WIN-PROLOG

4.2

Win32 Programming Guide

by Brian D Steel

Brian D Steel

WIN-PROLOG Win32 Programming Guide

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)

Logic Programming Associates Ltd
Studio 4
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England

phone:	**+44 (0) 20 8871 2016**
fax:	**+44 (0) 20 8874 0449**
email:	**support@lpa.co.uk**
web:	**http://www.lpa.co.uk**

Table of Contents

Introduction

Welcome to **LPA-PROLOG** for Win32 (**WIN-PROLOG**)! This exciting version of LPA's acclaimed Prolog compiler is the successor to the previous **LPA-PROLOG** for Windows. The previous system provided a 32-bit programming environment within the Win16 API (Application Programming Interface) of Windows 3.1, while the new version is a true Win32 application, working directly with the 32-bit Windows NT and Windows 95 API. As well as providing access to as much of your machine's memory as you want (previously limited to two 16Mb segments), **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, 32-bit Prolog compiler and programming environment. Fully conforming to the Edinburgh standard, a complete set of Clocksin and Mellish predicates is augmented to provide a high degree of compatibility 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 Win32 GUI and 32-bit DLLs, DDE and OLE automation applications.

WIN-PROLOG provides an MDI-compliant programming environment, featuring multiple program edit windows, 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 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 Windows programming features of **WIN-PROLOG**, and should be used in conjunction with the Technical Reference and 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, 18 Sep 01

Chapter 1 - WIN-PROLOG

This chapter serves as an introduction to WIN-PROLOG and compares it with its sister product, DOS-PROLOG. A number of topics are dealt with in brief: later chapters deal with specific subjects in greater detail.

WIN-PROLOG and DOS-PROLOG

As mentioned above, there are in fact two versions of LPA-PROLOG (three if you include the old Win16 version): these two products are based upon a single 32-bit Prolog kernel, and differ only in their operating system interfaces. WIN-PROLOG runs on any Intel-hosted Win32 environment, including Windows NT 3.51, NT 4.0 and 95. It can also run under Windows 3.1 with the help of Microsoft's Win32s subsystem. DOS-PROLOG, on the other hand, runs directly under DOS on 386 or better machine, using an integral DOS extender.

Around 90% of the internal programming code is shared between these versions, assuring both exceptional compatibility between the Windows and DOS platforms, and guaranteed parallel development: whenever new features are added to the kernel, they are automatically included in both versions.

The Anatomy of LPA-PROLOG

Irrespective of its operating system environment, LPA-PROLOG consists of three main components: a kernel, an overlay file, and an operating system interface. A basic understanding of this anatomy will help appreciate the main differences between WIN-PROLOG and DOS-PROLOG, as well as point to those areas where total compatibility is assured.

The kernel is written entirely in 32-bit, 386 assembly language, and provides the inference engine, garbage collector, data and memory management, fundamental built-in predicates and device-independent input/output facilities. It is literally a complete Prolog system, minus the operating system interface and those predicates implemented in Prolog. Except for the operating system interface, both WIN-PROLOG and DOS-PROLOG are assembled from the same source code files, guaranteeing the continued low-level compatibility of these two products.

The overlay file is written entirely in Prolog. This provides the numerous predicates which are effectively variants or combinations of the fundamental ones built in to the kernel itself, and hooks between the kernel and the operating system. Debugging, file editing, error reporting and recovery, and many other features of the complete Prolog system are built at this level. Again, the bulk of the overlay file for both WIN-PROLOG and DOS-PROLOG is compiled from the same source code files, guaranteeing the products' intercompatibility.

The operating system interface for WIN-PROLOG is written in a mixture of 32-bit, 386 assembly language and 32-bit C, and is physically compiled into the same file as the kernel. This is the only portion of the file PRO386W.EXE which is completely independent of DOS-PROLOG. The latter's operating system interface is written completely in 32-bit, 386 assembly language, and is bound together with a third-part "DOS extender" to allow the program to run directly under MS-DOS or a DOS box. This DOS extender is simply a program which runs under MS-DOS (which is a 16-bit operating system) and provides a 32-bit environment for an application.

Differences between WIN-PROLOG and DOS-PROLOG

Two major groups of constraints govern the limits of compatibility betweenWIN-PROLOG and DOS-PROLOG: operating system features and operating limitations. The former group is perhaps the more obvious, so let’s start here.

Anyone who has used Windows will appreciate the ease with which applications can be learnt and used, since they all share a (more or less) common interface. The mouse is used to point at menu items or dialog buttons, and text windows can be scrolled and edited at will. There is no such common interface in DOS itself, and each individual application has taken its own approach to the user interface. This difference between Windows and DOS has affected the design of one key area of WIN-PROLOG: its GUI functions.

DOS-PROLOG has a powerful, but low-level set of text, graphics, mouse and keyboard handling predicates, which can be used to create fast, multi-coloured window-based applications in virtually any style. Because of their low-level nature, these predicates are very flexible, but before they can be utilised in an application, they require a considerable amount of programming.

The majority of the low-level text and graphics interface predicates are absent from WIN-PROLOG, which instead includes a number of predicates which call Win32 API functions to create and manipulate dialogs, windows, menus, fonts, graphics and so on.

Some Minor Limitations in Windows

Compared to DOS, Windows actually imposes some limitations. Some low-level features work only in DOS-PROLOG, where it is possible, for example, to program a number of mouse functions, access the video BIOS for changing screen modes, and to use hardware I/O ports to control time, sound and other attributes. Most of these functions are simply not available under Windows, which handles such matters internally. Other limitations of the previous Win16 version, such as a coarse timer resolution (55ms) and memory segment size limits (16Mb) are no longer present in the Win32 version of WIN-PROLOG.

Few of these limitations of Windows are likely to affect programs seriously, although they must be borne in mind when attempting to port between the two LPA-PROLOG platforms. By and large, the existence of these small limitations will be considerably outweighed by the ease of use and elegance of the Windows environment compared with DOS.

What's in this Manual

The remainder of this manual concentrates on WIN-PROLOG, referring to DOS-PROLOG only when necessary, to contrast a feature or note a subtle difference. Further information about WIN-PROLOG can be found in the *Prolog Programming Guide* and *Technical Reference.*

Chapter 2 - The Console Window

This chapter describes the uses of the "console" window of WIN-PROLOG, covering basic input and output, command editing, scrolling, and other features. The console window provides the fundamental interaction between WIN-PROLOG and the user.

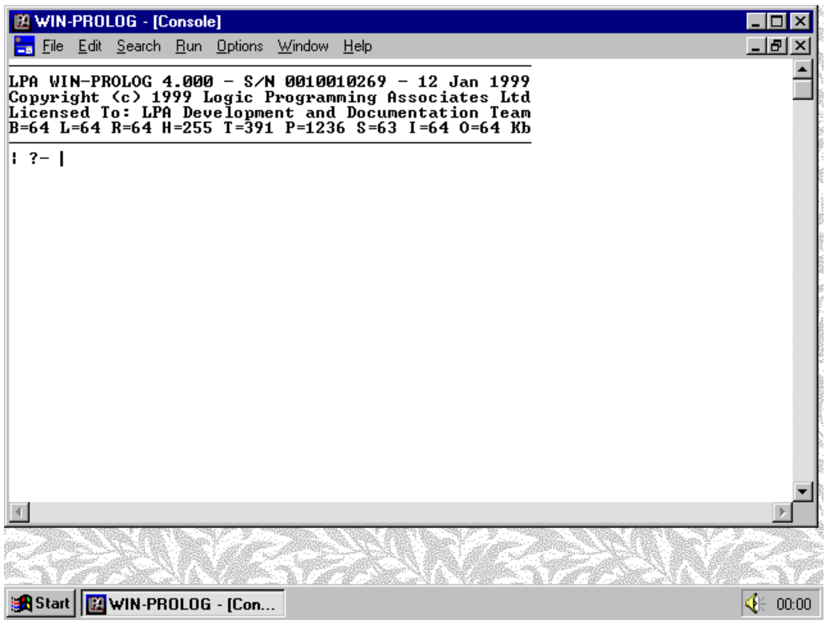
The Main and Console Windows

When WIN-PROLOG starts up, it initially displays a graphical window showing a welcome banner, as shown in*Fig 2.1*, which remains on the screen for a short period while the kernel and overlay files are loaded. Once WIN-PROLOG is ready for input, it removes this banner, replacing it with two windows, the "main" and "console" windows. Initially, the two windows appear as one, as

Fig 2.1 - The WIN-PROLOG welcome banner

shown in *Fig 2.2*. This is because WIN-PROLOG is a Multiple Document Interface (MDI) application, and the main window surrounds and contains all other windows, including the console. Initially, the console is maximised to fill the entire main window client area.

The main and console windows are WIN-PROLOG's command centre: all operations are launched from either the main window's menus or by commands typed into the console's client area. The menus provide the usual collection of file, edit, search, window and help commands, together with a number of run and options commands which include compilation, syntax checking and debugging features. The menu functions are described in the WIN-PROLOG



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- |
```

Fig 2.2 - The WIN-PROLOG main and console windows

User Guide: the present manual is concerned with the fundamental behaviour and programming aspects of *WIN-PROLOG*.

The client area of the console window is a scrollable edit control which functions just like a "glass teletype": you can type commands, list programs, and use standard Prolog output predicates to display results and other data. Unlike a traditional glass teletype, however, you can scroll back over data which has disappeared from view, modify and edit any text (including output), and use cut-and-paste features to edit and re-enter previous commands.

Typing Commands

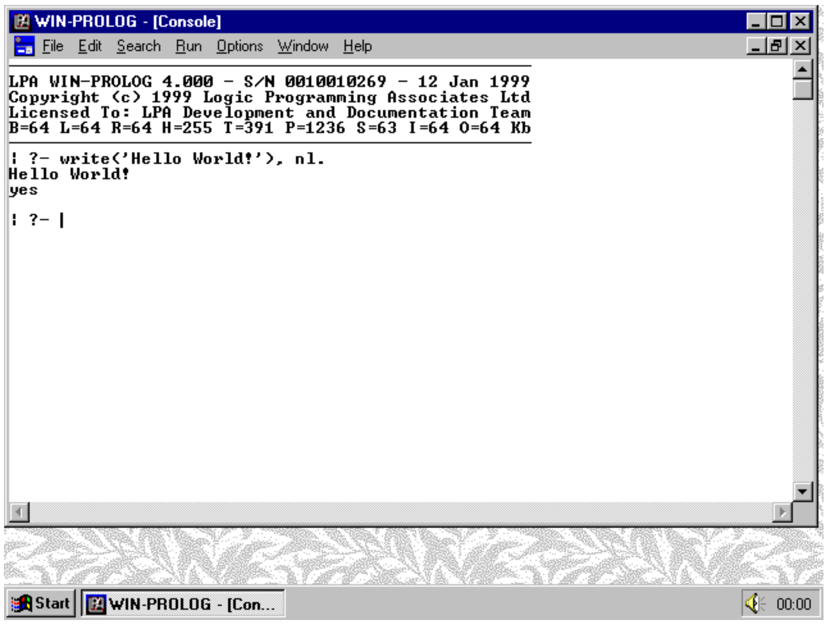
You type commands into the console window just like you would into any glass teletype application, using the keyboard and pressing <*enter*> when you are ready to submit your input. Each time you press <*enter*>, the line of text you are working on is given to Prolog. For example, type the characters:

?- **write('Hello World!'), nl.** <*enter*>

In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in <*italics*>. The result of typing this command is shown in *Fig 2.3*. Any Prolog command or query can simply be typed and entered, line by line, in this way.

Editing Commands

If you make a mistake during input, you can use the <*backspace*> key to correct it, or the mouse or cursor keys to reposition the cursor to insert text or delete



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- |
```

Fig 2.3 - A simple command in the console window

mistakes. You can also cut and paste text from anywhere within the console window to build complex commands. In fact, you can cut or paste data between this and any other text window on the screen which supports the Windows clipboard (though see below for a discussion about character sets and international language support).

