finally each machine will "close" its respective socket, which simply frees up some memory and related resources.

In theory, any one computer could have up to 65536 sockets simultaneously open, though in practice this is limited to fewer than 100. Each socket needs to be associated with a "Port", which is an integer in the range 0..65535. Some ports are, by convention, used for special purposes: for example, most HTTP (web) servers use port "80", while Telnet servers typically use port 23; other ports have no common uses, and can be freely assigned by applications. On any given computer at any one time, each port can only be used by a single socket attached to a single application.
The really nice thing about sockets is that they really are that simple, in principle. Additionally, with Winsock, an application can have numerous sockets simultaneously open, each communicating with another machine across the network or Internet. And yet, all that the application ever needs to do is set up the socket, make the connection, send and receive data, and then close things down.

**Blocking and Asynchronous Operations**

Many of the original Berkeley sockets functions operated in what is called "Blocking" mode: for example, a request to receive some data across the network would physically hang the calling application until that data arrived. Tests could be made prior to such a call to see if any data was ready for collection, but even so, it was a characteristic of early Sockets applications that they would regularly hang, sometimes requiring a system reboot.

Winsock, backed by the Windows message-loop model, introduced the concept of "Asynchronous" sockets: here, rather than (say) waiting to receive some nonexistent data, the application carries on with its other, normal activities. When some data arrives, the application is notified in a Windows message, just as it is when the mouse is moved, or the user types some input at the keyboard. In response to the message, the application can safely receive the data, storing or processing it as required, before returning to its other duties.

The asynchronous Winsock functions mirror the original blocking functions, but enable TCP/IP applications to be far more responsive and "crash-proof", albeit at the cost of some additional programming complexity.

**Stream, Datagram and Raw Sockets**

Another consideration with Sockets, whether of the Berkeley or Windows variety, is how they are connected. Three main types of socket exist; firstly, "Stream" sockets are designed for a persistent connection between two computers, and are "reliable": if machine "A" sends some data, machine "B" will receive it intact, in the exact sequence it was sent, or will be notified of a network error. Secondly, "Datagram" sockets are used to send short, one-off messages between machines, but without any guarantee that it will arrive in the same sequence that they were sent, or even that they will arrive at all: in fact, they can (and often do) simply disappear without anyone knowing. Finally, "Raw" sockets operate at an even lower level, and are beyond the scope of this discussion.

Stream sockets are the most widely used in most Internet applications, and form the basis of the "TCP" protocol; datagram sockets, because they lack the sophisticated error-checking and packet-ordering, pose a far smaller overhead on networks, and used to support the "UDP" protocol for such things as streaming media and online games, etc.

The built-in sockets predicates in **WIN-PROLOG** are exclusively of the connection-oriented, stream variety: that is not to say that UDP applications cannot be written, but doing so will require some simple calls to the *winapi/4* predicate. They are also based on the asynchronous model, utilising messages to notify the Prolog application when data can be read or written, or connections have been connected, or may be accepted or closed.
A Simple Example

The following program implements a minimal "HTTP" (web) client, designed to pull files down off the LPA web server. It requires just five sockets predicates; firstly, screate/2 is used to create a named socket, called "lpa", which attempts to connect to the server, "www.lpa.co.uk". Next, sstat/2 is used to check the connection status, after which ssend/2 is called to send a string, "GET /xxx~M~J~M~J", where "xxx" is the filename that was passed into the program by the user; srecv/2 is then used to retrieve the data, before sclose/1 closes the socket.

/* The Ultimate Simple Get URL 1 - Brian D Steel - 12 Mar 05 / 19 Sep 06 */

% create socket, wait until ready, send http request, read all data and close

get_lpa( File, Text ) :-
(  write( `GET /` ),
  write( File ),
  write( ` ~M~J~M~J` )
) ~> Http,
(  screate( lpa, `www.lpa.co.uk` ),
  repeat,
  sstat( lpa, Stat ),
  cmp( 1, Stat, 0 ),
  repeat,
  ssend( lpa, Http ),
  repeat,
  srecv( lpa, Data ),
  write( Data ),
  Data = ``
  ) ~> Text.

The only complications in the above program are the calls to repeat/0, and these are present for two reasons. Because screate/2 is designed to be asynchronous, the predicate returns even before the socket has been connected to the given server. Before this connection has been established, any attempt to send data with ssend/2 is likely to generate an error, so a repeated call is made to sstat/2 effectively to wait until the returned socket status indicates the connection is complete. Even then, it is possible that the socket is not ready for writing, so repeated attempts are made to call ssend/2.

The third call to repeat/0 is required for similar reasons: srecv/2 will fail instantly if no data has arrived, so this repeat ensures that the predicate is called over and again until it returns something; however, there is a second reason: srecv/2 may only return a part of the file, and needs to be called repeatedly.
until it returns an empty string, "``", which is the signal that there will be no further data.

Although the socket predicates, `screate/2`, `sstat/2`, `ssend/2`, `srecv/2` and `sclose/1` are "non-blocking" and designed for asynchronous programming, using `repeat/0` in the way illustrated above imitates the behaviour of a blocking application, by waiting until the functions succeed before proceeding. Unlike a true blocking application, at least the above can be interrupted by pressing `<ctrl-del>` should it appear to hang. Let's check it out now; type the command:

```
| ?- get_lpa( 'index.htm', Text ).
Text = `&lt;HTML&gt; ... &lt;/HTML&gt;M~M~J`
```

The file, "www.lpa.co.uk/index.htm" will be returned, "..." will depend upon its contents.

**An Asynchronous Example**

Now we'll look at a more detailed example which receives files asynchronously, reacting to socket messages that notify the application when certain events have occurred. In order to process the messages, we must direct them to a program of our own: the `socket_handler/2` predicate provides this mechanism, linking a socket name to an arity-3 predicate of our choice, here, `get_lpa/3`.

The first clause, `get_lpa/2`, takes a file name (as before) and a socket name, which is simply an atom of your choice, but preferably one that is not already in use: remember, this program works asynchronously, in the background, meaning that you could be running several downloads in parallel. A call is made to `socket_handler/2` to associated your socket with the `get_lpa/3` predicate, after which a socket is created using `screate/2`, just as before; this time, however, that's where the program stops.

The remaining three clauses are for the `get_lpa/3` "socket handler" predicate; each handles just one event. The first is called when a "sck_write" notification is received, indicating that the socket has connected and is ready to send data: a call is made to `ssend/2` to send the "GET /xxx~M~J~M~J", string, once again where "xxx" is the filename that was passed into `get_lpa/2` by the user; on completion, the handler just returns.

The second clause handles "sck_read", which is sent whenever some data has arrived, so `srecv/2` is used to receive the data. Remember that any one call to `srecv/2` might not return all available data, but there is no need to make a repeated or recursive call: if more data is available, another "sck_read" event will be generated automatically. When the remote socket closes down, an attempt to receive data will succeed, but returns an empty string (``): this is the only time such a string can be returned by `srecv/2`.

As each data fragment is received, it is simply asserted for later retrieval. There is no need for a catch-all clause: handlers can succeed or fail as they see fit, and even leave choice points behind: the low level socket manager simply succeeds anyway, performing a cut (/I/O) for safety.

The final clause handles "sck_close", which is initiated by the remote LPA web server after the file has been sent; all the data fragments stored in successive invocations of the "sck_read" case are collected up and glued together with `cat/3`, before a call is made to `sclose/1` to close the socket. In this example,
the final assembled data is simply written to the console window, but it could just as easily be saved in a file, or rendered in a browser.

/* The Ultimate Simple Get URL 2 - Brian D Steel - 12 Mar 05 / 19 Sep 06 */

get_lpa( File, Sock ) :-
    socket_handler( Sock, get_lpa ),
    dynamic( Sock/1 ),
    dynamic( Sock/2 ),
    time( Tops, Divs ),
    assert( Sock(File) ),
    assert( Sock(Tops,Divs) ),
    screate( Sock, `www.lpa.co.uk` ),
    (  write( `Getting ` ),
        write( File ),
        write( ` on ` ),
        write( Sock ),
        write( `~M~J` )
    ) ~> user.

% on sck_write, send a simplified HTTP request

get_lpa( Sock, sck_write, 0 ) :-
    Sock( File ),
    dynamic( Sock/1 ),
    (  write( `GET /` ),
        write( File ),
        write( `~M~J~M~J` )
    ) ~> Http,
    ssend( Sock, Http ).

% on sck_read, receive and assert a single segment of data

get_lpa( Sock, sck_read, 0 ) :-
    srecv( Sock, Data ),
    Data \= ``,
    assert( Sock(Data) ).
% on sck_close, gather all saved data and concatenate it

get_lpa( Sock, sck_close, 0 ) :-
    findall( Data, Sock(Data), List ),
    cat( List, File, _ ),
    dynamic( Sock/1 ),
    assert( Sock(File) ),
    sclose( Sock ),
    ( retract( Sock(Tops,Divs) )
    -> time( Tops, Tick )
    ),
    Time is Tick / Divs,
    socket_handler( Sock, socket_handler ),
    ( write( `Socket ` ),
    write( Sock ),
    write( ` done in ` ),
    write( Time ),
    write( ` seconds~M~J` )
    ) ~> user.

An HTTP Server Example

So far, we've only looked at sockets in one type of application, where we are a "client" on the network, requesting services from a "server". However, thanks to the clean design of the sockets predicates, it is just as simple to write a server application in WIN-PROLOG. The next example implements a simple HTTP (web) server, which is powerful enough to serve up an entire static (ie, HTML-based, rather than script-driven) website. Thanks to the asynchronous nature of the socket predicates, it can serve multiple simultaneous clients, as well as sending out multiple files simultaneously to any one client.

Until now, we've been calling screate/2 with a "domain name", such as "www.lpa.co.uk", encoded as a WIN-PROLOG string, which causes it to look this name up to convert it into an "IP address", before attempting to connect to the HTTP (web server) port, number 80, on the resulting machine. Had we wanted to, we could also have used the "IP address" instead, again encoded as a string. We could have even have gone a stage further, and specified the port number, in case we wanted to connect to some other service. Thus, assuming that the domain "www.lpa.co.uk" has an IP address of "217.207.1.226", the following calls are all equivalent:

| ?- screate( foo, `www.lpa.co.uk` ).
| ?- screate( foo, `217.207.1.226` ).
?- screate( foo, (`www.lpa.co.uk`,80) ).

?- screate( foo, (`217.207.1.226`,80) ).

The key to writing a server, rather than a client, lies in a second use of `screate/2`; first, we need to create a special type of socket which, rather than connecting and sending/receiving data, simply "listens" for incoming connection requests: for an HTTP (web) server, we should, by convention, listen at port number 80. All we need to do is called `screate/2` with this integer as its second argument, rather than using a string, for example:

?- screate( foo, 80 ).

In this instance, the socket, "foo", will simply wait for asynchronous connection requests at the local machine's port number 80. If and when some other machine attempts to connect to this port, a "sck_accept" message will be received, and this should be serviced by a third and final use of `screate/2`. In response to this message, we need to create a new socket (its name does not matter), and link it to "foo", for example:

?- screate( bar, foo ).

will create a new socket, "bar", which accepts a connection from another machine on behalf of the socket "foo". This connection will appear to the outside world as being on port 80, but will use another, automatically-assigned port locally: we don't need to worry about this.

When the remote machine sends data, we receive it through "bar", not "foo"; likewise, when we want to send data to the remote machine, again we write it to "bar", not "foo". The reason for this use of a new socket for accepting each incoming connection is so that the server can remain actively listening for accept requests, while any number of connections can be active simultaneously.

Now let's look at the web server example to see how this all works out in practice. The first predicate, http/0, simply calls `socket_handler/2` to associate the socket name, "http" to use the handler, http/3, and then calls `screate/2` to create a listening socket called "http" at port "80": that's it!

The socket handler, http/3, has four clauses. The first handles up "sck_accept", and uses `hide/2` to generate a unique atom to use for the new server socket name; it assigns http/3 to be the handler for this new name with `socket_handler/2`, before creating the socket with `screate/2`, using the listening socket's name (passed into the handler as the variable, "List") as the second argument. Again, that's all for now.

The second clause of http/3 picks up the "sck_read" event, which expects an HTTP request: such requests contain a variety of pieces of information, among which should be the command, "GET xxx~M~J~M~J", where "xxx" is the file name that has been requested by the remote application, generally a web browser. This clause uses `find/3` to search for "GET ", and then calls the predicate a second time to search for a trailing space, copying the intermediate characters into a string, "File". This is then concatenated, using `cat/3`, to a directory name (here, "/htmlpa") where the website's HTML files reside, before being converted to an atom with `stratom/2`. A check is made to see whether this file exists, and if not, a simple "404" (file not found) message is stored in a memory file, created with `fcreate/5` using the name the same name as the socket; if the file does exist, then a "200" (file about to follow) message is sent, and `fcreate/5` is used to open the disk file, again using the name of the socket to refer to it.
The third clause gets invoked in response to the "sck_write" event, which simply says, "this socket can be written to". It responds by trying to send the first 1024 bytes of data, using `ssend/2`, from whichever file is named the same as the socket (this might be the 404 error message created above, or the actual file that is open). When it is possible to send more data, another "sck_write" event will be signalled, so there is no need to call `ssend/2` repeatedly. When no more data can be sent because the network is busy, this case simply fails, hopefully resume in due course in reaction to another "sck_write" event. Once the sending of the input file has been completed, the file (or memory file) is closed with `fclose/1`, and the corresponding socket is closed with `sclose/1`.

The final clause of the handler simply tidies up if the server socket is closed unexpectedly, for example by the user of a remote browser aborting a page or clicking on another URL before the existing page and resources have been loaded. On detecting "sck_close", this clause simply closes any file associated with the socket, together with the socket itself and any housekeeping assertions.

```prolog
/* The Ultimate Simple HTTP Server - Brian D Steel - 12 Mar 05 / 19 Sep 06 */

% set up an http server on port 80

http :-
    dynamic( http/4 ),
    socket_handler( http, http ),
    screate( http, 80 ).

% on sck_accept, create a server socket

http( List, sck_accept, 0 ) :-
    hide( Sock, 0 ),
    socket_handler( Sock, http ),
    screate( Sock, List ),
    time( Tops, Divs ),
    assert( http(Sock,0,Tops,Divs) ).

% on sck_read, read the HTTP request and locate the target file

http( Sock, sck_read, 0 ) :-
    srecv( Sock, Text ),
    ( write( Text )
    ) ~=> user,
    ( find( `GET `, 0, Find ),
```
find( ` `, 2, _ ) \rightarrow \text{File}
\)
\text{Find} \not= ` `, \\
\text{cat( `/htmlpa`,File], Path, _ )}, \\
\text{stratm( Path, Atom )}, \\
\text{file( Atom, -1, Flag )}, \\
( \text{Flag} = 0 \\
\rightarrow \text{fcreate( Sock, [], 0, 5, 0 )}, \\
\text{wintxt( Sock, 7, 1, `404~M~J~M~J` )} \\
; \text{fcreate( Sock, Atom, 0, 0, 0 )} \\
), \\
( \text{retract( http(Sock,0,Tops,Divs) )} \\
\rightarrow \text{assert( http(Sock,1,Tops,Divs) )}, \\
\text{ssend( Sock, `HTTP/1.0 200 OK~M~J~M~J` )}, \\
\text{http( Sock, sck_write, 0 )} \\
).

\% on sck_write, send a single segment of data

\text{http( Sock, sck_write, 0 ) :-} \\
\text{http( Sock, 1, _, _ )}, \\
\text{input( Input )}, \\
\text{input( Sock )}, \\
( \text{eof} \\
\rightarrow \text{input( Input )}, \\
\text{fclose( Sock )}, \\
\text{sclose( Sock )}, \\
( \text{retract( http(Sock,1,Tops,Divs) )} \\
\rightarrow \text{time( Tops, Tick )}, \\
\text{Time is Tick / Divs}, \\
\text{socket_handler( Sock, socket_handler )}, \\
( \text{write( `Output Complete on ` )}, \\
\text{write( Sock )}, \\
\text{write( ` in ` )}, \\
\text{write( Time )}, \\
\text{write( ` seconds~M~J` )} \\

\text{(% on sck_write, send a single segment of data}
Once compiled, this program can be run by typing the command:

| ? http.

Assuming that the directory specified in the above code contains the HTML files belonging to a website, you should now be able to connect your choice of browser to the machine. For example, running http/0 on a machine at IP address 192.168.1.2 on the local network, the Opera browser connects and appears as shown in Figure 1.

Meanwhile, on the console window of the copy of WIN-PROLOG running http/0, a
series of reports will be displayed showing the server activity as it continues:

| ?- GET /index.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows 98; en) Opera 8.01
Host: 192.168.1.2
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en
Accept-Charset: windows-1252, utf-8, iso-8859-1;q=0.6, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Connection: Keep-Alive

Output Complete on ebkbehxk
| ?- GET /white.css HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows 98; en) Opera 8.01
Host: 192.168.1.2
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en
Accept-Charset: windows-1252, utf-8, iso-8859-1;q=0.6, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Referer: http://192.168.1.2/index.htm
Connection: Keep-Alive, TE
TE: deflate, gzip, chunked, identity, trailers

Output Complete on yoezmnol
| ?- GET /but_pro0.gif HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows 98; en) Opera 8.01
Host: 192.168.1.2
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en
Accept-Charset: windows-1252, utf-8, iso-8859-1;q=0.6, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Referer: http://192.168.1.2/index.htm
Connection: Keep-Alive, TE
Other Predicates

As we have seen, most of the programming of Windows Sockets in WIN-PROLOG can be achieved using just four primary predicates: screate/2 creates client, server and listening sockets, ssrecv/2 and ssend/2 receive and send data to client or server sockets, and sclose/1 closes the sockets. Additionally, we have used sstat/2 to check socket status, and socket_handler/2 to direct events relating to any given socket to some code of our choice.

All that’s left to describe are some ancillary predicates: sdict/2 returns a list of all currently open socket names, while sckhdl/2 converts between socket a name and its raw, internal socket “handle”. This latter predicate is mainly used when calling other Winsock function through winapi/4, though it can also be used to check whether a given name is that of a socket.

Socket Events and Errors

There are a total of seven events which can be passed to a socket handler, whether this is the default handler (socket_handler/3) or one that has been written by the user and assigned to a socket using socket_handler/2. These are listed in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sck_read</td>
<td>1</td>
<td>Some data has arrived at the input buffer of a socket, and can be read by calling ssrecv/2 once: sck_read will be signalled again if more input is available</td>
</tr>
<tr>
<td>sck_write</td>
<td>2</td>
<td>The output buffer of a socket is ready to send data, and a data can/should be written by calling ssend/2 once: sck_write will be signalled again if more output is possible</td>
</tr>
<tr>
<td>sck_oob</td>
<td>4</td>
<td>An out-of-band message has been received: this is unlikely to occur, and should be ignored</td>
</tr>
<tr>
<td>sck_accept</td>
<td>8</td>
<td>An incoming connection has been requested on a socket, and this can be satisfied by calling screate/2 with two atoms</td>
</tr>
<tr>
<td>sck_connect</td>
<td>16</td>
<td>An outgoing connection has been created: this requires no immediate action, but should be followed by one ore more other events</td>
</tr>
<tr>
<td>sck_close</td>
<td>32</td>
<td>A socket connection has been closed remotely, and sclose/1 should be called to tidy up an data structures associated with it</td>
</tr>
<tr>
<td>sck_error</td>
<td>64</td>
<td>An error has occurred at the socket, and it is up to the application how to react; usually, the socket is closed with sclose/1</td>
</tr>
</tbody>
</table>

When a socket handler is called, the first argument is bound to the name of the socket (corresponding to the atom used in the first argument in the call to `screate/2`) that was used to create the socket. The second argument will be bound to one of the above names, and the third argument will contain an error value, which is an integer in the range 0..65535. Normally, this value will be zero, but if something has gone wrong, it will be an error number. Most Winsock errors are represented as integers in the range 10000..10093, with a few appearing in the range 11001..11004. The names and meanings of these error conditions are available in any Winsock documentation, or if you have access to one of the Microsoft SDKs, can be found in the file, WINSOCK.H.
Appendix Q - MIDI and Music

This appendix explains the built-in handling of MIDI (Musical Instrument Digital Interface), first implemented in version 4.700 of WIN-PROLOG, which provides a unified approach to the programming of musical research and utility applications.

Introduction

During the 1960s, a new type of experimental musical instrument was beginning to be developed, pioneered by such luminaries as the late Robert (“Bob”) Moog. These early analogue electronic synthesizers were hand-built, at great expense, from standard analogue components, such as valves, transistors and capacitors, and were monophonic: they could only play a single note at a time.

A typical instrument was composed of a large number of "modules", each of which performed some simple audio processing function, such as wave generation or low-pass filtering. The entire setup, or "Modular Synthesizer" as it became known, resembled an old-fashioned telephone switchboard, covered with "patch" cables routing audio signals between the modules, and was played, typically, by a piano-style keyboard which simply generated a control voltage that was in proportion to the distance of any given key from one or other end of the keyboard.

Apart from the audio signal routing between modules, and the control voltage from the keyboard that determined the pitch at which to play the resulting signal, there was no form of communication between these analogue synths and the outside world. Polyphonic performance was impossible, and recordings, such as Walter (Wendy) Carlos' seminal "Switched On Bach", were realised over many weeks and months of painstaking sound programming, retuning, and multitracking and overlaying single-line performances on analogue tape recorders.

Towards the end of the 1970s, digital computer technology was getting to the point where it could cheaply replace at least some of the discrete circuitry in synthesizers, and scanning keyboards began to replace the control voltage models of early synths, leading to instruments that no longer cost more than a three bedroom house, and which were finally polyphonic: affordable synthesizers which could play more than one note at once, made electronic music interesting and desirable to a wide audience of potential buyers.

Various manufacturers, such as Yamaha, Roland and Korg in Japan, began to ship large volumes of instruments to the mass markets, while traditional monophonic instruments, like the Mini-Moog, were becoming obsolete. Each keyboard was a self-contained instrument, able to reproduce a wonderful array of sounds, playing chords of up to 6 or maybe 8 notes at a time, but ultimately, was limited by the designs of its built-in, non-modular circuitry.

Advanced musicians began to collect keyboards, often from different manufacturers, in order to expand their sonic potential, and ended up with performance rigs containing anything up to half a dozen keyboards, arranged round the performer on a series of stands. While such rigs looked dramatic, and gave performers like Rick Wakeman a fantastic stage presence that could match any heavy metal drummer, the fact that a typical musician has only two arms, each with just five digits, meant that there were still limits to what could be performed with all this hardware.
The Birth of MIDI

In the early 1980s, a group of manufacturers, Roland and Yamaha among them, got together to design a digital interface that would allow not only the remote control of multiple synthesizers from a central source, such as a “master” keyboard, but would also allow full interoperability between devices from different companies. The interface was to become known as the Musical Instrument Digital Interface, or "MIDI", and this industry standard was to be overseen by a new organisation, called the MIDI Manufacturer’s Association, or MMA, which published the initial MIDI 1.0 specification in 1983.

Given the still relatively immature state of digital computer technology at the time (remember, this was in the days of the original 4.77MHz, 8088-based IBM PC), it was decided to standardise on a simple serial interface. Using widely-available, cheap, and most importantly, robust 5-pin DIN plugs, MIDI utilised an opto-isolated connector in order to avoid building earth loops which might otherwise cause hum and interference to the audio circuitry. Running at a mere 31.5 kilobaud, MIDI transmitted information about keyboard notes being played and released, or controllers being pushed, twiddled or blown, using a simple 3-byte message format. As no audio data was being transmitted, the relatively slow bit rate was adequate for most purposes, and had the added advantage that long cable runs, for example from an on-stage performer to a rack of synthesizers hidden somewhere backstage, could be achieved with simplicity and reliability.

MIDI and Computers

Pretty soon, it became increasingly apparent that computers could make good use of MIDI. Roland designed a module, the MPU-401, which could be installed in an IBM PC, providing it with the same opto-isolated 5-pin DIN connections that by now were becoming standard on synthesizers.

By recording note data on the computer, using a simple time-aware program that was to become known as a "sequencer", users could edit out mistakes, or add notes to play chords that were physically impossible to reach with the fingers of a hand. The computer-enhanced recording could then be played back, over MIDI, controlling a synthesizer or collection of synths, to create rich, complex performances.

Before long, companies were producing software which could display performance data not just as a list of numbers, but as a "piano roll" graphics representation, and later still, as rudimentary "classical" music notation. More and more synths were being sold as "modules", not to be confused with the modules that comprised the early analogue synthesizers; rather, these were entire instruments minus the keyboard, which could only be used via MIDI.

Serial, USB, FireWire and Soundcards

As computers continued to develop and grow in power and speed, the clunky 31.5 kilobaud serial connection began to feel like a bottleneck, but as the whole purpose of MIDI is to interface instruments from different manufacturers, including some that are now virtually "vintage" instruments, such as the Yamaha DX7 and Roland D50, the original standard has stuck. Instead, musical instrument builders began to look for alternative, additional ways to interface with computers, and came up with various schemes to use the ubiquitous RS-232 (Serial) ports found on most computers, and more recently, USB and FireWire. Many USB and FireWire interfaces utilise the high bandwidth of these ports to provide a number of logically distinct MIDI channels, often up to 8 or 16, each of which still runs at 31.5 kilobaud, but which work in concert to provide higher throughput. Entire MIDI studios can now be set up without a
single 5-pin DIN plug in sight.

As the growth in MultiMedia (MM) applications continued, special sound-processing hardware began to be added to computers as standard. Even some of the early "Sound Cards" contained a basic 4-operator, Frequency Modulation (FM) chip, the FB01, made by Yamaha, that could emulate basic musical instrument sounds, but later cards build in sample-based synthesizers that produced musical sounds by playing back pitch-shifted recordings of real instruments. All such cards respond to MIDI messages in much the same way as a dedicated stand-alone synth, and many also support some form of external MIDI interface, often by means of a cable which attaches to the joystick or games port.

**MIDI and Prolog**

One might ask, why MIDI and Prolog? The answer is in several parts, but first we'll look at the original motivation. Prolog, as a language, is excellent for writing applications with powerful pattern matching, and which exhibit elements of Artificial Intelligence; music is a form of art in which patterns are an integral component, at least in mainstream forms, and which requires some comprehension or understanding in order to be fully appreciated.

While simple sequencers are little more than time-stamped data recorders, Prolog at least offers the potential to write applications which process music at a deeper level. Whether attempting to analyse a given musical performance for style, or perhaps trying to harmonise a given melody, it seems, intuitively, that Prolog is better suited to the task than a lower-level language like C++ or Basic.