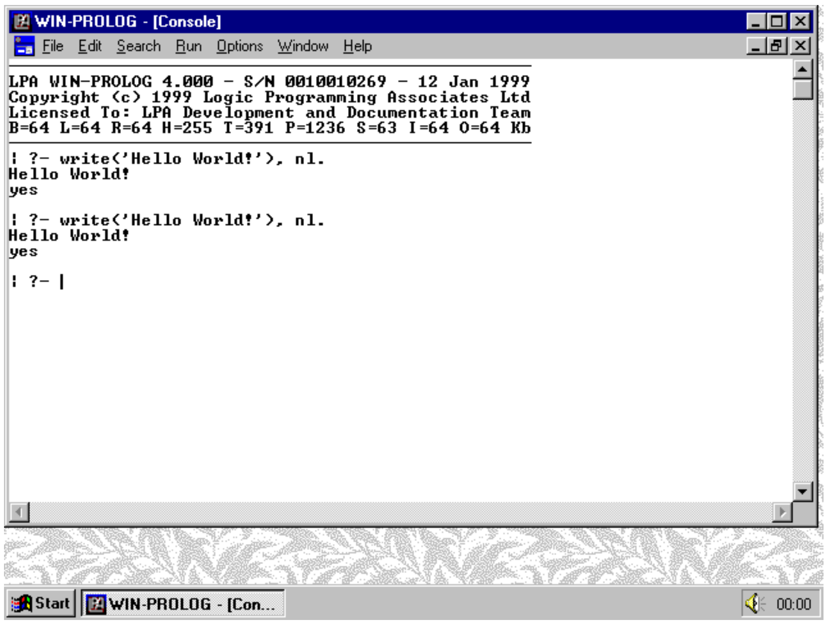
Please note: the keystrokes and mouse operations required to perform editing are defined by Windows itself, and so are beyond the scope of the current manual. If you are new to Windows, you should consult your Windows User Guide or help files for further information.

Re-entering Commands

As mentioned above, when you type <*enter*>, the line of text you are working on is given to Prolog. This feature is not limited to the last line of text in the console window: if you move the caret (the flashing Windows cursor) to any line of text, and then press <*enter*>, that line will be copied to the bottom of the console window, and submitted automatically. To try this out, click the mouse somewhere in your "write('Hello World!'), nl." command, without dragging a text selection area, and then press<*enter*>. This will have the effect of retyping your command, as shown in *Fig 2.4*.

You do not have to reenter the command as it stands: you can edit it before pressing <*enter*>. For example, using the mouse, drag a selection box over the word "World" in one of your copies of the command, as show in *Fig 2.5*. Then, type the word (**bold** letters only):

?- write('Hello **There**!'), nl. <*enter*>



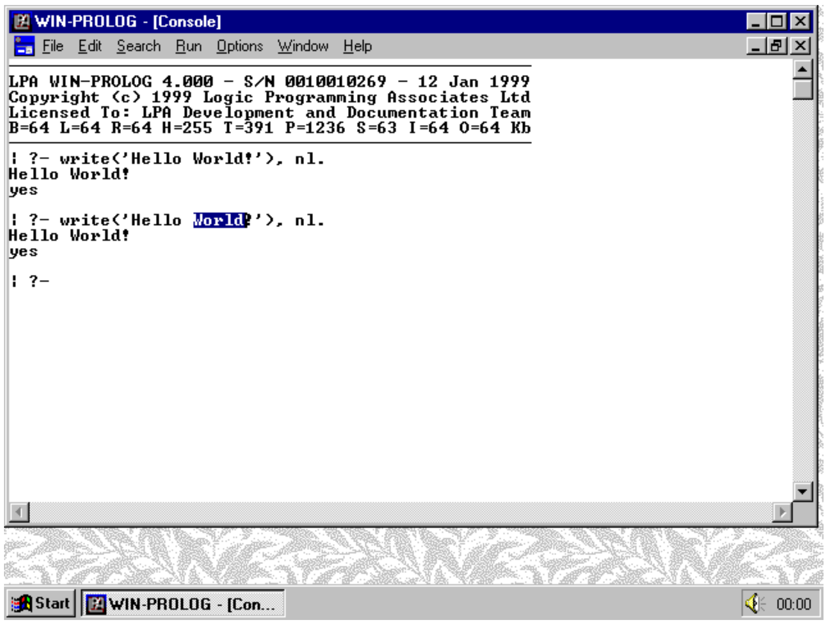
```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- |
```

Fig 2.4 - Re-entry of a command in the console window

followed by *<enter>*. This time, your edited version of the command will be copied and executed, as shown in *Fig 2.6*. This editable command reentry is a key feature of *WIN-PROLOG*.

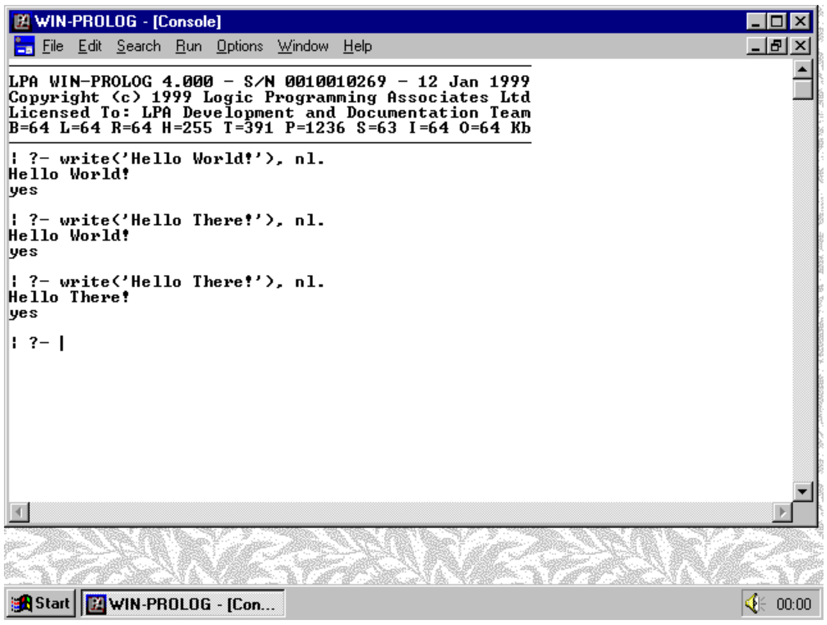
Multi-line Commands

Up until now, if you have been following the instructions, you will have entered commands of exactly one line's length. There will be times when you only want to enter part of a line: this can of course be achieved by cutting and pasting, or



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?-
```

Fig 2.5 - Preparing to edit a command for reentry



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello There!
yes

| ?- |
```

Fig 2.6 - The result of reentering an edited command

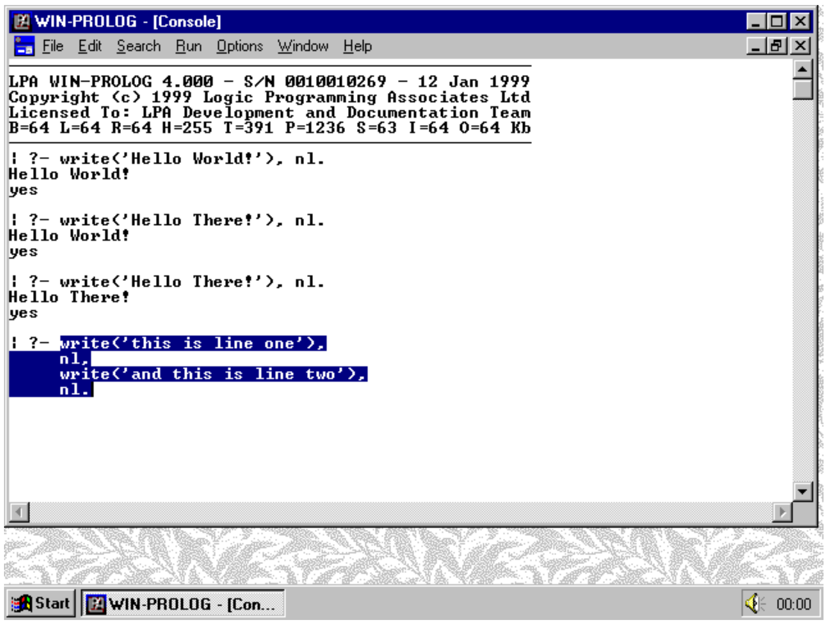
by deleting the parts of the line you do not want, but there is an easier method: if you use the mouse to drag a selection box around the bit of the command you want to submit, and then press <*enter*> to enter, you will submit just the bit of text that is highlighted.

The identical technique can be used to enter multi-line commands. Remember that each time you press <*enter*>, either the current line or (if there is one) the selection area is copied to the bottom of the console and submitted to Prolog.

If you want to type a complex command over several lines, with the option of editing any part of the command before finally submitting it as a whole, you can do so by pressing <*ctrl*> and <*enter*> together at the end of each line, rather than just <*enter*>. Remember, the <*enter*> key means "submit this line (or selection area", while <*ctrl-enter*> means "insert a new line in the text". When you are ready to submit the whole command, simply drag the mouse to enclose it in a selection box, and press <*enter*>. Try the following simple example for practice:

```
?- write('this is line one'),                     <ctrl-enter>
      nl,                                         <ctrl-enter>
      write('and this is line two'),              <ctrl-enter>
      nl.
```

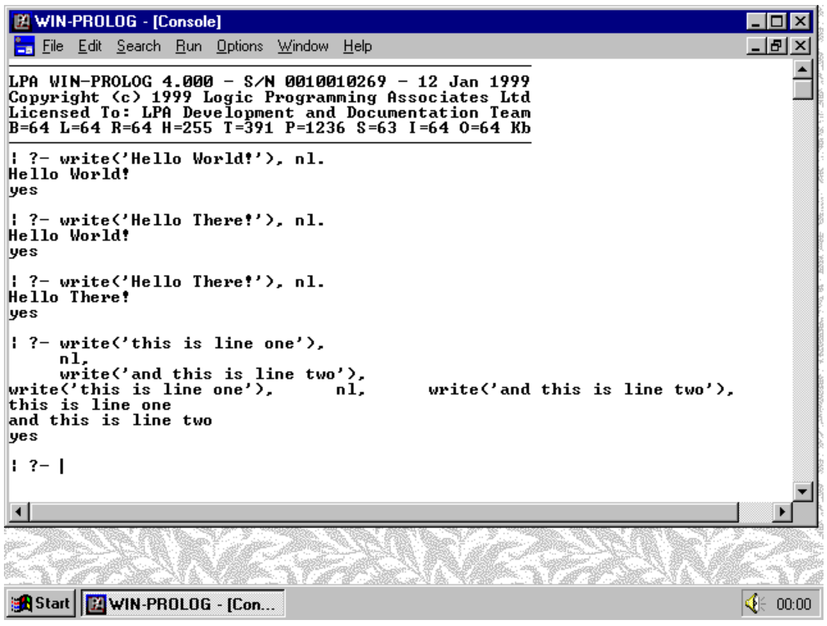
Once you have typed this, using <*ctrl-enter*> rather than just <*enter*> at each line break, use the mouse to drag a box around the whole command, as shown in *Fig 2.7*. Next, press <*enter*> to submit the whole command at once: your selection will be copied into a single command line at the bottom of the console window, and executed as shown in *Fig 2.8*.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello There!
yes

| ?- write('this is line one'),
     nl,
     write('and this is line two'),
     nl.
```

Fig 2.7 - Entering a multi-line command

The size of any one submission is limited to 1kb (1024 characters), which is the maximum physical length of a single line, although a command may be made up of any number of smaller entries. In the example just shown, you could have pressed *<enter>* after each line: the difference is that you would have been submitting the command line by line, and if, for example, you spotted an error in line 1 while typing line 3, it would have been too late to fix it other than by breaking in and starting over (see next section).



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello There!
yes

| ?- write('this is line one'),
     nl,
     write('and this is line two'),
write('this is line one'),      nl,      write('and this is line two'),
this is line one
and this is line two
yes

| ?- |
```

Fig 2.8 - The result of entering a multi-line command

Breaking In

If you make a mistake while entering a command, or the system appears to ignore your input, perhaps because you have left a quote or a bracket unclosed, you can break in by typing the *<ctrl>* and *<break>* keys together. A message will be shown to say you have interrupted execution, giving you the option to start over at the "?-" prompt.