A second inspiration for MIDI in Prolog is to provide the bits and pieces required to write music-generating algorithms. There already exists a number of commercial applications, such as "Band-in-a-Box", which generate accompaniments, bass lines, and even melodies, but most of these are simple template-driven applications: "You want Jazz? Download the Jazz template!". Prolog has the potential to describe musical ideas declaratively, and then to generate the MIDI data to perform this music.

As well as the AI/Research concepts outlined above, a decent MIDI interface API also provides the Prolog programmer with the means to write simple utilities, such as System Exclusive librarians, arpeggiators, or perhaps a Pedal Harp emulator.

While **DOS-PROLOG** had a MIDI interface as long ago as 1989, and **micro-PROLOG** even before that, it has taken until version 4.700 for MIDI support to be built into **WIN-PROLOG**. Hopefully the above introduction and background has explained some of the rationale; now we will look at the specifics of the **WIN-PROLOG** MIDI implementation in greater detail.

**MIDI in WIN-PROLOG**

The implementation of MIDI in **WIN-PROLOG** is somewhat analogous to that of Windows Sockets (WinSock) and TCP/IP: just as in the latter case, named entities are defined which act as communications channels, across which data can be sent and received, and asynchronous messages notify a program when events have occurred or need to be processed.
Unlike TCP/IP, however, timing is critical to MIDI data. The exact time at which a note sounds, or stops sounding, must be recorded if there is to be any hope of replaying that note in its correct musical context on some later occasion, and timing is therefore at the heart of WIN-PROLOG's MIDI implementation.

When a MIDI device is created, by calling `mcreate/4`, a dedicated associated millisecond timer is set to zero and starts counting immediately. If the MIDI device is set up for input, then every time a MIDI event occurs (for example, a "Note On" message), it is timestamped to the nearest millisecond. Even if the message is "read" by Prolog considerably later, the `mrecv/3` predicate reports the actual event time as its second argument.

Named MIDI devices can support input or output, but not both, and if one is set up for output, no MIDI-generated events can occur. However, a dedicated timer is set up, as before, and is used to help the sending of MIDI data that might have been recorded earlier. When `msend/3` is used to send a message, the time specified in its second argument is compared with that of the dedicated timer: if the given time is greater than or equal to the running clock, the data is sent; if not, an event is scheduled to occur at the exact time the data can be sent.

The simple MIDI event model allows basic sequencer functions to be implemented with a minimum of code, while working in a "lazy" background way, allowing Prolog to be getting on with other jobs.

### Long and Short Messages

Unlike TCP/IP, where all data is simply text, and represented in WIN-PROLOG as strings, MIDI has two types of message. Short messages always comprise three bytes, starting with a "status" byte, and followed with a pair of "data" bytes. These messages indicate events such as "Note On", "Note Off", "Control Change", and so on, which make up the bulk of most MIDI traffic. The status byte is distinguished by having its high bit set to "1", and so will have a decimal value in the range 128..255 inclusive; data bytes always have this bit clear, and therefore will be in the range, 0..127 inclusive. Short messages are handled more or less identically across different models and makes of synthesizer.

Long messages are used to transfer bulk data, such as blocks of setup information, or even sound samples (very slow!), and are generally uniquely defined and interpreted by different models of instrument, and are therefore described as "System Exclusive", or SysEx messages. Each such message begins with a status byte whose value is 16'F0 (240), which is followed by one or more data bytes (all in the range 0..127). The message terminates with another status byte, this time 16'F7 (247), usually after some form of checksum has been transmitted. The format and content of individual SysEx messages is beyond the scope of this document, but all manufacturers issue MIDI Data Implementation charts, for any given instrument, which specify this information in detail.

Short messages are represented in WIN-PROLOG as a triplet of integers:

```
(Status,Data_0,Data_1)
```

The Status byte can be thought of as comprising two 4-bit nybbles: the high nybble will always be 1xxx, where "xxx" is between 2'000 (0) and 2'110 (6), while the low nybble will represent a number in the range 0..15 inclusive. The high nybble defines the message meaning - note on, note off, and so on,
while the low one defines which of 16 MIDI "Channels" it applies to. These channels are supported by "Multitimbral" MIDI instruments in order

Long messages are represented as Prolog strings.

Buffering MIDI Events

As mentioned earlier, the timing of MIDI events is critical to being able to record, analyse and/or play back the data with the correct spatial time references. In a simple implementation, this would mean eagerly consuming MIDI data in a dedicated, high-priority loop, recording the data together with time offsets as and when messages arrived; replaying the data would also require a tight loop, comparing recorded time offsets against a running clock, and sending out messages as the time advances to the right point.

Such programs would be fine in a single-tasking, MS-DOS style environment, but would be "bad neighbours" in the context of Windows. Instead, in WIN-PROLOG, MIDI messages are processed in the background, using interrupt-style events: as MIDI data arrives on an input device, it is automatically times-tamped and placed in a queue, so that it can be read at leisure. When sending MIDI data to an output device, a WIN-PROLOG program simply specifies the time at which it wants the data to be sent: if this time has been reached already, the data is sent at once; otherwise, an event is scheduled for when the data can be sent.

When an input MIDI device is created in WIN-PROLOG, using mcreate/4, two parameters are used to define how much data can be buffered. The default call, containing zero (0) for each parameter, allows for up to 8192 timed messages to be buffered, together with up to 64 SysEx messages of up to 1Kb in size, each. Longer SysEx messages are simply split into 1Kb chunks, and can be joined later by searching for the opening 16'F0 and closing 16'F7 status bytes. If it is desired to buffer less, or more, mcreate/4 accepts any other positive integers for these two parameters.

For output MIDI devices, event buffering is not applicable, and the first of the two parameters is set to "-1": in fact, this is the signal that a given device is to be used for output, rather than input. The SysEx parameter now represents a single buffer, defaulting to 64Kb, that can be used for sending a message. Larger messages can be sent either if the buffer is created with a larger size, of simply by splitting them into several shorter ones, each of which will fit within the declared buffer.

Creating a MIDI Device

To create a MIDI input device, a WIN-PROLOG program calls mcreate/4; the simplest call is as follows:

?- mcreate( midi_in, 0, 0, 0 ).
yes

The three numerical parameters refer to the Event buffer size (defaults to 8192 events), SysEx buffer size (defaults to 64 buffers of up to 1Kb each), and the "Device Index", which is simply an ordinal into a particular computer's table of MIDI input devices.
To create a MIDI output, the simplest call is this:

\[
\text{?- mcreate( midi_out, -1, 0, 0 ).}
\]

The only difference from the previous call is that we have used minus one (-1) for the Event buffer size: this is a special value that is simply interpreted to mean output, in which state no event buffering is required. The third parameter, zero (0), indicates a single SysEx buffer of the default 64kb in size, while the final parameter is once again a device index, this time referencing the computer's table of MIDI output devices.

The very instant either of these calls return, a dedicated millisecond-resolution timer, associated with the given Prolog MIDI device name (midi_in and midi_out above), starts counting up from zero. The value of this timer can be checked at any given point by calling `mtime/2`:

\[
\text{?- mtime( midi_in, Time ).}
\]

\[
\text{Time = 12345}
\]

This indicates that "midi_in" has been active for 12.345 seconds.

### Sending and Receiving MIDI Data

Each named MIDI "device" is set up, from the outset, for input or output, but not both. This is not to say that you cannot, for example, receive data and play it back on the same physical synthesizer, but rather, that you have to open two WIN-PROLOG MIDI devices, one each for input and output, in order to do so.

To send a "Note On" message to channel zero (0) of out MIDI output device, to indicate (say) that Middle-C (MIDI note "60") should be played with a medium velocity ("64"), we could type, for example:

\[
\text{?- mtime( midi_out, Time ),}
\text{msend( midi_out, Time, (144,60,64) ).}
\]

This call is, however, overly complex: `msend/3` is designed to fail if an attempt is made to send a MIDI message before the specified elapsed time has been reached on the given MIDI channel, but there is nothing to stop the programmer from simply specifying a very small time in order to guarantee the information is sent immediately; in fact, zero (0) is the best parameter to use when sending an immediate message:

\[
\text{?- msend( midi_out, 0, (144,60,0) ).}
\]
In this last example, the velocity has been specified as zero (0), which has the effect of cancelling the note that was played earlier. For some reason, most synthesizers use a "Note On" message with a velocity of zero (0), rather than a "Note Off" message, to cancel a note.

Assuming a keyboard or some other MIDI source is connected to our input device, we can attempt to retrieve some data with \texttt{mrecv/3}:

\begin{verbatim}
?- mrecv( midi_in, Time, Data ).
no
\end{verbatim}

Failure, as shown here, simply indicates that no data has yet been received. Let's suppose the user has now played and released Middle C on an attached keyboard; the next two calls will succeed:

\begin{verbatim}
?- mrecv( midi_in, Time, Data ).
Time = 1234 ,
Data = (144,60,64)

?- mrecv( midi_in, Time, Data ).
Time = 5678 ,
Data = (144,60,0)
\end{verbatim}

The values returned in "Time" are the offsets, in milliseconds, between the moment \texttt{mcreate/4} completed the creation of the MIDI device, and the moment that the respective MIDI event was signalled to the underlying \textsc{WIN-PROLOG} MIDI subsystem. These times are not related to the time \texttt{mrecv/3} was called, allowing "recording" programs to work at their own pace. By default, up to 8192 such events can be buffered, together with their time stamps, before a single call is made to \texttt{mrecv/3}, without loss of data: in practice, though most applications will more or less try to keep up with incoming data, especially if performing output at the same time.

\section*{A Simple Polling Sequencer}

Imagine a simple program for recording a sequence of MIDI data, with a view playing it back later. We could write something like this:

\begin{verbatim}
midi_record :-
    dynamic( midi_data/2 ),
    mcreate( midi_in, 0, 0, 0 ),
    repeat,
    (  mrecv( midi_in, Time, Data ),
        assert( midi_data(Time,Data) ),
    )
\end{verbatim}
fail
;  grab( _ ),
  mclose( midi_in ),
  !).

This program would initialise a database relation (midi_data/2), before creating a MIDI input device (midi_in) and entering a tight repeat loop. Each call to mrecv/3 would either fail immediately (because there was no data to receive) or succeed, allowing assert/1 to record the data together with its timestamp, before failing at the call to fail/0. The program will then backtrack to grab/1, which will check the (QWERTY) keyboard for a keystroke, and fail back to continue grabbing MIDI data if nothing has been typed. If the user presses any key on the (QWERTY) keyboard, the sequencer will stop recording, and the MIDI device closed by the call to mclose/1.

A corresponding one-shot MIDI playback routine could be written:

    midi_playback :-
      mcreate( midi_out, -1, 0, 0 ),
      midi_loop.

    midi_loop :-
      retract( midi_data(Time,Data) ),
      repeat,
      msend( midi_out, Time, Data ),
      !,
      midi_loop.

    midi_loop :-
      mclose( midi_out ).

Here, after creating a MIDI output device, a loop is entered during which each recorded MIDI event is retracted, before a repeat loop is entered during which msend/3 continually tries to send the timed data, failing back to the repeat when unsuccessful. As soon as the appropriate time is reached, msend/3 will succeed, and the loop is called recursively to process the next event. When no more events are left, a call to mclose/1 closes down the MIDI device.

This simply polling "sequencer" will work, but will be a CPU hog, and will prevent anything else running simultaneously in the same instance of WIN-PROLOG. A far better approach is to use a MIDI Event handler.
MIDI Events and Handlers

Again, like TCP/IP, the WIN-PROLOG MIDI implementation provides event notifications which can be used to drive a MIDI application asynchronously with respect to other processing. The \texttt{midi_handler/2} predicate can be used to associate a handler (an arity-3 predicate) with a given device name; this can (and usually should) be done before creating the device, for example:

\begin{verbatim}
?- midi_handler( midi_in, my_input_handler ).
yes
\end{verbatim}

This associates a handler predicate, \texttt{my_input_handler/3}, with a MIDI device called \texttt{midi_in}. The handler itself can be very simple; for example, to implement an event-based sequence recorder, the handler might be written as follows:

\begin{verbatim}
my_input_handler( Midi, _, _ ) :-
mrecv( Midi, Time, Data ),
assert( midi_data(Time,Data) ) .
\end{verbatim}

The actual recording program would now look something like this:

\begin{verbatim}
midi_record :-
dynamic( midi_data/2 ),
midi_handler( midi_in, my_input_handler ),
mcreate( midi_in, 0, 0, 0 ).
\end{verbatim}

As each MIDI event arrives, \texttt{my_input_handler/3} will automatically be called, and it will in turn call \texttt{mrecv/3} and \texttt{assert/1} to read and store the timed data. If more than one MIDI event is buffered, \texttt{mrecv/3} will automatically schedule a second event, so there is no need to check for this case or loop.

Playback is similarly straightforward: when \texttt{msend/3} fails because the specified time offset has yet to be reached, it schedules a MIDI event for the earliest, exact time when the call will work. Furthermore, if \texttt{msend/3} succeeds, it schedules an event immediately, so that, as with recording with \texttt{mrecv/3}, there is no need to loop:

\begin{verbatim}
my_output_handler( Midi, _, _ ) :-
( midi_data( Time, Data )
 -> msend( Midi, Time, Data ),
   retract( midi_data(Time,Data) )
).
\end{verbatim}
This picks up the first recorded event, without deleting it (yet), and attempts to send it with \texttt{msend/3}: if the sending works, the item is removed by \texttt{retract/1}, and a new event is signalled automatically. If \texttt{msend/3} fails, an event is scheduled for the earliest exact time when the call will work. The playback application now becomes:

\begin{verbatim}
midi_playback :-
    midi_handler( midi_out, my_output_handler ),
    mcreate( midi_out, -1, 0, 0 ),
    mtime( midi_out, 0 ).
\end{verbatim}

The only call here which might cause surprise is the last one, to \texttt{mtime/2}: like \texttt{msend/3}, this predicate will succeed if the specified time is less than or equal to the current elapsed time, and fail if greater; also like \texttt{msend/3}, it will schedule an event immediately if it succeeds, or at the specified time, if it fails. In the above example, it is simply used to commence the event chain that will drive the playback sequencer.

\section*{More on Timer Events}

Because music, and MIDI, is polyphonic (it contains multiple overlapping notes and events), in all but the simplest sequencing examples, as just described, it may be necessary to play back events that are not stored in any logical order. One example might be where an arpeggiator has added further delayed notes to a performance stream, and these arpeggios cascade around one another. Here, what we want to be able to do is send all those events that are ready for sending, while making sure an event is scheduled for the earliest subsequent event that needs to be sent.

The design of the MIDI timers in WIN-PROLOG is such that this is a relatively simple matter to support: successive calls to \texttt{mtime/2} or \texttt{msend/3}, whether they fail or succeed, will schedule an event which will occur at the earliest of the specified times. For example, let's suppose the MIDI timer for "midi_out" has only reached 1000; the calls:

\begin{verbatim}
?- midi_time( midi_out, 1234 ) ; midi_time( midi_out, 5678 ).
\end{verbatim}

and:

\begin{verbatim}
?- midi_time( midi_out, 5678 ) ; midi_time( midi_out, 1234 ).
\end{verbatim}

will both do exactly the same thing, namely fail, but schedule a MIDI event for the time, "1234", because this is less than "5678". In other words, when scheduling an event, both \texttt{mtime/2} and \texttt{msend/3} use an algorithm which compares any existing pending event with that which has been requested: if the existing event is sooner than the requested one, the existing event is retained and the requested one discarded; otherwise, the existing event is destroyed and the new one scheduled.

With this in mind, an output handler can be defined which will send all messages that are ready to go, while simultaneously scheduling its next call for
when the earliest subsequent message should be sent:

```prolog
my_output_handler( Midi, _, _ ) :-
    findall( (Time,Data),
             (  retract( midi_data(Time,Data) ),
     \+ msend( Midi, Time, Data )
             ),
        List),
    forall( member( (Time,Data), List ),
            assert( midi_data(Time,Data) )
        ).
```

The call to findall/3 returns a list of all MIDI items which were deleted by retract/1, but not successfully sent by msend/3, and the call to forall/2 simply puts these unprocessed messages back in the queue. A MIDI event will have been scheduled for the exact time the next message should be sent.

**More on Handlers**

As has been noted, a MIDI handler is simply an arity-3 predicate which receives and processes MIDI events. Here, we will look at the handler in a little more detail:

```prolog
handler( Midi, Time, Lapse ) :-
    ...  
```

The handler is called whenever an event occurs on an associated MIDI channel, with its three arguments bound as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Midi</td>
<td>Name if the MIDI device (always an atom)</td>
</tr>
<tr>
<td>Time</td>
<td>Time at which the handler was invoked, relative to the timer associated with the given MIDI device</td>
</tr>
<tr>
<td>Lapse</td>
<td>A &quot;lapse&quot; count indicating how many events have been received on the given MIDI device since the last time the handler was called</td>
</tr>
</tbody>
</table>

For most purposes, Time and Lapse can be ignored, but they serve as useful performance indicators, especially with respect to the real time behaviour of an application. For example, if a handler is lagging behind the incoming data stream, it is possible that multiple items will be received and buffered before the handler is finally invoked: this will be indicated by a lapse count that is greater than one (1). Likewise, if the first thing a handler does is call mrecv/3,
and the time reported in the handler's second argument is markedly different from that returned by \texttt{mrecv/3}, it gives a measure of the time lag between message receipt and processing.

Ideally, as far as possible, the "Time" parameter should be within a millisecond or so of any time returned by \texttt{mrecv/3}, while the "Lapse" parameter should be zero (0) or one (1).

With this in mind, and as with all event handlers in \textbf{WIN-PROLOG}, it is best to avoid long-winded computation within the body of a handler, and rather to use them to perform short, one-shot processing and/or data storage and retrieval.

As with other \textbf{WIN-PROLOG} handlers, the system contains a "default" MIDI handler, called \texttt{midi_handler/3}. For consistency with other programs, your own handlers should call the above for any event they don't handle themselves; however, at the present time, this predicate does nothing.

\textbf{Ancilliary Predicates}

The bulk of work in the MIDI interface is performed by \texttt{mrecv/3} and \texttt{msend/3}, together with \texttt{mcreate/4} to create a MIDI device in the first place. We've seen how \texttt{mtime/2} allows programs to query a given MIDI device's timer, or to schedule events, but not how to reset the time in order, for example, to replay a sequence. In fact, there is no way to reset the time other than to call \texttt{mcreate/4} a second time to start up the MIDI device from scratch: this is intentional, as it also gives the MIDI subsystem the opportunity to cancel any outstanding notes, clear the event buffers, and generally tidy up what would otherwise be a messy heap of part-processed events and unfinished business, and in MIDI terms, this can mean hanging notes and other noisy effects!

The status of a MIDI device can be queried with \texttt{mstat/3}:

\begin{verbatim}
?- mstat( midi_in, Event, Sysex ).
  Event = 12 ,
  Sysex = 3
\end{verbatim}

shows that there are 12 pending MIDI events waiting to be picked up by calls to \texttt{mrecv/3}, and that 3 of these are SysEx messages that will be reported as strings, rather than integer triplets. Meanwhile, the call:

\begin{verbatim}
?- mstat( midi_out, Event, Sysex ).
  Event = -1 ,
  Sysex = 0
\end{verbatim}

shows that "midi_out" is an output device (the event count is always -1, just as was used in \texttt{mcreate/4} when creating the device, while in this case, the SysEx count reports back as 0).
While `mstat/3` describes the current status of a MIDI device, `mdata/4` returns its basic settings:

```prolog
?- mdata( midi_in, Event, Sysex, Index ).
Event = 8192 ,
Sysex = 64 ,
Index = 0
```

The returned values give the total number of Event and SysEx strings that can be buffered before calling `mrecv/3`, without losing data; for output devices, Event is reported as -1, and SysEx returns the size, in kilobytes, of the single available output buffer.

A list of existing MIDI devices can be returned by `mdict/2`:

```prolog
?- mdict( 0, Dict ).
Dict = [midi_in,midi_out]
```

When MIDI activity is complete, a MIDI device can be closed by calling `mclose/1`:

```prolog
?- mclose( midi_in ).
yes
```

Should it be desired to call a Windows MIDI API directly (see `winapi/4`), it might be necessary to convert a Prolog MIDI name into its raw, Windows MIDI handle; this can be achieved by calling `midhdl/2`:

```prolog
?- midhdl( midi_out, Handle ).
Handle = 2468091
```
### Appendix R - MIDI Message Definitions

This appendix comprises a series of single tables, based on the MIDI 1.0 Specification Message Summary, as updated in 1995 by the MIDI Manufacturers Association. Please note that, apart from SysEx messages, none of the System Messages are used by WIN-PROLOG.

#### Channel Voice Messages [nnnn = 0-15 (MIDI Channel Number 1-16)]

<table>
<thead>
<tr>
<th>Status D7----D0</th>
<th>Data D7----D0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000nnnn</td>
<td>Okkkkkkkk 0vwwvvv</td>
<td>Note Off event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This message is sent when a note is released (ended).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(kkkkkkk) is the key (note) number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(vvvvvvv) is the velocity.</td>
</tr>
<tr>
<td>1001nnnn</td>
<td>Okkkkkkkk 0vwwvvv</td>
<td>Note On event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This message is sent when a note is depressed (start).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(kkkkkkk) is the key (note) number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(vvvvvvv) is the velocity.</td>
</tr>
<tr>
<td>1010nnnn</td>
<td>Okkkkkkkk 0vwwvvv</td>
<td>Polyphonic Key Pressure (Aftertouch).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This message is most often sent by pressing down on the key after it &quot;bottoms out&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(kkkkkkk) is the key (note) number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(vvvvvvv) is the pressure value.</td>
</tr>
</tbody>
</table>
### Control Change.

This message is sent when a controller value changes. Controllers include devices such as pedals and levers.

Controller numbers 120-127 are reserved as "Channel Mode Messages" (below).

(cccccccc) is the controller number (0-119).

(vvvvvvv) is the controller value (0-127).

### Program Change.

This message sent when the patch number changes.

(ppppppp) is the new program number.

### Channel Pressure (After-touch).

This message is most often sent by pressing down on the key after it "bottoms out". This message is different from polyphonic after-touch. Use this message to send the single greatest pressure value (of all the current depressed keys).

(vvvvvvv) is the pressure value.

### Pitch Wheel Change.

This message is sent to indicate a change in the pitch wheel. The pitch wheel is measured by a fourteen bit value. Center (no pitch change) is 2000H. Sensitivity is a function of the transmitter.

(011111 0mmmmmmmm) are the least significant 7 bits.

(mmmmmmmmm) are the most significant 7 bits.

---

<table>
<thead>
<tr>
<th>Channel Mode Messages (See also Control Change, above)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Status</strong> D7----D0</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>1011nnnn</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
System Common Messages

<table>
<thead>
<tr>
<th>Status D7----D0</th>
<th>Data D7----D0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
<td>0iiiiii 0ddddd 11110111</td>
<td>System Exclusive. This message makes up for all that MIDI doesn't support. (iiiiii) is usually a seven-bit Manufacturer's I.D. code. If the synthesizer recognizes the I.D. code as its own, it will listen to the rest of the message (ddddd). Otherwise, the message will be ignored. System Exclusive is used to send bulk dumps such as patch parameters and other non-spec data. (Note: Real-Time messages ONLY may be interleaved with a System Exclusive.) This message also is used for extensions called Universal Exclusive Messages.</td>
</tr>
<tr>
<td>11110001</td>
<td>Undefined. (Reserved)</td>
<td></td>
</tr>
<tr>
<td>11110010</td>
<td>0lllllll 0mmmmmmm</td>
<td>Song Position Pointer. This is an internal 14 bit register that holds the number of MIDI beats (1 beat= six MIDI clocks) since the start of the song. l is the LSB, m the MSB.</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>11110011</td>
<td>Song Select. The Song Select specifies which sequence or song is to be played.</td>
<td></td>
</tr>
<tr>
<td>11110100</td>
<td>Undefined. (Reserved)</td>
<td></td>
</tr>
<tr>
<td>11110101</td>
<td>Undefined. (Reserved)</td>
<td></td>
</tr>
<tr>
<td>11110110</td>
<td>Tune Request. Upon receiving a Tune Request, all analog synthesizers should tune their oscillators.</td>
<td></td>
</tr>
<tr>
<td>11110111</td>
<td>End of Exclusive. Used to terminate a System Exclusive dump (see above).</td>
<td></td>
</tr>
</tbody>
</table>

### System Real-Time Messages

<table>
<thead>
<tr>
<th>Status D7----D0</th>
<th>Data D7----D0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111000</td>
<td></td>
<td>Timing Clock. Sent 24 times per quarter note when synchronization is required (see text).</td>
</tr>
<tr>
<td>11111001</td>
<td></td>
<td>Undefined. (Reserved)</td>
</tr>
<tr>
<td>11111010</td>
<td></td>
<td>Start. Start the current sequence playing. (This message will be followed with Timing Clocks).</td>
</tr>
<tr>
<td>11111011</td>
<td></td>
<td>Continue. Continue at the point the sequence was Stopped.</td>
</tr>
<tr>
<td>11111100</td>
<td></td>
<td>Stop. Stop the current sequence.</td>
</tr>
<tr>
<td>11111101</td>
<td></td>
<td>Undefined. (Reserved)</td>
</tr>
<tr>
<td>11111110</td>
<td></td>
<td>Active Sensing. Use of this message is optional. When initially sent, the receiver will expect to receive another Active Sensing message each 300ms (max), or it will be assume that the connection has been terminated. At termination, the receiver will turn off all voices and return to normal (non-active sensing) operation.</td>
</tr>
</tbody>
</table>
| 11111111 | Reset.  
Reset all receivers in the system to power-up status. This should be used sparingly, preferably under manual control. In particular, it should not be sent on power-up. |
Appendix S - Standard Syntax

From its inception, the 386-PROLOG kernel has been built around a set of input/output routines that were inspired by the Lisp-like "Standard" syntax that was originally established in LPA's pioneering software product, micro-PROLOG, back in the early 1980s. This earliest of BDS Prolog interpreters contained just one compound data type, the linked list, and used it for all purposes: program clauses, goals, and, of course, dynamic data structures. The lists were written very much in the same style as in Lisp, using round brackets, spaces between elements, and an optional final "cons" constructor denoted by the vertical bar character:

```
(t this is a micro-PROLOG\_list)
```

The simplicity of micro-PROLOG was wonderful, but it had its limitations; in particular, there was no native way in which to represent the different executable structures of the increasingly common "Edinburgh" Prolog variants that were popping up towards the end of the 1980s, when 386-PROLOG was born.