The *<ctrl-break>* key combination is used throughout **WIN-PROLOG**, not only to interrupt the command line editor, but also to break in to programs which have entered infinite loops, or which are just performing long computations that you decide to abort.

Clearing the Console Window

The console window never needs to be cleared out: as it gradually fills to capacity (somewhere between 32..48kb), it automatically discards data from the top as more space is needed. Apart from a brief flicker while it performs this task, you will not know that the clean-out has occurred.

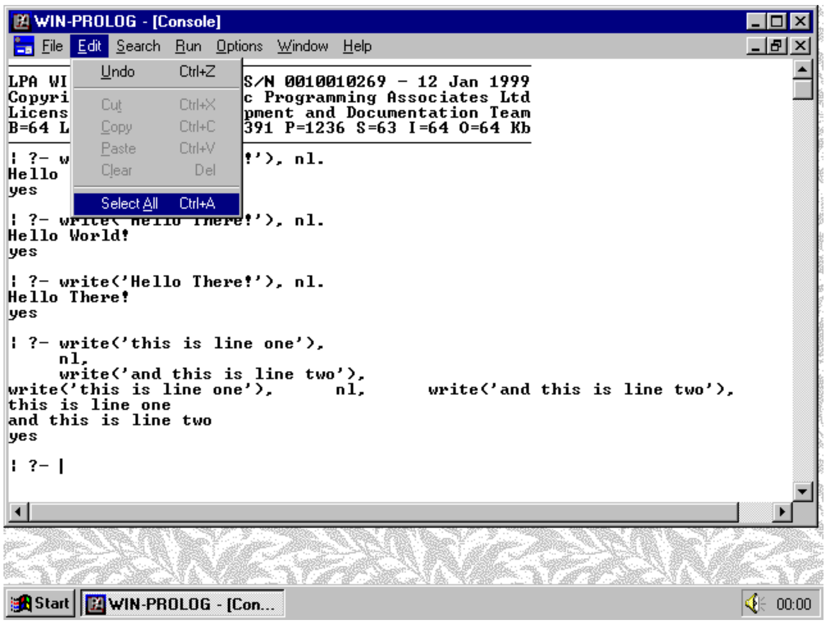
Should you wish to, you can, of course, clear the console window at any time, using any one of a number of methods. The simplest way is to select all the text in the window and then press *<backspace>* or *<del>*. You can select the whole window either by dragging the mouse, by choosing the "select All" option from the "Edit" menu, or by typing *<ctrl-A>*.

For the sake of making the examples in the rest of this manual easier to see in the illustrations, we will be clearing out the console window at regular intervals. Try it now: using the mouse, click on "Edit" on the console window menu, and choose the "select All" option, as shown in *Fig 2.9*.

When you have completed your menu choice, the whole window will be highlighted as shown in *Fig 2.10*. Now press *<backspace>* to delete the highlighted text, and to give yourself a clean console as shown in *Fig 2.11*.

Output to the Console Window

As you will have seen from the examples above, writing to the console window is simply a matter of using standard Prolog output predicates. There are other, more advanced ways in which to perform output to this and any other window: these will be discussed fully in *Chapter 3*. Staying for now with standard output predicates, there are some considerations to bear in mind when comparing the behaviour of **WIN-PROLOG** with other glass teletype applications. The next three sections discuss these points.

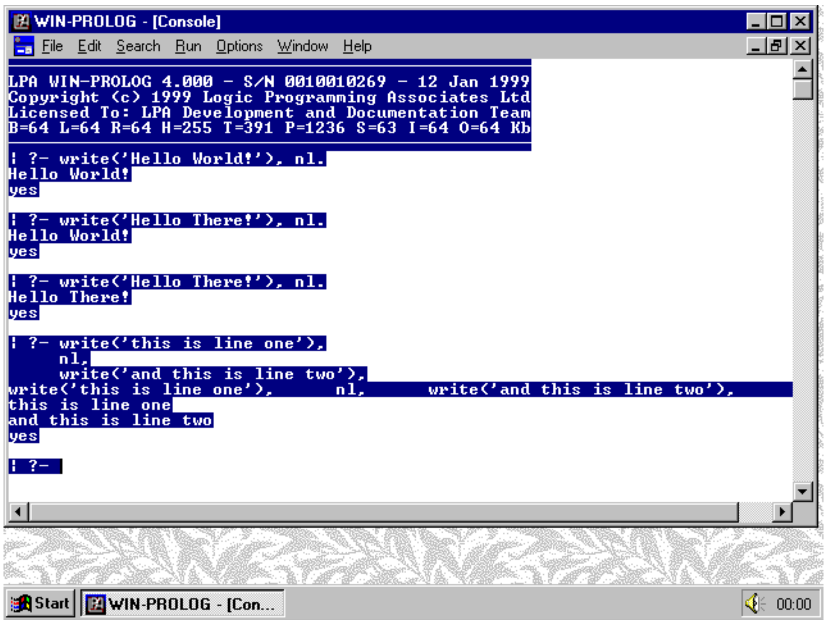


```
LPA WI          S/N 0010010269 - 12 Jan 1999
Copyri          c Programming Associates Ltd
Licens          pment and Documentation Team
B=64 L          391 P=1236 S=63 I=64 O=64 Kb

| ?- w          !'), nl.
Hello
yes

| ?- write('Hello There!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello There!
yes

| ?- write('this is line one'),
     nl,
     write('and this is line two'),
write('this is line one'),      nl,      write('and this is line two'),
this is line one
and this is line two
yes

| ?- |
```

Fig 2.9 - Selecting text with the "Edit/select All" option

Output Buffering

One of the problems with Windows is that individual output operations are very slow. In particular, it simply would not be practical to update the screen every time a character was output: for this reason, output to the console window is buffered. The output buffer holds up to 1kb, and when it is full, the whole block is displayed at once. The buffer is also flushed whenever an input predicate tries to read from the console window, so that any prompts are up to date.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- write('Hello World!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello World!
yes

| ?- write('Hello There!'), nl.
Hello There!
yes

| ?- write('this is line one'),
     nl,
     write('and this is line two'),
write('this is line one'),      nl,      write('and this is line two'),
this is line one
and this is line two
yes

| ?-
```

Fig 2.10 - The result of selecting all text

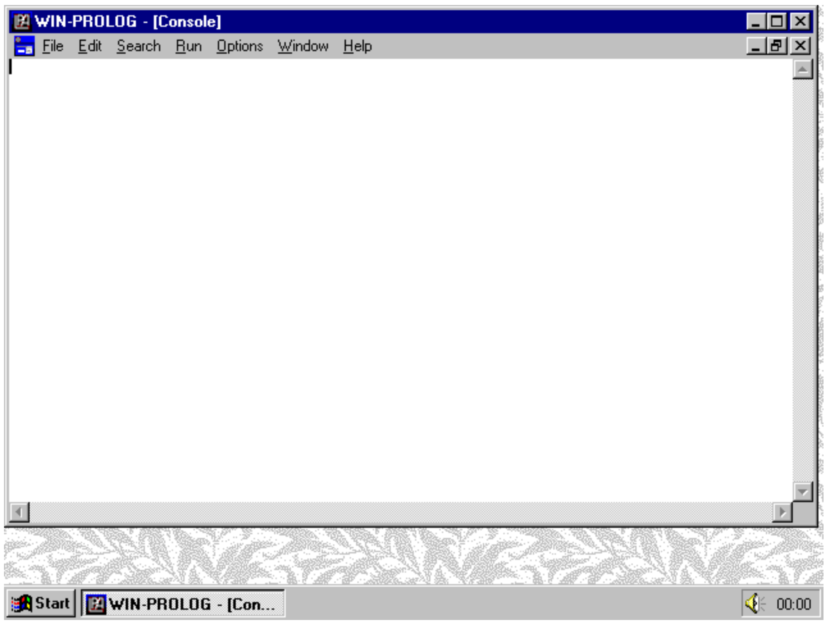


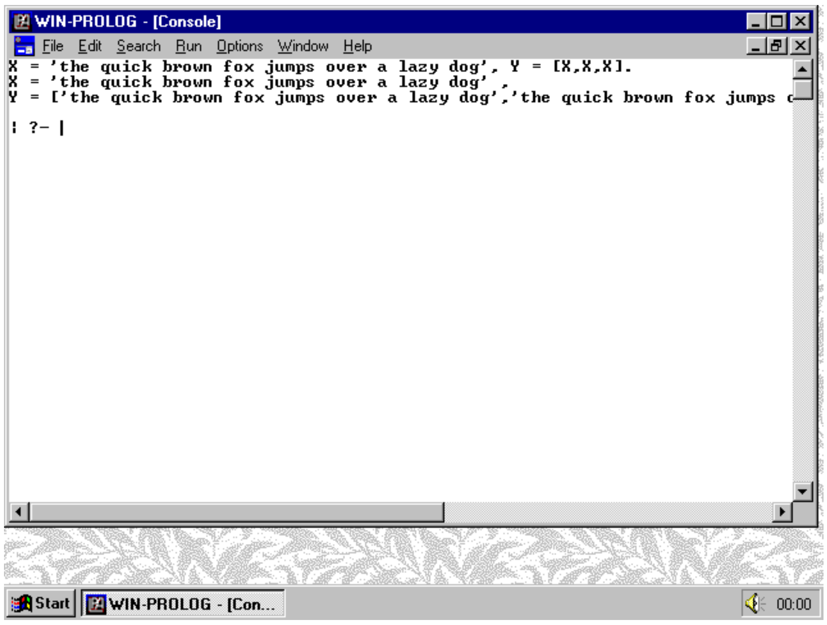
Fig 2.11 - The console window cleared of all text

Normally, you will not need to worry about this buffering, since its only effect will be to speed up output: occasionally, however, you might want output to be displayed independently of input, for example when printing diagnostic messages. By "displaying" the end of file character, *<ctrl-Z>* (code 26), you can flush the output buffer. The simplest way of flushing the buffer would be to use the *putb/1* predicate with "26" as its argument (see *Appendix C* for further discussion of buffering). The *ttyflush/0* predicate also performs this operation.

Two-way Scrolling

The console window is equipped with both vertical and horizontal scroll bars, and has a logical width of around 1024 columns and a logical height limited only by available memory. The console window may hold up to 48kb (the exact figure depends upon external factors including the version of Windows being used), corresponding to about 48 full-width text lines, or, say, rather over 1000 lines of 40 column text. Whenever **WIN-PROLOG** is waiting for input, you can use the vertical scroll bar to review previous output, up to the limit stored. You cannot scroll the window during output, because this could cause information to be missed, or even written into the wrong part of the console window. All output goes to the bottom of the console window, and if you attempt to scroll during output, your actions will be overridden by **WIN-PROLOG**!

Unlike some glass teletype applictions, text does not automatically wrap when it reaches the border of a window, so any long lines of output, such as large lists, will simply disappear off the right hand edge. Character-level wrapping will occur if a line exceeds the logical width of the window (around 1024 columns). Depending upon your video adapter and monitor, as well as the size of the console window, you may be able to see up to around 75-150 columns of text



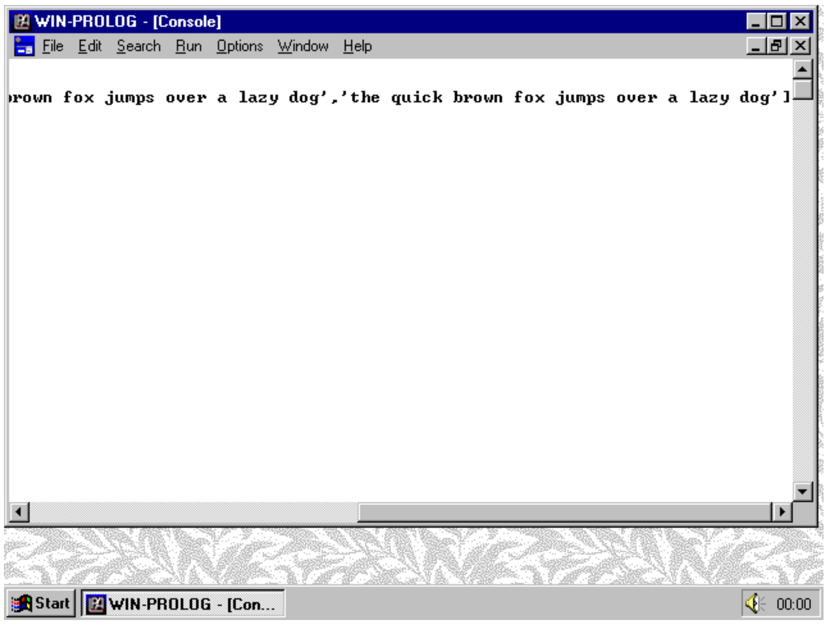
```
X = 'the quick brown fox jumps over a lazy dog', Y = [X,X,X].
X = 'the quick brown fox jumps over a lazy dog' ,
Y = ['the quick brown fox jumps over a lazy dog','the quick brown fox jumps c