The "New" Standard Syntax

In designing 386-PROLOG, it was decided to include four fundamental compound data types, in order to optimise the new system for the most common data structures found in classical Edinburgh Prolog systems. These data types are the List (used for dynamic data), Tuple (used as single goals), Conjunctions (used as an "and" of goals) and Disjunctions (used as an "or" of goals).

It was also felt that it would be advantageous to be able to write initial Prolog code in a format other than Edinburgh, and which was simpler to parse, in order to bootstrap basic system code into the system, which in turn could support the full Edinburgh syntax, complete with its user-definable operators and other such features. The resulting new "Standard" syntax mapped onto "Edinburgh" in a fairly intuitive manner, as shown here:

```
<table>
<thead>
<tr>
<th>Standard</th>
<th>Edinburgh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>_var</td>
</tr>
<tr>
<td>atom</td>
<td>'Quoted'</td>
</tr>
<tr>
<td></td>
<td>'-&gt;'</td>
</tr>
<tr>
<td>&quot;char list string&quot;</td>
<td>&quot;char list string&quot;</td>
</tr>
<tr>
<td><code>packed format string</code></td>
<td><code>packed format string</code></td>
</tr>
<tr>
<td>12345</td>
<td>123.45</td>
</tr>
<tr>
<td></td>
<td>1.23e45</td>
</tr>
<tr>
<td>[this is a</td>
<td>list]</td>
</tr>
<tr>
<td>(this is a</td>
<td>tuple)</td>
</tr>
<tr>
<td>{this is a</td>
<td>conjunction}</td>
</tr>
<tr>
<td>&lt;this is a</td>
<td>disjunction&gt;</td>
</tr>
<tr>
<td>/* this is a comment */</td>
<td>/* this is a comment */</td>
</tr>
</tbody>
</table>
```
Canonical Operator-Free Syntax

The key features of Standard syntax is that it is canonical and operator-free, with no potential ambiguities such as can be brought on by the redefinition of operators in Edinburgh syntax. Compound data types are delineated by the type of bracket used, and all tuples are shown in prefix form, with the first element of the round-bracketed term being the principal functor.

Several features of the Standard syntax make it much easier to parse than Edinburgh. Firstly, it is deterministic: using Standard syntax, on encountering one of the four opening bracket types, a parser immediately "knows" what kind of data structure is to be built; in Edinburgh, because of the use of operators, decisions as to data type have to be delayed until multiple tokens have been read, in turn needing retrospective adjustments to structures that have previously and provisionally been built. For example, in the term:

```
foo + bar.
```

an Edinburgh parser would first see the atom, "foo", and provisionally build that into its input structure; it would then encounter the atom, "+", and test to see if it was an infix or postfix operator. Finding that it is infix, the parser would search for a third data item, here finding the atom "bar". At this stage, it is possible to build the tuple structure:

```
+(foo,bar)
```

The ensuing ".", followed by white space, shows that the term is complete, and the parser can stop. In Standard syntax, this term would be written:

```
(+ foo bar)
```

The Standard parser would read the opening bracket, and immediately "know" that a tuple was to be created. It would then read "+", and place that as the principal functor; it would go on to read "foo", then "bar", as arguments, before encountering the closing bracket, and the tuple would be complete.

Edinburgh Operator Ambiguities

A second reason Standard syntax is easier to parse than Edinburgh, is because of the latter's facility to define new, or redefine existing, operators. This can be amply illustrated with a simple example; first, let's simply output the term, "+foo", in canonical Edinburgh form:

```
?- eread( X ) <- `+foo`, display( X ), nl. <enter>
+(foo)
X = + foo
```

As can be seen, the term represents a one-tuple, with "+" as the principal functor, and "foo" as its argument. This term results from parsing "+foo" because
"+" is defined as a prefix operator, of type "fx", with a precedence of 500. Now see what happens if we define "foo" as a postfix operator with a higher precedence, as shown here:

```prolog
?- op( 900, xf, foo ).
yes
?- eread( X ) <- `+foo. `, display( X ), nl.
foo(+)
X = + foo
```

The identical character sequence, "+foo", this time represents a one-tuple in which "foo" is the principal functor, and "+" is the argument. Furthermore, if we redefine "foo" to be a postfix operator with to have the same precedence as "+", we get a different result again:

```prolog
?- op( 500, xf, foo ).
yes
?- eread( X ) <- `+foo. `, display( X ), nl.
! ----------------------------------------
! Error 42 : Syntax Error
! Goal : eread(_1)
```

In Standard syntax, the tuple would be explicitly written one way or the other, with no possibility of ambiguity:

```prolog
?- sread( X ) <- `(foo +) `, display( X ), nl.
foo(+)
X = (+) foo
?- sread( X ) <- `(+ foo) `, display( X ), nl.
+(foo)
X = + (foo)
```

In short, apart from being far less complex to parse than Edinburgh syntax, a given Standard syntax stream will always represent the same term, every time, because it is entirely free of user-definable operators.
Interoperability with Other Languages

One key use of Standard syntax is that it can easily be parsed in languages other than Prolog. Without the need to track and process operators, or to allow for non-determism as demanded by a typical Edinburgh parser, programs written in C, C++, Python and other languages can exchange data structures with Prolog with minimal effort.

Interestingly, Standard syntax bears a passing resemblance to JSON, a simple JavaScript-inspired data interchange format: writing a reader or writer for Standard syntax is pretty much as easy as writing one for JSON. However, unlike JSON, which has limited data types and atomic values, Standard syntax allows any Prolog term to be represented in a clear, easily human- or machine readable form.

The Standard Syntax Predicates

Just as Edinburgh syntax is supported by a set of reading and writing predicates, so is Standard syntax. At the "low" level, Edinburgh syntax terms are read by the \texttt{eread/n} predicates, and written quoted or unquoted by the \texttt{eprint/n} and \texttt{ewrite/n} predicates respectively; at the same level, Standard syntax terms are read by the \texttt{sread/n} predicates, and written quoted or unquoted by the \texttt{sprint/n} and \texttt{swrite/n} predicates respectively. For example, here is a term output in Edinburgh syntax by \texttt{eprint/2}, containing a named variable:

```prolog
?- \texttt{eprint( foo(bar) + [a,b|X] = (123,4.56,7e89), [('VAR',X)] ), nl.}
foo(bar) + [a,b|VAR] = (123,4.56,7e89)
X = _
```

Here is the same term, output in Standard syntax by the equivalent call to \texttt{sprint/2}:

```prolog
?- \texttt{sprint( foo(bar) + [a,b|X] = (123,4.56,7e89), [('VAR',X)] ), nl.}
(= (+ (foo bar) [a b|VAR]) {123 4.56 7e89})
X = _
```

The Standard syntax term representation makes the nesting of subterms explicit, and can be compared directly to the canonical Edinburgh form, as shown here by a call to \texttt{eprint/3}:

```prolog
?- \texttt{eprint( foo(bar) + [a,b|X] = (123,4.56,7e89), [('VAR',X)], -1 ), nl.}
= (+ (foo(bar),[a,b|VAR]),(123,4.56,7e89))
X = _
```
Pretty Printing

The sprint/3 predicate can be used to pretty-print terms in Standard syntax, designed to make listings of Prolog programs easy to read when displayed in Standard syntax. Consider the following simple "listing" program:

```prolog
list( Pred, Arity ) :-
    functor( Head, Pred, Arity ),
    forall( (  clause( Head, Body ),
                vars( (Head,Body), Vars ) ),
            (  Body = true
                -> sprint( [Head], Vars, 0 )
                ;  sprint( [Head,Body], Vars, 0 ) )
    ).
```

This picks up each clause for a given dynamic predicate and arity, and then displays it in an automatically formatted Standard syntax style. Suppose we had written the following code:

```prolog
:- dynamic( my_append/3 ).

:- dynamic( my_reverse/2 ).

my_append( [], L, L ).

my_append( [H|T], L, [H|A] ) :-
    append( T, L, A ).

my_reverse( [], [] ).

my_reverse( [H|T], R ) :-
    my_reverse( T, L ),
    append( L, [H], R ).
```

We can now list this program in Standard syntax with the following call:
Terminating Standard Syntax Terms

Just as Edinburgh syntax terms must be terminated with a special token ("." followed by either a space or control character), so Standard syntax terms must be terminated by either a space or control character, as shown in the examples above. Note that this terminating space is not required when trying to read back terms that have been pretty printed, as the terms are always finished with a new line, which contains the necessary control character.

The Standard Syntax Source Format

Listings such as that shown above can be directly loading into 386-PROLOG using the `sload/1` predicate (not currently documented in this manual); this was implemented to enable the initial bootstrapping of the Prolog environment, where key code is written in Standard syntax, providing the support for subsequent Edinburgh syntax, including the definitions of the system operators and other such prerequisites. Clauses with no body ("facts") are represented simply as a head tuple (or atom) as the only term in a single-element list, for example:
while those with a body ("rules") are represented with the head and body as a pair of terms in a two element list:

```
[(pred arg1 arg2) body_term]
```

where the "body_term" can any callable term, such as an atom, a tuple, or a conjunction or disjunction of callable terms. The second clause for the example above, "my_reverse/2", shows a conjunction of two tuples.

This simple file format can also be used to hand-write an overlay for **386-PROLOG** - provided you are happy to work with the set of true built-in predicates and nothing more! Details of how to create such overlays are beyond the scope of this appendix, but may be documented at a future date.
Appendix T - TreeView Controls

One major addition to version 6.100 **WIN-PROLOG** was the introduction of "TreeView" controls. These were omitted from the first raft of "Common Controls", added in version 4.500 way back in July, 2004, partly because their programming at the user level was considered overly complex; indeed their addition at long last has necessitated an entire dedicated appendix in this reference manual.

About TreeView Controls

For many years, **WIN-PROLOG** has supported a clutch of special, extended "common controls", namely window types with pre-defined appearance and behaviour. Building on the internally defined standard controls, such as "ListBox", "Edit" and "ScrollBar", the common controls provide features such as "TabBox", "TrackBar", "ToolTip" and many more. One of this family, and an important one at that, "TreeView", has not been included in **WIN-PROLOG** until version 6.100: this has simply been down to its considerable complexity in setting up and subsequent querying.

So what is a TreeView control? In its simplest form, it's a semi-graphical tree representation of a two-dimensional data structure: a bit like a list box, in which you can add further sub-list-boxes to the individual entries, and which the user can collapse and expand as he or she sees fit. For example, a TreeView control could contain a list of all folders and files on a hard disk, with the folders at anyone level forming a vertical list, and the contents of each such folder, being contained in a collapsible subtree on the right. Any folders in a subtree, can have further sub-trees attached, in an effectively unlimited recursive fashion.

Rather than pass one, single large structure into a complex "show as treeview" predicate, **WIN-PROLOG** uses a collection of simple predicates, analagous to those used when programming 'ListBox' or 'ComboBox' controls, with individual predicates to add, delete, select and check for selection the nodes of the tree.

Keeping Track of Location

Because ListBox and ComboBox controls are simple and one-dimensional, there is no need for the user to keep track of "where they are" in the box when adding information. For example, when calling `wlistadd/4` to add an item to a list box, just item name as a string, its optional position and its optional value, need to be specified. In a TreeView control, the item being added needs to have its location in the tree specified: it could be at the top level of the tree, or deep down within, or to the "right" of (one level deeper) or "beneath" (next entry at current depth) some other node.

The way this is handled in **WIN-PROLOG** is that the individual predicates both accept, and return "Node ID" handles. For example, `wtvwadd/5` is very similar to `wlistadd/4`, adding a string and an optional data item to the tree, but rather than specifying an (optional) position as the second argument, the Node Id of an existing location in the TreeView control is specified. The additional, 5th argument (compared to the list box equivalent predicate) returns a 32-bit integer, which is the Node Id of the newly added tree node, and which is in turn passed to further calls to `wtvwadd/5` as the tree is populated.
Planting a Tree

When adding items to a list box or combo box, you can optional specify a linear, numerical position for the item, or leave it up to Windows to place the item, according to the control's window style. For example, given a list box called, say, "(foo,10)", the following call would add the string, "hello", and the associated value, "123", into the fifth place in the list:

?- wlstadd( (foo,10), 5, `hello`, 123 ).
yes

By specifying "0" as the second argument, rather than an explicit position number, wlstadd/4 will either place the new value at the bottom of the list, or insert it at the appropriate sorted location, depending on whether the control included the "lbs_sort" style.

With TreeView controls, there is no option to place an item at an explicit point in the tree: all new entries go to the bottom of the existing part of the tree, as identified by the "Node ID" (hereafter, simply ID). By definition, the root node of a tree has an ID of zero ("0"); given a tree view control called, say, "(bar,20)", the following call would add the string, "there", and the associated value, "456", at the end of the root node of the tree:

?- wtvwadd( (bar,20), 0, `there`, 456, ID ).
ID = 12345678

The extra, variable argument, "ID", returns the ID of the tree node that has just been added, and this is the key to how to grow a tree depthwise, as well as linearly. Please note that the returned value could be more or less any number; "12345678" is simply shown here for the purposes of illustration.

If we want to add more entries to the tree root node, we simply call wtvwadd/5 again, specifying zero ("0") as the second argument. Each successfully added entry will return a different ID value. If we only ever specify zero ("0") as the second argument, then effectively, we will have created a single, collapsible, unsorted list box. The fun begins when we make use of those returned ID values.

Branching Out

Suppose we want to create a tree that looks something like this:

|-- one
  |-- two
  |-- three
|-- four
|-- five
The root node of the tree will contain three entries, "one", "four" and "five", while the first of these entries will itself contain a further two entries, "two" and "three". Imagine the above tree is in a control called "(foo,10)"; here is the sequence of calls we would need to make to build the tree:

```prolog
?- wtvwadd( (foo,10), 0, `one`, 1, ID ).
ID = 12345678
?- wtvwadd( (foo,10), 0, `four`, 4444, ID ).
ID = [random]
?- wtvwadd( (foo,10), 0, `five`, 55555, ID ).
ID = [random]
?- wtvwadd( (foo,10), 12345678, `two`, 22, ID ).
ID = [random]
?- wtvwadd( (foo,10), 12345678, `three`, 333, ID ).
ID = [random]
```

The first three calls in this sequence are appending strings to the root node, whose ID is always zero ("0"); the last two calls are appending strings to the first node, whose ID is whatever value was returned from the first call. In this example, the root node was fully populated before the first entrie was given its subtree, effectively using a breadth-first approach. There is nothing to stop you building trees in a depth-first fashion; all you have to do is keep track of the IDs that are returned. The following sequence will create exactly the same tree as before, but in a depth-first way:

```prolog
?- wtvwadd( (foo,10), 0, `one`, 1, ID ).
ID = 12345678
?- wtvwadd( (foo,10), 12345678, `two`, 22, ID ).
ID = [random]
?- wtvwadd( (foo,10), 12345678, `three`, 333, ID ).
ID = [random]
?- wtvwadd( (foo,10), 0, `four`, 4444, ID ).
ID = [random]
?- wtvwadd( (foo,10), 0, `five`, 55555, ID ).
ID = [random]
```

Simply by specifying any returned ID as the second argument to `wtvwadd/5`, you can append strings and numerical values to any node already in the tree. In the above examples, some arbitrary numerical values (1, 22, etc) have been specified: these values have no significance to `WIN-PROLOG`, and are available to be used in any way the user sees fit.
Climbing the Tree

When building a tree, it is usually best to keep track of where you are at any one time, holding onto the IDs of any nodes you want to populate with their own subtrees. But suppose you have an existing tree, and want to add more to it, or simply navigate around it? Two key predicates are used to retrieve information from a TreeView control: \texttt{wtvwlnk/4} and \texttt{wtvwget/4}. The first of these allows you to link from any given node in a tree, to another, using a numerical flag to specify the direction in which you wish to move.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tvgn_root</td>
<td>16'0000</td>
<td>Retrieves the topmost item in a TreeView control</td>
</tr>
<tr>
<td>tvgn_next</td>
<td>16'0001</td>
<td>Retrieves the next sibling item</td>
</tr>
<tr>
<td>tvgn_previous</td>
<td>16'0002</td>
<td>Retrieves the previous sibling item</td>
</tr>
<tr>
<td>tvgn_parent</td>
<td>16'0003</td>
<td>Retrieves the parent item</td>
</tr>
<tr>
<td>tvgn_child</td>
<td>16'0004</td>
<td>Retrieves the first child item</td>
</tr>
<tr>
<td>tvgn_firstvisible</td>
<td>16'0005</td>
<td>Retrieves the first visible item</td>
</tr>
<tr>
<td>tvgn_nextvisible</td>
<td>16'0006</td>
<td>Retrieves the next visible item</td>
</tr>
<tr>
<td>tvgn_previousvisible</td>
<td>16'0007</td>
<td>Retrieves the previous visible item</td>
</tr>
<tr>
<td>tvgn_drophilite</td>
<td>16'0008</td>
<td>Retrieves the target item of a drag and drop</td>
</tr>
<tr>
<td>tvgn_caret</td>
<td>16'0009</td>
<td>Retrieves the currently selected item</td>
</tr>
<tr>
<td>tvgn_lastvisible</td>
<td>16'000a</td>
<td>Retrieves the last visible item</td>
</tr>
</tbody>
</table>

Suppose we have built the tree shown in the previous example, but has lost track of the returned IDs; the following call will retrieve the handle of the topmost item (tvgn_root=“0”) in the root node (ID=“0”) of the TreeView control:

```prolog
?- wtvwlnk( (foo,10), 0, 0, ID ).
ID = 12345678
```

Now we can make a call to retrieve the node with this first ID:

```prolog
?- wtvwget( (foo,10), 12345678, String, Value ).
String = `one`
Value = 1
```

Having found the ID of the topmost item in the tree, we can work breadthwise to its siblings, or depthwise to its children, as desired; first, let’s link to the
next sibling (tgvn_next="1") of the node at our retrieved ID:

?- wtvwlnk( (foo,10), 12345678, 1, ID ), wtvwget( (foo,10), ID, String, Value ).
ID = [random]
String = `four`
Value = 4444

The following similar call gets the first child (tgvn_child="4") of the node at our retrieved ID:

?- wtvwlnk( (foo,10), 12345678, 4, ID ), wtvwget( (foo,10), ID, String, Value ).
ID = [random]
String = `two`
Value = 22

Simply by choosing the appropriate constants, it is easy to traverse any tree within a TreeView control, using natural recursive techniques.

**Pruning and Decorating the Tree**

You can delete any given node, including all its subtrees, with the `wtvwdel/2` predicate; for example, assuming the ID value of "12345678" used to illustrate the above examples, the following call would delete the "one" node and its subtree of "two, three":

?- wtvwdel( (foo,10), 12345678 ).
yes

The subtree will disappear from the TreeView control, and the remaining siblings of the deleted node, in this case "four" and "five", will shuffle up to close the gap that would otherwise be left.

Providing that the "tvs_hasbuttons" style was selected when the TreeView control was created, each node in the tree can have an associated button, which can be retrieved by the `wtvwbbtn/5` predicate; the following call will return the state of the button associated with the specified node:

?- wtvwbbtn( (foo,10), 12345678, Button ).
Button = 2

Return values include 0 (button not present), 1 (button present, but not ticked) and 2 (button ticked). If you specify the button value as one of these three integers at the time of the call, you can set the button's visibility and state.
The final predicate in the "wtvw???/n" TreeView collection lets you to retrieve or set the currently selected node in a tree. For example, suppose you have manually highlighted "four" in the above tree; you can find this out and retrieve the data with the following call:

```prolog
?- wtvwsel( (foo,10), ID ), wtvwget( (foo,10), ID, String, Value ).
ID = [random]
String = `four`
Value = 4444
```

The same predicate can be used to move the selection programatically: simply supply a valid node ID as the second argument.

### Messages in the Trees

Throughout the life of a TreeView control, messages are sent to WIN-PROLOG to notify any changes to its, for example as the user navigates the tree, expands or collapses subtrees, or even (if the tvs_editlabels style has been included) as the user starts to edit, or completes the editing of a label on the tree. None of the available messages are used directly by WIN-PROLOG, but they can be intercepted just like any other message by the user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg_tvedit</td>
<td>31</td>
<td>user has started to edit the given node label</td>
</tr>
<tr>
<td>msg_tvrename</td>
<td>32</td>
<td>user has finished editing the given node label</td>
</tr>
<tr>
<td>msg_tvcancel</td>
<td>33</td>
<td>user has cancelled editing the given node label</td>
</tr>
<tr>
<td>msg_tvshow</td>
<td>34</td>
<td>user has expanded the subtree of the given node</td>
</tr>
<tr>
<td>msg_tvhide</td>
<td>35</td>
<td>user has collapsed the subtree of the given node</td>
</tr>
<tr>
<td>msg_tvleave</td>
<td>36</td>
<td>user has navigated away from the given node</td>
</tr>
<tr>
<td>msg_tventer</td>
<td>37</td>
<td>user has navigated into the given node</td>
</tr>
<tr>
<td>msg_tvdrag</td>
<td>38</td>
<td>user has commenced a drag from the given node</td>
</tr>
</tbody>
</table>

For reasons unknown, there do not appear to be any standard message notifications when the user changes the state of a node's button: this appears to be to be a curious omission on Microsoft's part.

### Decorating the Tree

The basic black text-and-line on white background appearance of TreeView controls can be change radically by the user with the help of the sndmsg/5 predicate. A large collection of TreeView control messages let you change line, text and background colours, the font used to display items, as well as the ability to attach an "image list" to the control, containing a set of icons to be displayed as the state of the tree changes.
The full set of control messages is as documented on the Microsoft Developer Network (MSDN), and can easily be accessed by search engines such as Google; a comprehensive enumeration and explanation is beyond the remit of the present document. However, a few examples will help give you the general idea. Let's start by changing the foreground (text/line) and background colours; assuming we have a TreeView control called ");(foo,10)", the following will change the background, text and line colours to dark blue, yellow and green respectively:

```
?- sndmsg( (foo,10), tvm_setbkcolor, 0, 16'400000, _ ),
       sndmsg( (foo,10), tvm_settextcolor, 0, 16'00ffff, _ ),
       sndmsg( (foo,10), tvm_setlinecolor, 0, 16'00ff00, _ ).
```

Notice that all the relevant messages begin with the prefix, "tvm _". The second (lParam) data parameter contains a simple RGB value, although (thanks to Intel's "little endian" storage convention, it is written back-to-front, in hexadecimal as "16BBGRR", where "BB" is the blue level (00-FF), "GG" is the green level (00-FF) and "RR" is the red level (00-FF). Hence 16'400000 is 64 parts blue out of 256 (25%), and zero green and zero red, making a dark, navy blue. Likewise, 16'00ffff is zero blue, 256 parts green out of 256 (100%), and 256 parts red out of 256 (100%), making a bright yellow.

To change the display font in a TreeView control, you can use the regular WIN-PROLOG font predicates; for example, the following call will our example TreeView to use "Courier New":

```
?- wfcreate( font, 'Courier New', 36, 2 ),
       wfont( (foo,10), font ).
```

A number of small bitmap images (.BMP) can be combined into something called an "image list", which effectively is an indexed array of same-size bitmaps, any one of which can be displayed at a given time. These image lists can then be attached to TreeView controls, which in turn will automatically display one or other of the images at each node, depending upon the nodes state. Check boxes within the TreeView can display up to three consecutive images in place of the stand "checked" and "unchecked" conditions, and a pair of images can also be used to indicate "selected" or "non-selected" state.

Let's imagine that we have created two bitmap files, each containing four 48*48 pixel icons. The four images in each list are placed side by side, making an overall bitmap of 192*48 pixels size, perhaps like the examples shown in Figure 1, and which can be found in your WIN-PROLOG examples folder.

![Image 1](example1.png)

Figure 1: Two sets of four 48x48 pixel icons
To load these bitmaps as image lists, we need to make a `winapi/4` call to the common control function, "ImageList_LoadImageW", whose arguments are fully documented on MSDN, but which for our purposes, can be fixed apart from the file name string (here, "tvdemo_0.bmp") and the width of each component icon (here, 48):

```prolog
?- winapi( (comctl32,'ImageList_LoadImageW'),
            [0,`tvdemo_0.bmp`,48,1,-1,0,16'0010],
            2,
            Handle
         ),
        assert( known_img(demo_0,Handle) ).
Handle = [random]
```

The returned handle number will be different every time, and is to all intents and purposes, a random number. However you will need this value in order to utilise your image list, and so it is a good idea to assert it together with a name you would like to use to recall it; in the example able, we've used "known_img/2" to store the handle, and in this instance, we've used the name, "demo_0" to reference it.

Suppose we did likewise with another file, "tvdemo_1.bmp", storing its handle with the name, "demo_1"; we can later pick up these handles and assign them to the TreeView control, with calls like this:

```prolog
?- known_img( demo_0, Img0 ),
   known_img( demo_1, Img1 ),
   sndmsg( (foo,10), tvm_setimagelist, 0, Img0 , _ ),
   sndmsg( (foo,10), tvm_setimagelist, 2, Img1 , _ ).
Img0 = [random]
Img1 = [random]
```

The "tvm_setimagelist" message takes a numerical constant as its first (wParam) parameter, and the handle of a loaded image list as its second (lParam) argument. The constant can have of only two values, 0 ("tvsil_normal") or 2 ("tvsil_state"). The former value indicates that the image list contains the selected, non-selected and overlay images for items in a TreeView control, while the latter indicates that the image list contains images to reflect user-defined states within the TreeView control.

**TVDEMO.PL and TVUTIL.PL**

Demonstrations of how to handle TreeView controls are included in a pair of Prolog source files, TVDEMO.PL and TVUTIL.PL, which you will find in the WIN-PROLOG examples folder. The simplest initial demo, is to load both the above files, and run the query:
This will create a simple dialog, with just one control, a TreeView called "(foo,10)", and populate it with strings, as shown in Figure 2. The strings/0 demo calls a useful utility predicate, tv_add_text/4, defined in TVUTIL.PL, which stores a list of nested tuples of Prolog strings in a TreeView control:

```
strings :-
  make_tree,
  tv_add_text((tree,81), 0, 0, ['hello'('there'('goes', 'another'), 'world'('news', 'story'))],
  wshow(tree, 1).
```

The four arguments of tv_add_text/4, (1) specify the TreeView control itself, (2) the node ID within the TreeView at which to add the given items, (3) an integer from which to start numbering successive entries, and (4) the list of nested tuples. In the above example, the list of nested tuples is placed in the root of the tree (0), and numbering starts at zero (0).