| ?- |
```

Fig 2.12 - Output which is too wide for the console

at any one time: you will need to use horizontal scrolling to view the ends of long lines of text. Try the following example:

X = 'the quick brown fox jumps over a lazy dog', Y = [X,X,X]. *<enter>*

This command creates a list in *Y* which is too long to display all at once, so the end of the result will disappear off the right border of the window, as shown in *Fig 2.12*. By using the horizontal scroll bar, you can move the text area around to see more of your output, as shown in *Fig 2.13*.



```
rown fox jumps over a lazy dog','the quick brown fox jumps over a lazy dog']
```

Fig 2.13 - The console scrolled to reveal a long text line

Some glass teletype programs rely on screen-edge text wrapping to create formatted tables or the like; this approach is at best risky, since such programs often support multiple screen widths. In **WIN-PROLOG**, this trick will not work at all: when you want text to appear on a new line, you must output a new line, for example by using the *nl/0* predicate.

Control Characters

Many glass teletype programs use control characters (those in the range 00h..1fh) to perform special functions. For example, "displaying" the*<linefeed>* character (0ah) will move the cursor current column in the next line. Like most other control characters, this behaves differently under Windows.

In **WIN-PROLOG**, most control characters display graphic characters, rather than perform a function as in **DOS-PROLOG**. In *Table 2.1* you will find the list of control characters which perform special functions under **WIN-PROLOG**.

Table 2.1 - Control Characters used in Console Window Output

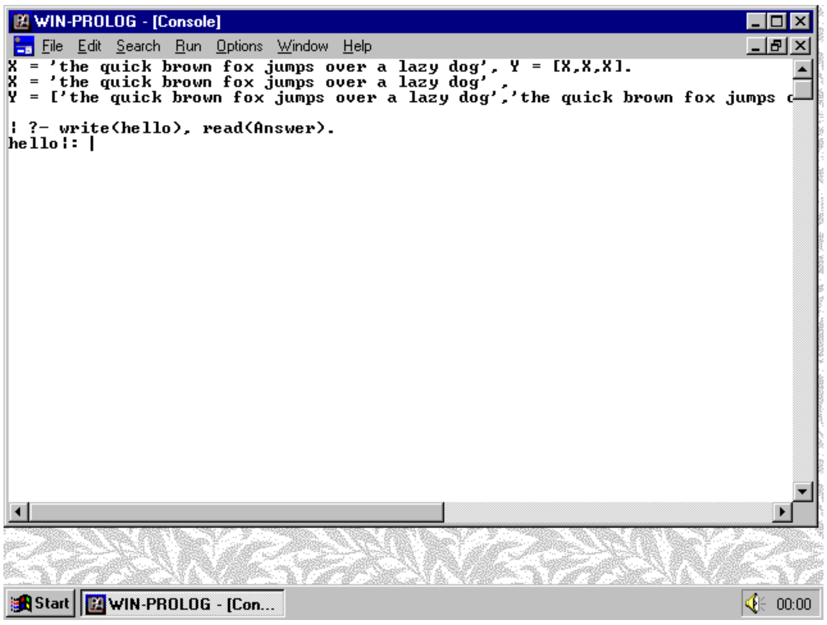
Character	Code	Name	Action
<ctrl-@>	00h	Null	Output a space character
<ctrl-G>	07h	Bell	Emit the standard beep
<ctrl-H>	08h	Back Space	Delete last character
<ctrl-I>	09h	Tab	Perform indentation to next tabstop
<ctrl-J>	0ah	Line Feed	Ignored (see <ctrl-M>)
<ctrl-L>	0ch	Form Feed	Clear whole of console window
<ctrl-M>	0dh	Carriage Return	Output a carriage return/line feed
<ctrl-Z>	1ah	End of File	Flush output buffer to console window

Most other control characters result in their graphical representations being displayed.

Discriminating between Input and Output

Unlike most glass teletype applications, where you are typically unable to move the cursor outside of the command you are typing, you can move freely around **WIN-PROLOG**'s console window, and edit, cut, paste and generally play about with its text. All this freedom presents **WIN-PROLOG** with a puzzle: when you finally type *<enter>* to submit your command, how does the system separate any prompt that might have been displayed from what you have typed?

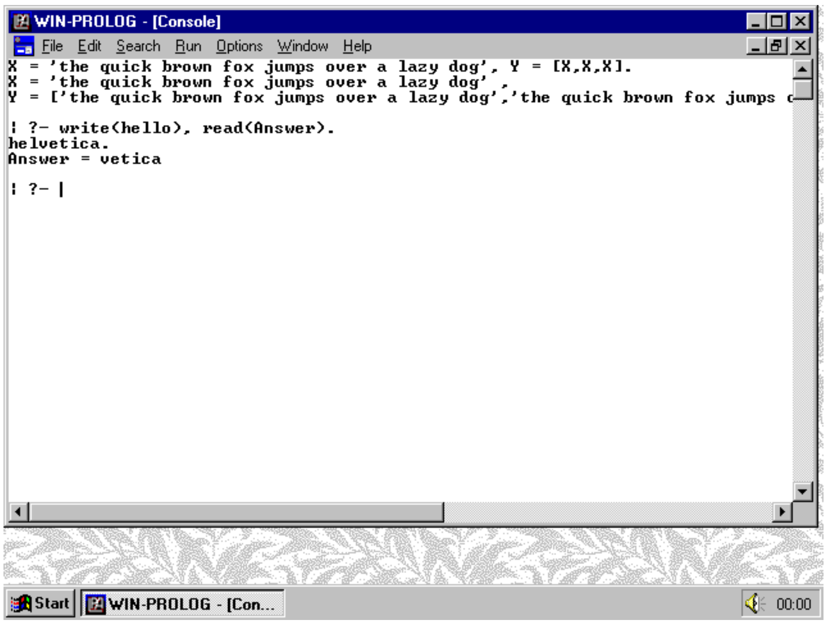
Once a read operation is underway, and before the mouse and keyboard have been enabled, **WIN-PROLOG** stores a copy of the text in the last line of the console window. When you submit your command, by typing *<enter>*, **WIN-PROLOG** compares your input with its stored copy of the original line, and ignores any leading characters which match.



```
X = 'the quick brown fox jumps over a lazy dog', Y = [X,X,X].
X = 'the quick brown fox jumps over a lazy dog' ,
Y = ['the quick brown fox jumps over a lazy dog','the quick brown fox jumps c

| ?- write(hello), read(Answer).
hello|: |
```

Fig 2.14 - Preparing to read from a prompt

To illustrate this point, consider the case where the standard "?-" prompt is displayed, and **WIN-PROLOG** is waiting for a command. **WIN-PROLOG** saves these characters in an internal buffer, enables the mouse and keyboard editing functions, and waits for you to press *<enter>*. When you have typed and submitted your command, the whole line (including "?-") is returned. **WIN-PROLOG** compares this line with the stored copy, and finding that "?-" was present before you began typing, removes these characters from the input before processing the remainder of the line. Type the following example, which helps show this:



```
X = 'the quick brown fox jumps over a lazy dog', Y = [X,X,X].
X = 'the quick brown fox jumps over a lazy dog' ,
Y = ['the quick brown fox jumps over a lazy dog','the quick brown fox jumps c

| ?- write(hello), read(Answer).
helvetica.
Answer = vetica

| ?- |
```

Fig 2.15 - The result of typing input at a prompt

```
write(hello), read(Answer).                              <enter>
```

The screen should appear as shown in *Fig 2.14*. Now, press <*backspace*> 8 times to delete the "|:" prompt and the word "hello", and then type:

```
helvetica.                                               <enter>
```

When you press <*enter*>, you might be surprised to see the answer to your query is "vetica", as shown in *Fig 2.15*. This is because the first three letters of "hello" and "helvetica" are the same. Even if you cleared the console window, or performed cut and paste operations to make up the word "helvetica" in this example, you would always get the result "vetica", because **WIN-PROLOG** ignores all leading characters of your input that match its internal copy of the line. In general, this difference algorithm works very well: confusion can only arise when you deliberately delete the prompt and then type some input which happens to begin with the same characters as the original prompt.

Chapter 3 - Text Windows

This chapter describes the "text" windows of **WIN-PROLOG**, covering their creation, manipulation, input and output, and low level programming. Text windows provide facilities for text editing, information display, and other features which go well beyond the capabilities of the console window.

Text Windows

In *Chapter 2*, we looked at the basic input and output features of the console window. While very easy to use for entering and displaying simple data, the techniques that were described are not sufficiently flexible for handling the more complex input and output requirements of many programs. For example, cursor addressing or the overwriting of text items is not possible, because output to the console window always goes to the bottom of the window.

WIN-PROLOG allows you to create "text" windows to perform several functions, including the editing of text (rather than command submission) and display of information. Since the console window is used for all standard Prolog input and output, special predicates are needed to perform input and output from text windows. These techniques will, incidentally, also work with the console window, giving considerably more power than was suggested in *Chapter 2*.

Modeless MDI Windows

Text windows are modeless, Multiple Document Interface (MDI) windows, each containing a single edit control. These windows can be moved around the screen by the user, resized, iconised, and so forth. Prolog programs can create, show, hide, resize and otherwise manipulate them, and an arbitrary number of them can exist at any one time, limited only by available memory resources.

Because text windows are modeless MDI windows, the user can select them at will, type text into their edit controls, and cut and paste text between them and any other windows (including the console window). Prolog programs can read and write any part, or all, of the contents of any text window, making it easy to load the windows' contents from disk files, save it as asserted clauses, or do pretty well anything else.

Creating a Text Window

Text windows are created with the *wcreate/8* predicate. Its arguments give the window its name, type, title, size and style. Let's try an example. If you are already in **WIN-PROLOG**, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start **WIN-PROLOG** up, and type the following command:

?- **wcreate(fred,text,`Freddie`,200,125,400,225,0).** *<enter>*

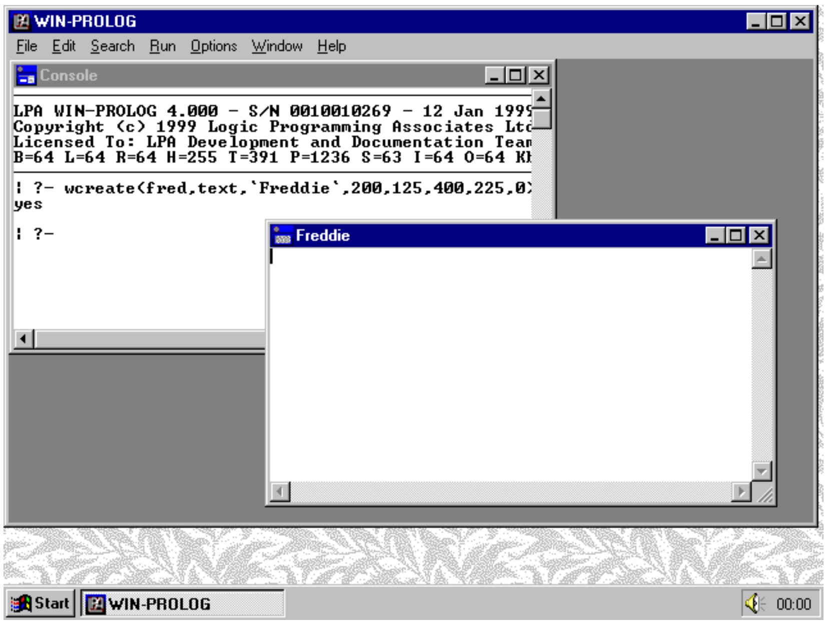
In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in <*italics*>.

This example will create a window known to Prolog as the atom "fred", and will create it as a text window with the title "Freddie". The initial position will be at (200,125) relative to the main window, and its size will be (400*225), as shown in *Fig 3.1*. The style (0) is a dummy parameter here: in text windows, the style argument is ignored.

Note the use of backwards quotes (`...`) around the title, denoting a string rather than an atom. Throughout the Windows predicates, strings are used for general text parameters. If you are not familiar with strings, and their relationships with atoms and char lists, you should refer to *Appendix B* for further information.

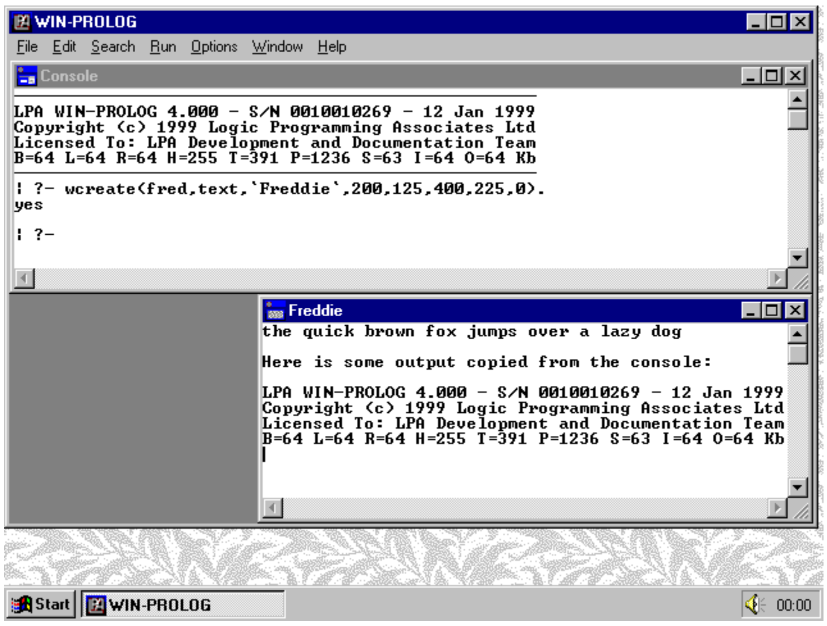
Entering Text

Now that your text window is in focus, you can begin to type text into it. Unlike the console window, where <*enter*> is used to submit a command or other input, in text windows this key simply inserts a new line into the text. Try typing some text into "fred": you can resize the window using the mouse, or cut data from the console window and paste it into "fred". After a little such playing around, the screen might look something like *Fig 3.2*.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Tean
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kl

| ?- wcreate(fred,text,`Freddie`,200,125,400,225,0)
yes

| ?-
```

Fig 3.1 - A text window created by wcreate/8



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,text,`Freddie`,200,125,400,225,0).
yes

| ?-
```

```
the quick brown fox jumps over a lazy dog

Here is some output copied from the console:

LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
```

Fig 3.2 - Sample text input after resizing the windows

Windows and Controls

At this point, we should say a few words about windows and controls. In the Windows environment, all dialogs, buttons, edit fields, scroll bars, and so forth, are actually windows, just like the WIN-PROLOG main and console windows. Many of the operations that you will be reading about here and in later chapters can be applied to all types of windows, while others are restricted to just a few window types.

Because a given operation, such as the setting of a window's text contents, can be applied to any type of window, it is necessary to be able to distinguish between a top level window or dialog and one of its controls when calling a window handling predicate. In WIN-PROLOG, all top level windows are known by a name which you give it when the window is created:

```
?- wcreate(fred,....
```

creates a top level window or dialog called "fred". All operations that you wish to perform on this window itself would use "fred" as the first argument:

```
?- wsize(fred,...
```

and so on. Controls within top level windows do not have names of their own, but rather are known by an integer ID code. To refer to a control within a given window, you simply use a conjunction of the form "(name,ID)" as the first argument:

```
?- wsize((fred,123),...
```

would refer to the control with ID "123" within the window "fred".

In text windows, there is a single predefined "edit" control, which has an ID of 1. Whenever you want to refer to the text window itself, perhaps to change its title or size, you just use the name that you gave it when you created it, for example "fred". Whenever you want to refer to its predefined edit control, perhaps to append to its text, you will use the conjunction of the name and the ID of 1, for example "(fred,1)".

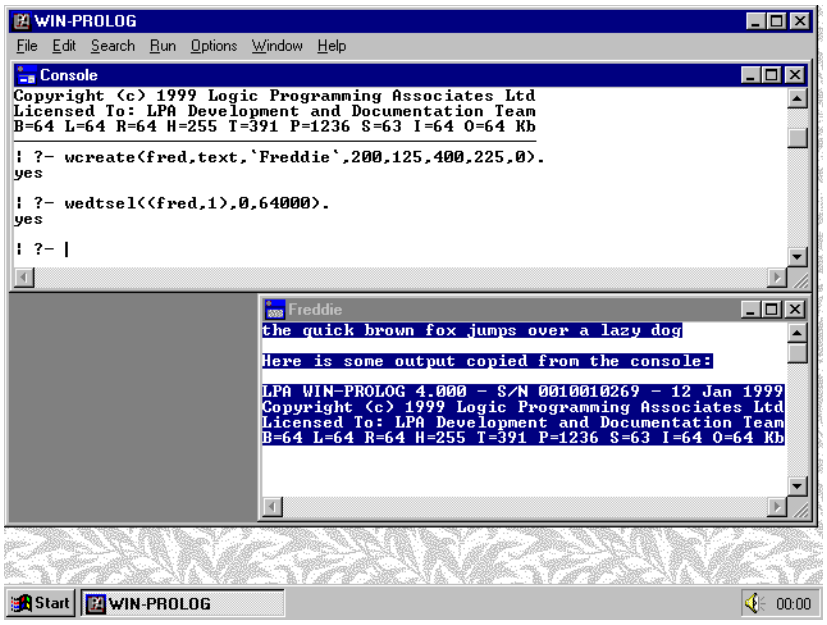
Entering Further Commands

When you want to enter another Prolog command, you must click back in the console window. This will take "fred" out of focus, and return control to the console window. Now type the following command into the console window (**bold** letters only):

```
?- wedtsel((fred,1),0,64000).                         <enter>
```

Note the use of the conjunction "(fred,1)", indicating that we want to work on the edit control (whose ID is 1) within "fred", and not "fred" itself. This creates a selection in the edit control of your text window from 0 (the beginning) to 64000 (the end), as shown in *Fig 3.3*. The reason for using 64000 is that this is greater than the maximum number of bytes a text window can hold, so will always refer to the end of the window. If you want to see how much text is actually selected, you can use the command:

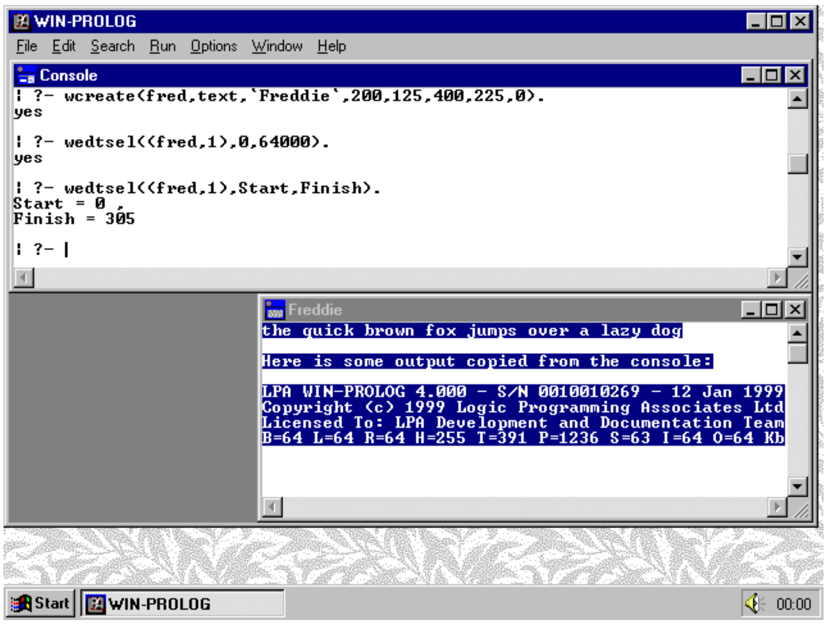
```
?- wedtsel((fred,1),Start,Finish).                    <enter>
```



```
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,text,`Freddie`,200,125,400,225,0).
yes

| ?- wedtsel((fred,1),0,64000).
yes

| ?- |
```

```
the quick brown fox jumps over a lazy dog

Here is some output copied from the console:

LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
```

Fig 3.3 - All text selected by the wedtsel/3 predicate



```
| ?- wcreate(fred,text,`Freddie`,200,125,400,225,0).
yes

| ?- wedtsel((fred,1),0,64000).
yes

| ?- wedtsel((fred,1),Start,Finish).
Start = 0 ,
Finish = 305

| ?- |
```

```
the quick brown fox jumps over a lazy dog

Here is some output copied from the console:

LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
```

Fig 3.4 - Returning the size of the contents of a text

WIN-PROLOG will return the start and finish of the selection box, as shown in *Fig 3.4*.

Reading from a Text window

Allowing the user to edit text in a window is all very well, but is pretty pointless unless there is a way to return this data to Prolog. It has already been stated that the console window alone provides support for standard Prolog input and output, so a new mechanism is needed for text windows. The *wedttxt/2* predicate is used to read (or write) the contents of the selection area of an edit control: in the current example, this is the whole of window "(fred,1)". Type the command:

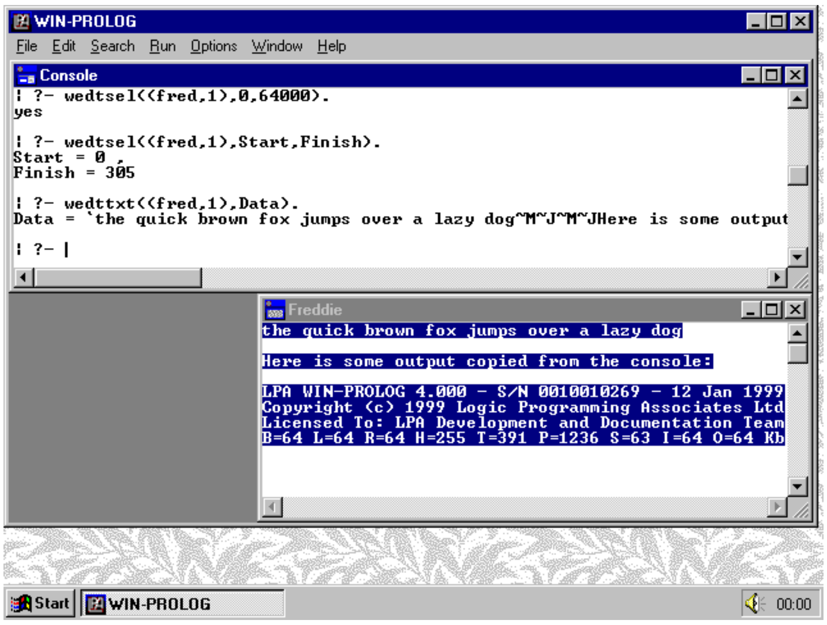
?- **wedttxt((fred,1),Data).** *<enter>*

This will collect all the data in "fred"'s edit control into an LPA-PROLOG string, as shown in *Fig 3.5*. The string data type is a compact representation used in LPA-PROLOG to store large amounts of text or even binary data, and is used for all user-defined text parameters in Windows programming.

Writing to a Text window

You can write to a text window in the same way you read from one, using *wedtsel/3* to select the area you want to modify, and *wedttxt/2* to replace the selection with a new string. Type the command:

?- **wedttxt((fred,1),`Hello World!`).** *<enter>*



WIN-PROLOG

File Edit Search Run Options Window Help

Console

```
| ?- wedtsel((fred,1),0,64000).
yes

| ?- wedtsel((fred,1),Start,Finish).
Start = 0 ,
Finish = 305

| ?- wedttxt((fred,1),Data).
Data = `the quick brown fox jumps over a lazy dog~M~J~M~JHere is some output

| ?- |
```

Freddie

```
the quick brown fox jumps over a lazy dog

Here is some output copied from the console:

LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
```

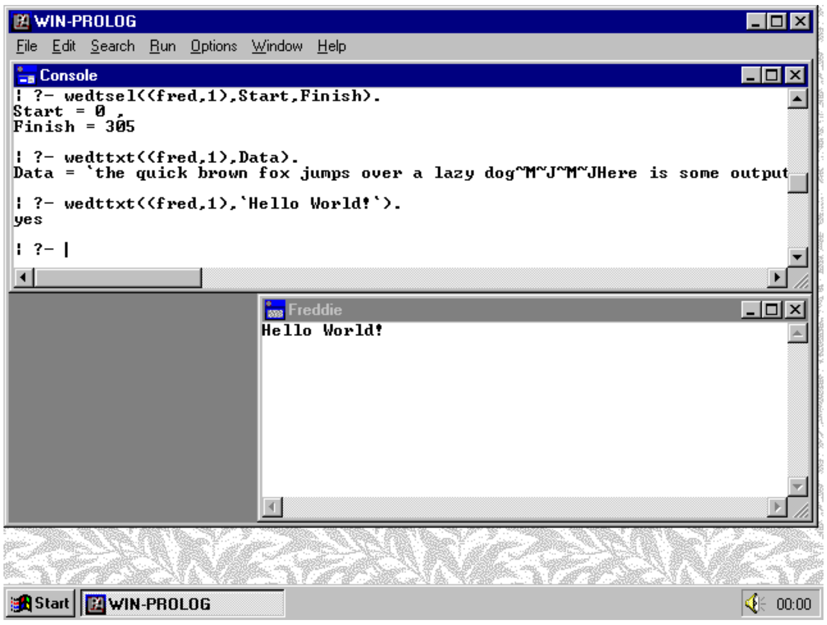
Start | WIN-PROLOG | 00:00

Fig 3.5 - Getting the selected contents of a text window

This will cause the selected text in "fred"'s edit control to be replaced with the text, "Hello World!", as shown in *Fig 3.6*. Note once again the use of backward quotes, which denote the string data type used by this predicate.

Because the *wedttxt/2* predicate simply replaces the selection area, it can be used to perform selective edits, or simply to append text to an edit control. Repeat the command:

?- **wedttxt((fred,1),`Hello World!`).** *<enter>*



WIN-PROLOG

File Edit Search Run Options Window Help

Console

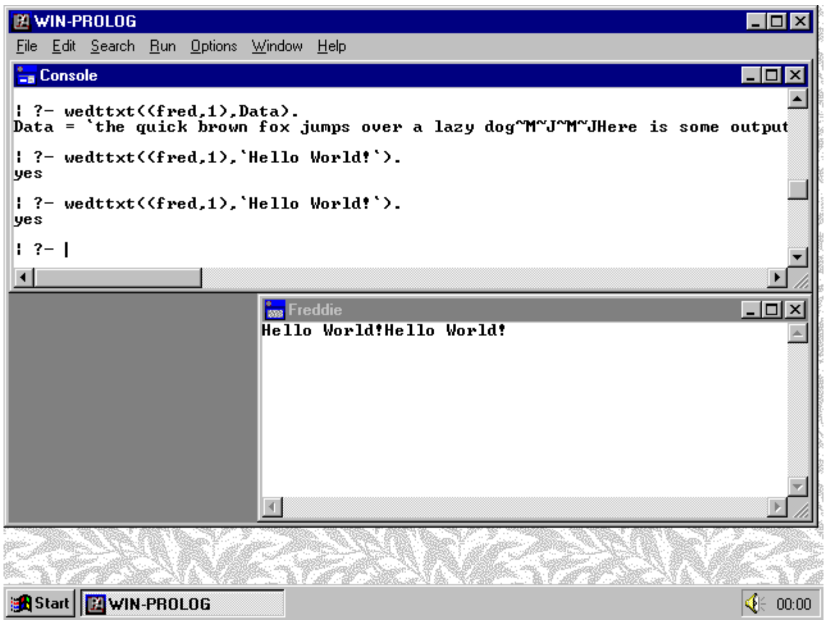
```
| ?- wedtsel((fred,1),Start,Finish).
Start = 0 ,
Finish = 305

| ?- wedttxt((fred,1),Data).
Data = `the quick brown fox jumps over a lazy dog~M~J~M~JHere is some output

| ?- wedttxt((fred,1),`Hello World!`).
yes

| ?- |
```

Freddie

```
Hello World!
```

Start | WIN-PROLOG | 00:00

Fig 3.6 - Replacing the selected contents of a text window



```
| ?- wedttxt((fred,1),Data).
Data = `the quick brown fox jumps over a lazy dog^M^J^M^JHere is some output

| ?- wedttxt((fred,1),`Hello World!`).
yes

| ?- wedttxt((fred,1),`Hello World!`).
yes

| ?- |
```

```
Hello World!Hello World!
```

Fig 3.7 - Appending to the contents of a text window

Because the selection area was reduced to zero width by the previous command, this second command will effectively insert the output immediately after the previous text, as shown in *Fig 3.7*. Any number of successive output calls can be made to the edit control of a text window, until such time as the window fills up (remember the approximately 64kb limit). A simple way to write arbitrary terms to a given text window's edit control is shown below:

```
write_to_text_window( Window, Term ) :-
        write( Term ) ~> String,
        wedtsel( (Window,1), 64000, 64000 ),
        wedttxt( (Window,1), String ).
```

This example uses the output redirection predicate, *~>/2*, to create a string from the given term, and then moves the text selection to the end of the edit control in the given text window, before writing the given string at this location. It does not attempt to check the amount of space left in the edit control, but it would be simple to add this test and to clear out the first few kilobytes from time to time.

Copying Text between Windows

As a practical example of the use of strings, we will now copy the whole contents of the console window into our text window, "fred". Although the console window is generally used for standard Prolog input and output, it can also be used in conjunction with the text window predicates. The console window has no name as such, and is referred to as the integer "1" in predicates such as *wfocus/1*. It too has a single edit control, whose ID is again 1. Type the following command into the console window (onto a single line):

```
?- wedtsel((1,1),0,64000), wedttxt((1,1),Data),
wedtsel((fred,1),0,64000), wedttxt((fred,1),Data).       <enter>
```

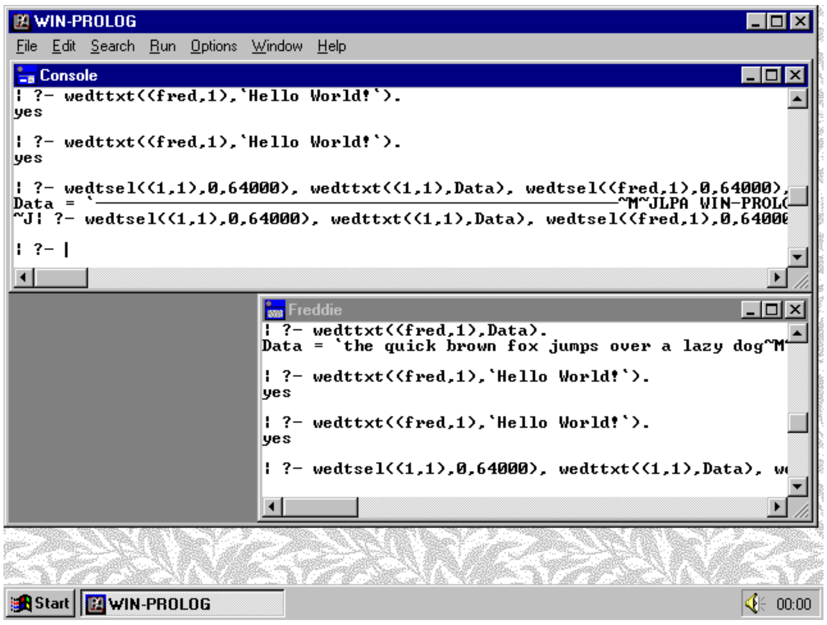
Notice how the console window scrolled sideways to make room for your command. Up to around 1024 characters can be typed on a single line if you want; alternatively, you can use the tricks described in *Chapter 2* to enter complex commands over several lines. When you press <*enter*>, there will be a brief flurry of screen activity, after which the original contents of the console window will be displayed in "fred", as shown in *Fig 3.8*. Notice how the output binding to "Data" has wrapped around: this is because its length exceeds the 1024 or so character logical width of the console window.

Closing a Text window

Text windows are closed with the *wclose/1* predicate. Closing a window destroys it, removing it from the screen and freeing up its memory resources. When a window is closed, its contents are lost: if you want to save the contents, simply use *wedtsel/3* and *wedttxt/2* to extract the text before closing the window. The text could then be saved in a Prolog clause, or written to a file. Type the command:

```
?- wclose(fred).                                          <enter>
```

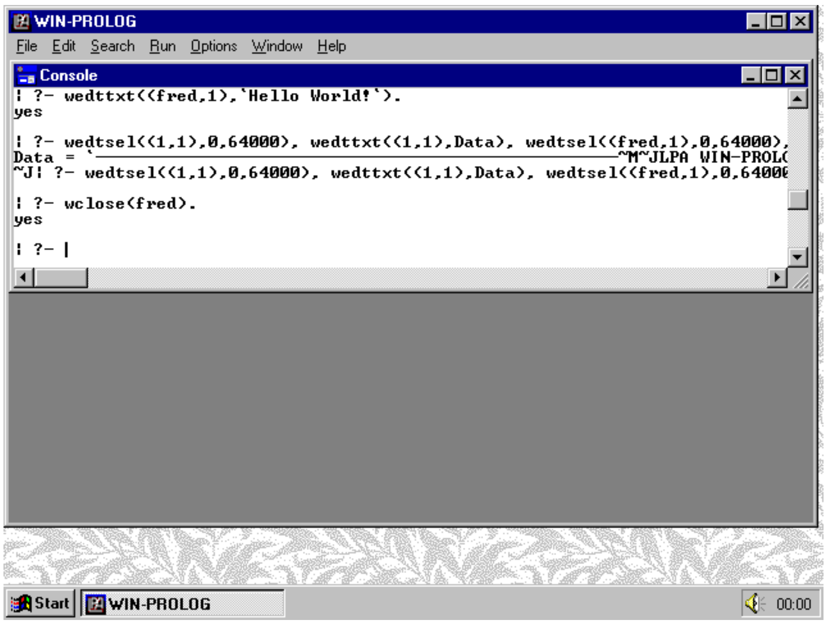
The window "fred" will disappear from the screen, leaving just the console window in view, as shown in *Fig 3.9*.



```
¦ ?- wedttxt((fred,1),`Hello World!`).
yes

¦ ?- wedttxt((fred,1),`Hello World!`).
yes

¦ ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), wedtsel((fred,1),0,64000),
Data = `───────────────────────────────────────────────────────~M~JLPA WIN-PROL
~J¦ ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), wedtsel((fred,1),0,6400

¦ ?- |
```

```
¦ ?- wedttxt((fred,1),Data).
Data = `the quick brown fox jumps over a lazy dog~M

¦ ?- wedttxt((fred,1),`Hello World!`).
yes

¦ ?- wedttxt((fred,1),`Hello World!`).
yes

¦ ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), w
```

Fig 3.8 - The console window copied to a text window



```
¦ ?- wedttxt((fred,1),`Hello World!`).
yes

¦ ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), wedtsel((fred,1),0,64000),
Data = `────────────────────────────────────────────────────~M~JLPA WIN-PROL
~J¦ ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), wedtsel((fred,1),0,6400

¦ ?- wclose(fred).
yes

¦ ?- |
```

Fig 3.9 - The result of closing the text window

Rows, Columns and Offsets

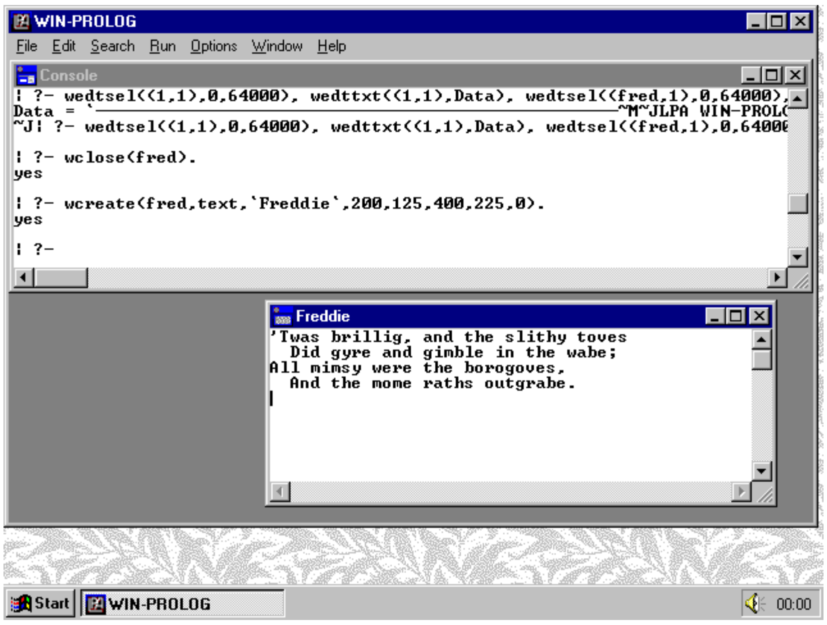
Unlike many glass teletype applications, where all text window-oriented input and output works in terms of column and row (cursor) positions, *WIN*-**PROLOG**'s window system is based on linear offsets. The first character in a window is numbered 0, the next character is 1, and so forth. There are times when it is important to be able to relate row and column positions to linear offsets. Examples include processing lines of input one at a time, implementing line-oriented edit features, or even porting code from the column/row based *DOS*-**PROLOG** system. Several predicates are provided to interface between the row/column and linear offset models, as we will see below. Create a new text window called "fred", using the command:

?- **wcreate(fred,text,`Freddie`,200,125,400,225,0).** *<enter>*

The window, "fred", should appear on the screen, as before, and focus will have switched to it. Next, resize it so that it does not overlap the console window, and type a few lines of text into it, so that it looks something like *Fig 3.10*. When you have done this, click back on the console window so that you can enter further commands.

Finding a Line

If you want to identify all the text in a single line of an edit control, you can use the *wedtlin/4* predicate. This takes a single offset as input, and returns the start and end offsets of the line containing that offset, not including any new line characters. The selection is not changed by this predicate: you must use *wedtsel/3* if you want to perform an operation on this line. Try the following example:



```
| ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), wedtsel((fred,1),0,64000),
Data = `————————————————————————————————————————————————~M~JLPA WIN-PROLC
~J| ?- wedtsel((1,1),0,64000), wedttxt((1,1),Data), wedtsel((fred,1),0,6400(

| ?- wclose(fred).
yes

| ?- wcreate(fred,text,`Freddie`,200,125,400,225,0).
yes

| ?-
```

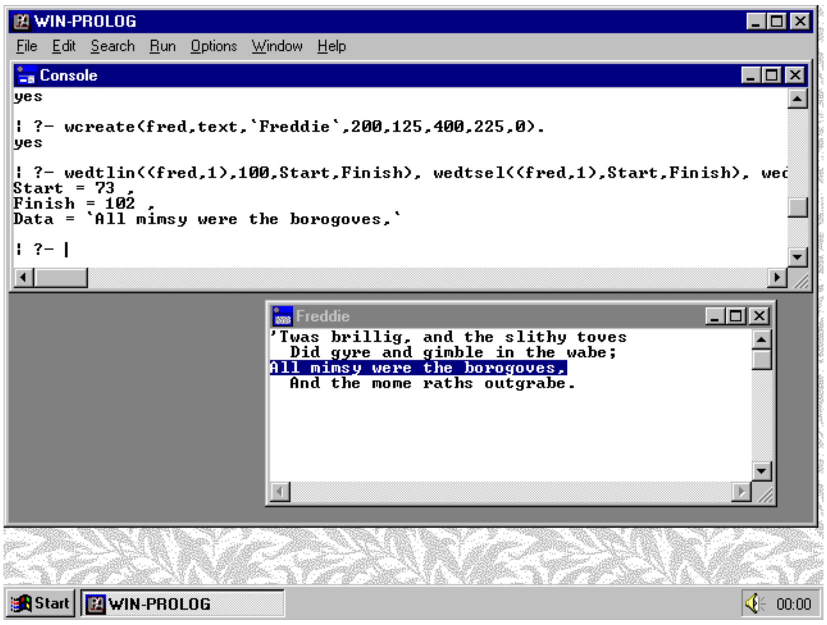
```
'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.
```

Fig 3.10 - The new text window with some sample input

?- **wedtlin((fred,1),100,Start,Finish), wedtsel((fred,1),Start,Finish), wedttxt((fred,1),Data).** *<enter>*

Depending upon the text that you entered, the result of this command will look something like *Fig 3.11*.

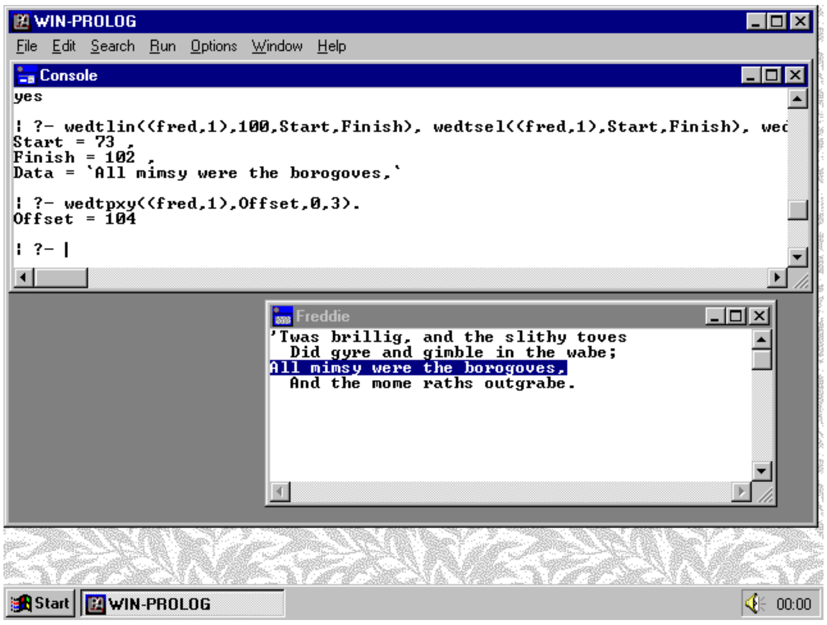
If you want to work from the row number, rather than the offset, you can use the *wedtpxy/4* predicate. This takes a given offset, and returns its column/row equivalent, or vice versa. The following command will return the offset of the



```
yes

| ?- wcreate(fred,text,`Freddie`,200,125,400,225,0).
yes

| ?- wedtlin((fred,1),100,Start,Finish), wedtsel((fred,1),Start,Finish), wed
Start = 73 ,
Finish = 102 ,
Data = `All mimsy were the borogoves,`

| ?- |
```

```
'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.
```

Fig 3.11 - Getting the start, finish and contents of a line