**Example Message Handler**

The demo program includes a simple message handler, which displays notifications in the WIN-PROLOG console while you navigate, expand and contract the tree. So for example, if you were to click around in the treeview control, you see output such as the following appearing in the console:
The handler code itself displays the TreeView control's handle, a chopped-off version of the message name (omitting "msg_tv" and converting what's left to upper case), then the node ID, the string itself, and finally, the data value which was assigned by the tv_add_text/4 predicate:

```prolog
tree_handler( Window, Message, Handle, _ ) :-
    cat( List, Message, [6] ),
    List = [msg_tv,Action],
    lwrupr( Action, Name ),
    wtvwget( Window, Handle, Text, Item ),
    sprint( Window ) ~> Wind,
    output( Out ),
    output( 0 ),
    fwrite( s, -10, -1, Wind ),
    fwrite( ` ` ),
    fwrite( a, -8, 0, Name ),
    fwrite( n, 10, 0, Handle ),
    fwrite( ` ` ),
    fwrite( s, -15, -1, Text ),
    fwrite( ` ` ),
    fwrite( n, 10, 0, Item ),
    fwrite( `~M~J` ),
    output( Out ).
```

Gaudy Embodied

Having called strings/0 to created a sample TreeView control, the following call will give you an indication of just how much you can adjust its appearance, by changing fonts and colours, and adding image lists. First make sure that TVDEMO_0.BMP and TVDEMO_1.BMP are in the current folder, then type:
?- decorate.  
    yes

This will change the font and colours as per the example calls on previous pages, and assign the two image lists to the TreeView, as shown in Figure 3.

**Seeing the Wood for the Trees**

Apart from being able to check which item in a tree is selected (wtvwsel/2) and then retrieve its data (wtwwget/4), thanks to wtwwlnk/4, it is relatively easy to navigate an entire tree, and return its contents as a nested Prolog structure. The TVUTIL.PL file includes a utility predicate to do just that; try the call:

?- tv_get_data( (tree,81), 0, Tree ),
    draw_tree( Tree ).

0 hello
1     there
2         goes
3         another
4     world
5     news
6     story

Tree = [(`hello`,0)((`there`,1)((`goes`,2),(`another`,3)),(`world`,4)((`news`,5),(`story`,6)))]

The three arguments for tv_get_data/3, (1) specify the TreeView control itself, (2) the node ID within the TreeView from which to retrieve the tree, and (3) a variable to return the tree. The draw_tree/1 predicate is defined in TVDEMO, is simply a pretty-printer to show trees in an intended format.

If you want to create trees with your own numeric values, rather than the automatic sequential numbering performed by tv_add_text/4, you can use tv_add_data/4 instead. which is also defined in TVUTIL.PL.

**Further Reading**

There is a lot more information about TreeView controls, publically available on the Microsoft Developer Network (MSDN). It is recommended that you read through their introduction, which will add further detail to the information presented in this appendix. Please also look through the TVDEMO.PL and TVUTIL.PL Prolog source files, which will help give you further ideas about what you can do with WIN-PROLOG's TreeView controls.
Appendix U - Full 64-Bit Implementation and Considerations

By far the most significant feature of WIN-PROLOG 7.000, was the creation of a genuine, full-bore 64-bit implementation of the system. As before, the 32-bit version runs on all versions of Windows from 98SE through 10; the new 64-bit version will run on all versions of Windows X64 from 7 through 10.

Why 64-bit?

In a nutshell, memory! The X86 (32-bit) architecture of 386-PROLOG gave users a theoretical 4Gb address space into which to cram all heaps, stacks and program code; in Windows this was halved to 2Gb, because Windows reserves half of the memory address space for itself.

The X64 (64-bit) architecture smashes through this limitation, allowing applications to access as much memory as can be installed into any given computer. 128Gb of program clauses? No problem any more!

Unfortunately, everything comes with a price, and the 64-bit memory pointers mean that the average Prolog memory structure is about 80% larger than before; where an 8Mb heap once sufficed for a given application in X86-PROLOG, it might be better to allocate 16Mb for X64.

Increasing the amount of data being processed from 32-bit chunks to 64-bit also impacts on speed, not least, because the number of cache hits will be reduced. In testing, X64-PROLOG has been found to be about 15% slower than X86-PROLOG (formerly, 386-PROLOG).

Distinguishing the 32-bit and 64-bit Versions

The two versions of WIN-PROLOG have slightly modified welcome banners, which include the label, "X86" for the 32-bit version, and "X64" for the 64-bit version, for example:

```
BDS WIN-PROLOG 7.000 X64 S/N 0020596320 28 Feb 2019
Copyright 1989-2019 Brian D Steel (www.solanum.org)
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=256 T=2048 P=8192 S=64 I=256 O=256
```

Compatibility of X86 vs X64 Prologs

The two versions of the engine maintain maximum mutual compatibility; in practice this means that about 99% of code "just works". This source code compatibility is extended to the object file (.PC) format: you can optimise files in X86, then load and run them in X64, or vice versa.
The only places where a programmer might need to know which version of Prolog is in use, is when handling large integers, such as in use of the rpn/2 predicate, when processing formatted integers with the fread/4 and fwrite/4 predicates, and when handling values in association with winapi/4 (see below). Simply, **X86-PROLOG** uses 32-bit integers, in the range -(2^31)..<2^31>-1 (-2147483648..2147483647), while **X64-PROLOG** uses 64-bit integers, in the range -(2^63)..<2^63>-1 (-2^63)..<2^63>-1 (-9223372036854775808..9223372036854775807). So:

```prolog
?- fread( r, 8, 16, X ) <~ `ffffffff`. <enter>
```

will, in **X86-PROLOG**, return:

```
X = -1
```

but will, in **X64-PROLOG**, return:

```
X = 4294967295
```

Issues can occur with hexadecimal formatted output, where (for example) negative or large positive integers require 16 digits of output in X64, rather than just 8 as in X86:

```prolog
?- fwrite( r, 0, 16, -1 ), nl. <enter>
```

will, in **X86-PROLOG**, display:

```
FFFFFFFF
```

but will, in **X64-PROLOG**, display:

```
FFFFFFFFFFFFFFFF
```

Another place where the larger native integer size in **X64-PROLOG** might affect programs compared to **X86-PROLOG**, is in use of the rpn/2 integer arithmetic predicate, so that:

```prolog
?- rpn( [16'80000000,1,1], X ), fwrite( r, 0, 16, X ), nl. <enter>
```

will, in **X86-PROLOG**, return and display:

```
1
```
\[ X = 1 \]

but will, in \texttt{X64-PROLOG}, return and display:

\[
\begin{align*}
100000000 \\
X &= 4294967296
\end{align*}
\]

In pretty much all code except that handling large integers, both the X86 and X64 versions of \texttt{WIN-PROLOG} will behave identically.

\section*{Programming X86 vs X64}

Just so that programs have the option of treating the 32-bit (X86) and 64-bit (X64) versions separately, for example when doing integer operations as highlighted in the previous section, two more cases have been added to the \texttt{ver/1} predicate:

\[
?- \text{ver}(32). \quad \text{<enter>}
\]

returns:

\[
\begin{align*}
\text{yes} & \quad \% \text{ in } \texttt{X86-PROLOG} \\
\text{no} & \quad \% \text{ in } \texttt{X64-PROLOG}
\end{align*}
\]

while:

\[
?- \text{ver}(64). \quad \text{<enter>}
\]

returns:

\[
\begin{align*}
\text{no} & \quad \% \text{ in } \texttt{X86-PROLOG} \\
\text{yes} & \quad \% \text{ in } \texttt{X64-PROLOG}
\end{align*}
\]

Both calls will fail in earlier versions of \texttt{386-PROLOG}, so this allows special cases to be created if necessary, with code constructs such as:

\[
\text{foo}(\ldots) :- \\
\begin{cases}
\text{ver}(32) \\
\rightarrow \text{<do 32-bit specific X86 code>}
\end{cases} ; \text{ver}(64)
\]
Again, it is stressed that you will need the above only in rare cases when dealing with large integers, or possibly in advanced uses of the winapi/4 predicate (see below).

**Windows API Predicate**

The X64.winapi/4 predicate retains the syntax and semantics of the X86 version, such that nearly all calls out to Windows APIs will work directly, with no further tweaking. There are, however, two cases to consider. Firstly, the size of the return value, as described in the next two sections.

**API Return Values**

Most integer X64 Windows API functions actually return 32-bit signed values, rather than 64-bit ones, so that, for example, "-1" is returned as "0xFFFFFFFF" (4294967295), rather than "0xFFFFFFFFFFFFFFFF" (-1).

Two new functions have been added to the rpn/2 predicate to help handle transitions between 32-bit and 64-bit values:

$ sign extension, which copies bit 31 into bits 63..32

& word truncation, which zeroes bits 63..32

As examples, see these calls in X64-PROLOG:

```prolog
?- rpn([4294967295,$], X). <enter>
X = -1

?- rpn([-1,$], X). <enter>
X = -1

?- rpn([-1,&], X). <enter>
X = 4294967295

?- rpn([4294967295,&], X). <enter>
X = 4294967295
```
So you would use "$" if (say) you wanted to compare the return value from a winapi/4 call against the value "-1"; previously, code might look like this:

```
winapi( (module,some_32_bit_function), [], 0, Result ),
( Result = -1
  -> fail
  ; true
 ),
```

but this will not work correctly in X64. Instead, replace ":=//2" with an call to \rpn/2:

```
winapi( (module,some_32_bit_function), [], 0, Result ),
( rpn( [Result,$], -1 )
  -> fail
  ; true
 ),
```

which will work both in X64 and X86.

In **X86-PROLOG**, both functions are present, but essentially do nothing.

**API Structures**

Passing simple values to \texttt{winapi/4}, such as integers, text strings, memory file buffer names, and so on, works identically on the X86 and X64 versions of Prolog, but care must be taken when calling APIs that expect, or return, structures. Simply, X86 structures tend to be packed with 32-bit values and pointers, while the equivalent X64 structures will contain a mixture of 32-bit or 64-bit values, and 64-bit pointers. To complicate matters further, X64 structures always align 64-bit values on 8-byte boundaries. So imagine you have a "C" structure containing the following fields:

```
{   INT size ;
    HWND handle ;
    INT mode ;
}
```

previous code to load this into a memory buffer might have looked like:

```
( putx( 4, Size ),
```
putx( 4, Handle ),
putx( 4, Mode )
) ~> Struct,

This would work for X86, if "Struct" was then passed into winapi/4, but not in X64. Thanks to the new extension to ver/1 (see above), we can now write something like this:

(  ver( 32 )
-> (  putx( 4, Size ),
    putx( 4, Handle ),
    putx( 4, Mode )
  ) ~> Struct
;  ver( 64 )
-> (  putx( 4, Size ),
    putx( 4, 0 ), % padding to align handle
    putx( 8, Handle ),
    putx( 4, Mode )
  ) ~> Struct
),

Alternatively, to cut down on code duplication, this would also work on both platforms:

(  ver( 32 )
-> Q = 4
;  ver( 64 )
-> Q = 8
),
(  putx( Q, Size ), % includes alignment padding
  putx( Q, Handle ),
  putx( 4, Mode )
) ~> Struct

The same tricks work with getx/2, when reading structures after calling a Windows API function.
Installation of 32-bit and 64-bit Versions

The X86 (32-bit) and X64 (64-bit) versions of WIN-PROLOG can happily co-exist. They are so highly compatible, that pretty much anything you write in one version, will "just work" in the other. This compatibility extends to object files: as outlined above, pretty much the only times you might need to think about which version of the system you are running, is when handling large integer values.

All files in the WIN-PROLOG installation (.PL, .PC, .OVL, etc.) are the same, whether you are running the X86 or X64 version of of the system. Only the executable file itself (.EXE), and any of its Dynamic Link Libraries (DLLs) is distinct. Your system ships with both executables, named, respectively, PROX86W.EXE and PROX64W.EXE. Matching DLLs are similarly named, for example PROX86X.DLL and PROX64X.DLL.

For compatibility with earlier versions of WIN-PROLOG, the "main" executable shipped and installed, and called PRO386W.EXE, is in fact nothing other than a pure copy of PROX86W.EXE: to run the 32-bit, X86 system, just execute this program as before. If you want to switch to using the 64-bit, X64 version, just copy PROX64W.EXE over PRO386W.EXE. To revert to the 32-bit, X86 version, copy PROX86W.EXE over PRO386W.EXE.

Alternatively, you can run both the X86 and X64 versions side by side for testing and comparison, without renaming any files, using these Windows commands at the "c>" prompt or in the Windows "Run" dialog:

```
c> prox86w @pro386w          % runs x86 (32-bit) prolog  <enter>
c> prox64w @pro386w          % runs x64 (64-bit) prolog  <enter>
```
Index

Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BMP</td>
<td>126</td>
</tr>
<tr>
<td>32-bit</td>
<td>29, 553, 699, 1061</td>
</tr>
<tr>
<td>64-bit</td>
<td>29, 553, 699, 1061</td>
</tr>
<tr>
<td>64-Bit Implementation</td>
<td>1061</td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute file name</td>
<td>93, 95</td>
</tr>
<tr>
<td>address</td>
<td>34</td>
</tr>
<tr>
<td>adjustment</td>
<td>847</td>
</tr>
<tr>
<td>anonymous variable</td>
<td>214, 220, 229, 609, 614, 646</td>
</tr>
<tr>
<td>ANSI</td>
<td>982, 986</td>
</tr>
<tr>
<td>ANSI fixed font</td>
<td>745, 748</td>
</tr>
<tr>
<td>ANSI var font</td>
<td>745, 748</td>
</tr>
<tr>
<td>API</td>
<td>45, 754</td>
</tr>
<tr>
<td>argument domain</td>
<td>34</td>
</tr>
<tr>
<td>argument mode</td>
<td>33</td>
</tr>
<tr>
<td>argument range</td>
<td>34</td>
</tr>
<tr>
<td>argument type</td>
<td>34</td>
</tr>
<tr>
<td>arithmetic</td>
<td>271, 381, 552</td>
</tr>
<tr>
<td>arithmetic functions</td>
<td>271, 381, 552</td>
</tr>
<tr>
<td>arity</td>
<td>34</td>
</tr>
<tr>
<td>ASCII</td>
<td>982, 986</td>
</tr>
<tr>
<td>assignment</td>
<td></td>
</tr>
<tr>
<td>destructive</td>
<td>104</td>
</tr>
<tr>
<td>atom</td>
<td>34</td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>backtracking</td>
<td>235</td>
</tr>
<tr>
<td>Berkeley</td>
<td>1011</td>
</tr>
<tr>
<td>big endian</td>
<td>983</td>
</tr>
<tr>
<td>Term</td>
<td>Page(s)</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>binary data</td>
<td>763</td>
</tr>
<tr>
<td>binary functions</td>
<td>271, 552</td>
</tr>
<tr>
<td>Bitmap</td>
<td>126</td>
</tr>
<tr>
<td>BOM</td>
<td>476, 983</td>
</tr>
<tr>
<td>break interrupt</td>
<td>132</td>
</tr>
<tr>
<td>browse box</td>
<td>134</td>
</tr>
<tr>
<td>busy cursor</td>
<td>135</td>
</tr>
<tr>
<td>Byte Order Mark</td>
<td>476, 983</td>
</tr>
<tr>
<td>Byte Swapped Byte Order Mark</td>
<td>983</td>
</tr>
<tr>
<td>cache directory</td>
<td>1007</td>
</tr>
<tr>
<td>canonical form</td>
<td>216</td>
</tr>
<tr>
<td>Carlos</td>
<td>1024</td>
</tr>
<tr>
<td>catch-and-throw</td>
<td>147, 221, 222, 223</td>
</tr>
<tr>
<td>char</td>
<td>34</td>
</tr>
<tr>
<td>char_list</td>
<td>34</td>
</tr>
<tr>
<td>character encoding</td>
<td>238</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-16LE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-32BE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-32LE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-8</td>
<td>983</td>
</tr>
<tr>
<td>UTF-BS</td>
<td>989</td>
</tr>
<tr>
<td>character set</td>
<td></td>
</tr>
<tr>
<td>ANSI</td>
<td>986</td>
</tr>
<tr>
<td>ASCII</td>
<td>986</td>
</tr>
<tr>
<td>ISO</td>
<td>986</td>
</tr>
<tr>
<td>ISO/IEC 8859-1</td>
<td>986</td>
</tr>
<tr>
<td>OEM</td>
<td>986</td>
</tr>
<tr>
<td>RAW</td>
<td>986</td>
</tr>
<tr>
<td>Unicode</td>
<td>986</td>
</tr>
<tr>
<td>clause</td>
<td>34, 107</td>
</tr>
<tr>
<td>adding</td>
<td>107</td>
</tr>
<tr>
<td>client</td>
<td>1016</td>
</tr>
</tbody>
</table>
Clocksin and Mellish 29
colour box 543
Command History 371
common dialog 48
compiler
    hashing 42
    incremental 42
    optimising 42
compound_term 34
computer font 31
conjunction 51
Console History 1007
console window 41, 115, 135, 584
CRC32 172, 413, 578
cut
    local 55
Cyclic Redundancy Check 172, 413, 578
cyphertext 184, 208
C interface 754

data compression 260, 640, 978
data encryption 184, 208, 980
date_time 34, 974
DDE 755
Delphi 754
destructive assignment 104
development environment 937
device default font 746, 748
dialog
    browse box 134
colour box 543
common 48
font box 264
message box 420
open box 477
print box 504
progress box 498, 501
save box 556
directory cache 1007
HELP 364, 365
dirty region 341
disjunction 57
DLL calling 754
domain name 1016
DOS-PROLOG 1026
DOS Box 197
double negation 269, 693
dynamic 201
Dynamic Link Libraries 29

Edinburgh syntax 219, 613
either:or 682
empty_list 34
end_of_file 219, 613
endian big 983
little 983
environment variables 211
equals 63
error handling
catch-and-throw 147, 221, 222, 223
hook 221, 222, 223
evaluation arithmetic 271, 381, 552
events MIDI 1028
existential quantification 84, 255, 575, 676
exit port 140, 373
fail 140
expr 34
expression
  arithmetic 271, 381, 552
Extensible Markup Language 563

F
FB01 1026
file_name 34
file_spec 34
file_specs 34
file name
  absolute 93, 95
  logical 93, 95
Fisher/Yates 629
fixed font
  Prolog 649
fixed format 276, 283
Fixed Zone 1006
flag
  shutdown 232, 555, 877
float 34
font
  ANSI fixed 745, 748
  ANSI var 745, 748
  computer 31
  device default 746, 748
  OEM fixed 745, 748
  Prolog fixed 745, 748
  system 745, 748
  system fixed 745, 748
text 31
font box 264
format
  fixed 276, 283
  free 276, 283
formatted input
formatted output
free format
Frequency Modulation
functor
games port
general protection fault
goal
GPF
GraFiX
Graphical User Interface
graphics
GUI
games port
general protection fault
goal
GPF
GraFiX
Graphical User Interface
graphics
GUI
games port
general protection fault
goal
GPF
GraFiX
Graphical User Interface
graphics
GUI
games port
general protection fault
goal
GPF
GraFiX
Graphical User Interface
graphics
GUI
input zone 371
integer 35
integer_expr 35
Intelligence Server 755
interrupt timers 663, 665, 666, 667, 668, 670
IP address 1016
ISO 986
ISO/IEC 8859-1 986

J
Java 754
joystick port 1026

K
kernel 94, 247, 622, 870

L
language extensions 205
linear congruential 572, 980
list 35
list_of 35
little endian 294, 523, 983
local cut 55
logical dynamic 202
logical file name 93, 95
logical reading 269
logical window handle 424, 564, 791
long MIDI message 1027
LZSS data compression 260, 640, 978

M
main window 585, 591
Marsaglia Zaman 184, 208, 629
MD5 172, 413, 578
MDI 29, 831, 846
memory adjustment 847
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory image</td>
<td>849, 851</td>
</tr>
<tr>
<td>message box</td>
<td>420</td>
</tr>
<tr>
<td>Message Digest Five</td>
<td>172, 413, 578</td>
</tr>
<tr>
<td>message loop</td>
<td>870</td>
</tr>
<tr>
<td>Microsoft</td>
<td>364, 365</td>
</tr>
<tr>
<td>Microsoft Bitmap</td>
<td>126</td>
</tr>
<tr>
<td>MIDI</td>
<td>1024</td>
</tr>
<tr>
<td>long message</td>
<td>1027</td>
</tr>
<tr>
<td>short message</td>
<td>1027</td>
</tr>
<tr>
<td>MIDI events</td>
<td>1028</td>
</tr>
<tr>
<td>MIDI event handler</td>
<td>1032</td>
</tr>
<tr>
<td>Midnight Rollover Bug</td>
<td>662, 973</td>
</tr>
<tr>
<td>Millennium Bug</td>
<td>973</td>
</tr>
<tr>
<td>Moog</td>
<td>1024</td>
</tr>
<tr>
<td>Robert</td>
<td>1024</td>
</tr>
<tr>
<td>multi-touch input</td>
<td>678</td>
</tr>
<tr>
<td>Multiple Document Interface</td>
<td>846</td>
</tr>
<tr>
<td>music</td>
<td>1033</td>
</tr>
<tr>
<td>Musical Instrument Digital Interface</td>
<td>1024</td>
</tr>
<tr>
<td>MZ55 data encryption</td>
<td>184, 208, 980</td>
</tr>
<tr>
<td>named variable</td>
<td>214, 229, 609, 646</td>
</tr>
<tr>
<td>negation</td>
<td>269</td>
</tr>
<tr>
<td>Node ID</td>
<td>1050</td>
</tr>
<tr>
<td>number</td>
<td>35</td>
</tr>
<tr>
<td>object_handle</td>
<td>35</td>
</tr>
<tr>
<td>OEM</td>
<td>982, 986</td>
</tr>
<tr>
<td>OEM fixed font</td>
<td>745, 748</td>
</tr>
<tr>
<td>OLE Automation</td>
<td>755</td>
</tr>
<tr>
<td>open box</td>
<td>477</td>
</tr>
<tr>
<td>optimising compiler</td>
<td>42</td>
</tr>
<tr>
<td>overlay</td>
<td>94</td>
</tr>
</tbody>
</table>
page increment 814
path_alias 35
plaintext 184, 208
Port 1011
port
  exit 373
games 1026
  joystick 1026
pred_spec 35
pred_specs 35
print box 504
pro386w.exe 247
procedural reading 269
progress box 498, 501, 550
Prolog fixed font 649, 745, 748
Prolog kernel 94
Prolog overlay 94
protocol
  TCP 1012
  UDP 1012
ProWeb 755
pseudo random number generator 36, 184, 208, 572, 980

random number generator 184, 208, 572, 980
RAW 986
raw window handle 564, 791
reading
  logical 269
  procedural 269
reverse polish notation 271, 552
Rich Syntax Colouring 548, 549, 550
Robert Moog 1024
RPN 271, 552
save box
sck_accept
sck_close
sck_connect
sck_error
sck_oob
sck_read
sck_write
search
  string
Secure Hash Algorithm
server
SHA-256
short MIDI message
shutdown flag
socket
  Asynchronous
  Blocking
  Datagram
  non-blocking
  Raw
  Stream
Sockets
socket event
  sck_read
  sck_write
socket handler
standard ordering
standard ordering of terms
Standard Syntax
Status Bars
stream
string
string search
supervisor loop
<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>system fixed font</td>
<td>745, 748</td>
</tr>
<tr>
<td>system font</td>
<td>745, 748</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>565, 566, 569, 570, 595, 616, 618, 620, 1011</td>
</tr>
<tr>
<td>TCP/IP Library</td>
<td>755</td>
</tr>
<tr>
<td>TCP protocol</td>
<td>1012</td>
</tr>
<tr>
<td>term</td>
<td>35</td>
</tr>
<tr>
<td>text font</td>
<td>31</td>
</tr>
<tr>
<td>timers</td>
<td>663, 665, 666, 667, 668, 670</td>
</tr>
<tr>
<td>token</td>
<td>225</td>
</tr>
<tr>
<td>tooltip</td>
<td>671</td>
</tr>
<tr>
<td>touch-enable</td>
<td>678</td>
</tr>
<tr>
<td>touch input</td>
<td>678</td>
</tr>
<tr>
<td>TreeView</td>
<td>1050</td>
</tr>
<tr>
<td>type conversion</td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td>383</td>
</tr>
<tr>
<td>UDP protocol</td>
<td>1012</td>
</tr>
<tr>
<td>unary functions</td>
<td>271, 552</td>
</tr>
<tr>
<td>Unicode</td>
<td>982, 986</td>
</tr>
<tr>
<td>user device</td>
<td>41, 115</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-16LE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-32BE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-32LE</td>
<td>984</td>
</tr>
<tr>
<td>UTF-8</td>
<td>983</td>
</tr>
<tr>
<td>UTF-B5</td>
<td>989</td>
</tr>
<tr>
<td>variable</td>
<td>35</td>
</tr>
<tr>
<td>anonymous</td>
<td>214, 220, 229, 609, 614, 646</td>
</tr>
<tr>
<td>named</td>
<td>214, 229, 609, 646</td>
</tr>
<tr>
<td>varlist</td>
<td>35</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>754</td>
</tr>
<tr>
<td>W</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Walter Carlos</td>
<td>1024</td>
</tr>
<tr>
<td>WEBLOG</td>
<td>850, 852</td>
</tr>
<tr>
<td>web browser</td>
<td>364, 365</td>
</tr>
<tr>
<td>Win32 API</td>
<td>45, 754</td>
</tr>
<tr>
<td>window</td>
<td></td>
</tr>
<tr>
<td>console</td>
<td>41, 135, 584</td>
</tr>
<tr>
<td>main</td>
<td>585, 591</td>
</tr>
<tr>
<td>window_handle</td>
<td>35</td>
</tr>
<tr>
<td>Windows Sockets</td>
<td>565, 566, 569, 570, 595, 616, 618, 620, 1011</td>
</tr>
<tr>
<td>Windows Vista</td>
<td>364, 365</td>
</tr>
<tr>
<td>window handle</td>
<td></td>
</tr>
<tr>
<td>logical</td>
<td>424, 564, 791</td>
</tr>
<tr>
<td>raw</td>
<td>791</td>
</tr>
<tr>
<td>window handler</td>
<td>368, 580, 757, 870</td>
</tr>
<tr>
<td>window messages</td>
<td>592</td>
</tr>
<tr>
<td>window style</td>
<td>858</td>
</tr>
<tr>
<td>Winsock</td>
<td>565, 566, 569, 570, 595, 616, 618, 620, 1011</td>
</tr>
</tbody>
</table>

| X          | | |
|------------|--------------------|
| X64        | 29, 553, 699, 1061 |
| X86        | 29, 553, 699, 1061 |
| XML        | 563                |

| Y          | | |
|------------|--------------------|
| Y2K        | 973                |

| Z          | | |
|------------|--------------------|
| Zero Width No Break Space | 983                |
| Zone       |                    |
| Fixed      | 1006               |
| Input      | 1006               |
| ZWNBSP     | 983                |