```
yes

| ?- wedtlin((fred,1),100,Start,Finish), wedtsel((fred,1),Start,Finish), wed
Start = 73 ,
Finish = 102 ,
Data = `All mimsy were the borogoves,`

| ?- wedtpxy((fred,1),Offset,0,3).
Offset = 104

| ?- |
```

```
'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.
```

Fig 3.12 - Converting a row/column address into an offset

start (column 0) of the third row:

?- **wedtpxy((fred,1),Offset,0,3)**. *<enter>*

The result of this command is shown in *Fig 3.12*. As with *wedtlin/4*, this predicate simply returns information: it does not affect the selection. If you were to combine this command with the previous example, you could return the text for a given numbered row.

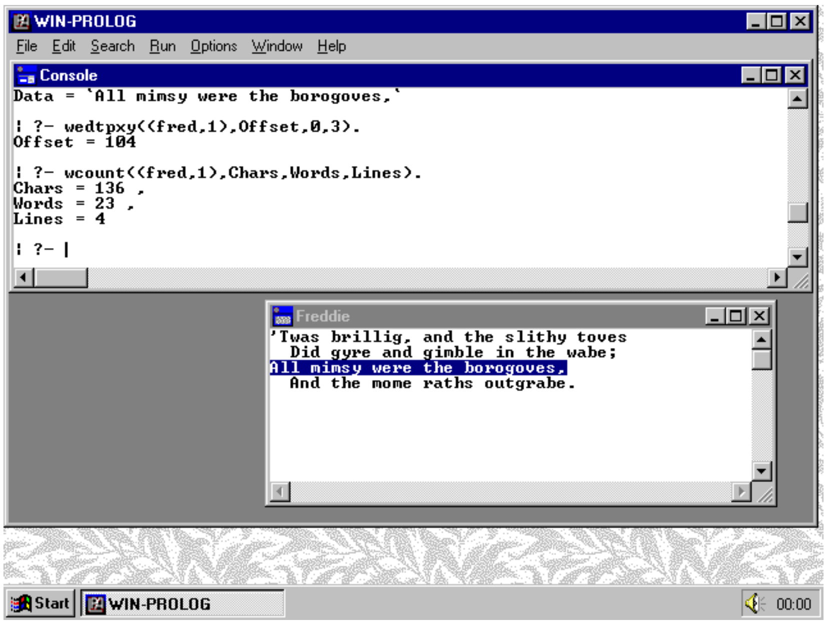
Counting Characters, Words and Lines

A text window can contain up to about 48kb (1kb is 1024 characters); once this limit is reached, further data can be stored only by removing text from other parts of the window. The *wedttxt/2* predicate simply fails if you attempt to add text which would overflow this limit, but you may prefer to know in advance whether a given text operation will work. The *wcount/4* predicate can be used to return the amount of text in a window. Type the command:

?- **wcount((fred,1),Chars,Words,Lines).** *<enter>*

This command returns the number of characters and lines in the current window, as shown in *Fig 3.13*. You could use these counts to implement automatic buffering to a disk file, or to trigger a warning message, before *wedttxt/2* got to the point where it failed through lack of space.

Notice that this predicate is named simply "wcount", rather than "wedtcnt" or suchlike: this is because it can be used on all types of window, and not just edit controls. Throughout WIN-PROLOG, any operations which are limited to just



```
Data = `All mimsy were the borogoves,`

| ?- wedtpxy((fred,1),Offset,0,3).
Offset = 104

| ?- wcount((fred,1),Chars,Words,Lines).
Chars = 136 ,
Words = 23 ,
Lines = 4

| ?- |
```

```
'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.
```

Fig 3.13 - Returning the character, word and line counts

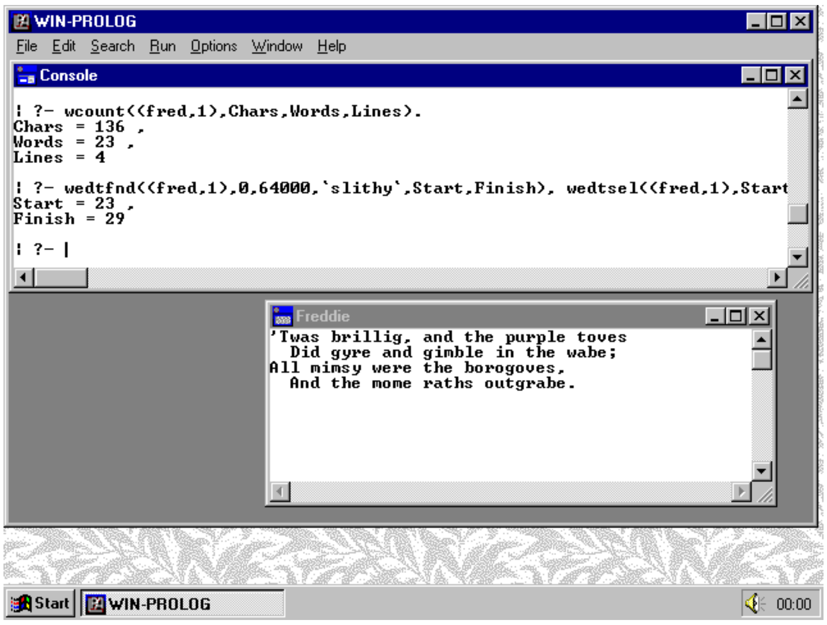
one type of control window are named "wxxxyyy", where "xxx" is an abbreviation of the control type and "yyy" is an abbreviation of the operation. Thus "wedttxt" is used to perform text operations on edit controls. Where a predicate is not limited to one type of control, but can be used in conjunction with any window, it is generally named "wzzz..", where "zzz.." is the full (non-abbreviated) name of the operation.

Finding Text

A very useful predicate, *wedtfnd/6*, allows you to search for a given text string in a given window. Once again, it does not move the selection box, but simply returns the start and end offsets of the found string. You must call *wedtsel/3* to select this area if you want to perform any operations upon it. Try the following command:

```
?- wedtfnd((fred,1),0,64000,`slithy`,Start,Finish),
wedtsel((fred,1),Start,Finish),  wedttxt((fred,1),`purple`).
                                                        <enter>
```

The call to *wedtfnd/6* searches for the string "slithy" from offsets 0 to 64000 (greater than the end) in "fred"'s edit control, returning the start and finish offsets of the first match. This area is then selected with *wedtsel/3*, and the text replaced with the word "purple", as shown in *Fig 3.14*: it would be just as easy to convert the word back to the original!



```
| ?- wcount((fred,1),Chars,Words,Lines).
Chars = 136 ,
Words = 23 ,
Lines = 4

| ?- wedtfnd((fred,1),0,64000,`slithy`,Start,Finish), wedtsel((fred,1),Start
Start = 23 ,
Finish = 29

| ?- |
```

```
'Twas brillig, and the purple toves
  Did gyre and gimble in the wabe;
All mimsy were the borogoves,
  And the mome raths outgrabe.
```

Fig 3.14 - Finding and replacing a string in a text window

The Window Dictionary

The names of all top level windows, including the text windows discussed here, are stored in an internal table which can be retrieved by the *wdict/1* predicate. The call:

?- **wdict(D).** *<enter>*

should return the list "D = [...,fred,...]", where other system-defined windows may or may not be present. The window dictionary is useful in programs which perform window management, and can be used in conjunction with *member/2* to perform operations on some or all currently defined windows.

Chapter 4 - Standard Dialogs

This chapter describes the standard dialogs of WIN-PROLOG, covering their appearance, parameters and uses. Standard dialogs provide facilities for message boxes, about boxes, file selection, and other general requirements.

Simple Modal Dialogs

In *Chapter 3*, we looked at some of the aspects of programming "text" windows. All of the functionality of these windows is programmable from Prolog, and these windows provide the necessary hooks for a wide range of text applications.

Certain requirements common to most Windows applications cannot be created using "text" windows, and a selection of standardised dialogs has been provided to give WIN-PROLOG easy access to some of these capabilities. These include the dialogs used to open, create and save files, as well as message boxes, about boxes, printer setup boxes, font boxes, status boxes, find boxes and change boxes. User-defined dialogs are discussed later in *Chapter 5*.

The Message Box

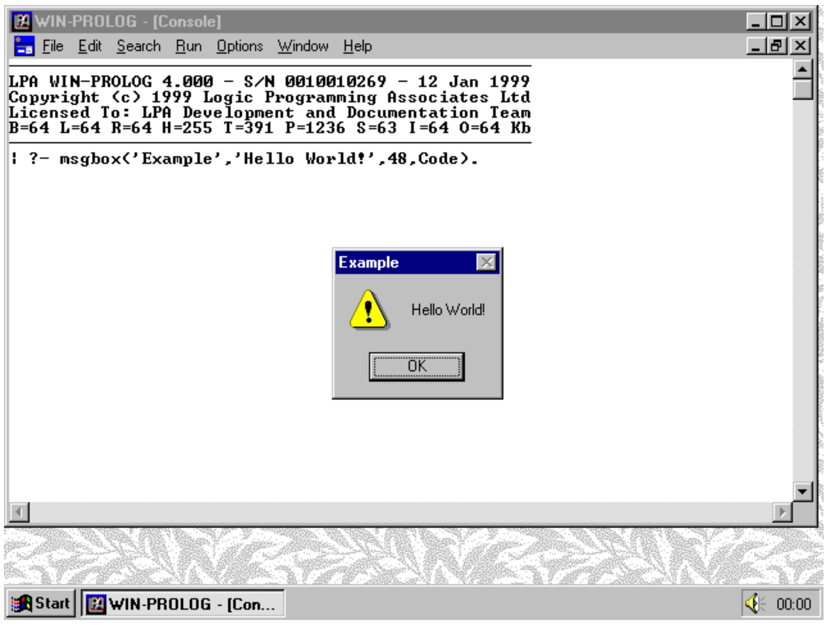
Windows provides many design guidelines, but very few truly standard user interface features. One notable exception to the above is the "message" box, used to alert the user and obtain confirmation about an action to be performed. Such message boxes can be created using the *msgbox/4* predicate. Let's try an example. If you are already in WIN-PROLOG, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start WIN-PROLOG up, and type the following command:

```
?- msgbox('Example','Hello World!',48,Code).            <enter>
```

In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in <*italics*>.

This command creates a standard message box with the word "Example" as its title, and the message "Hello World!" in its client area. The style parameter, "48", is responsible for the exclamation mark icon and "OK" button, as shown in *Fig 4.1*. The style is a 32-bit integer defined by Windows itself, and can generate a variety of icon and button combinations, as well as specifying different types of modality.

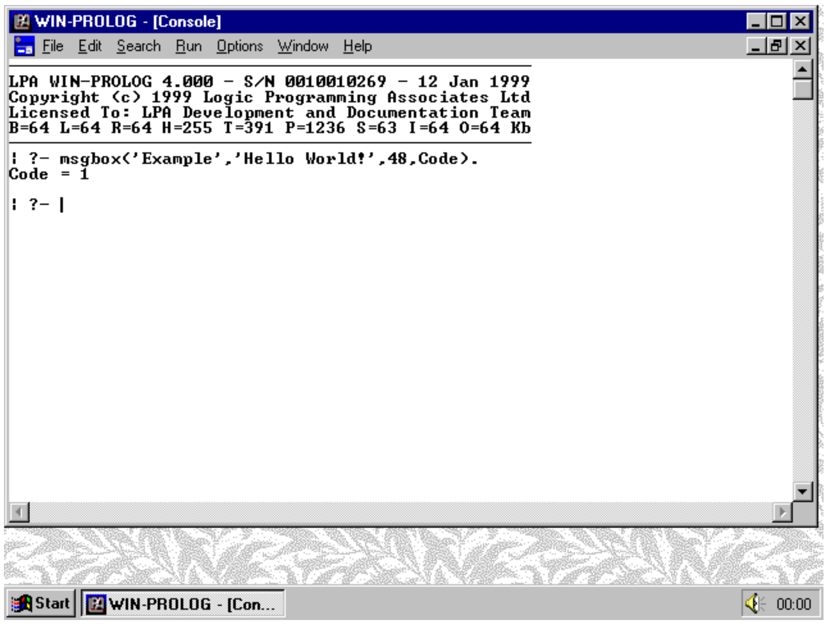
When the "OK" button is clicked with the mouse, or <*enter*> is pressed, the box vanishes, and returns the value "1", as shown in *Fig 4.2*, indicating that the "OK" option has been selected. Each of the several different buttons that can be included in a message box returns a unique number. For example, click the



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',48,Code).
```

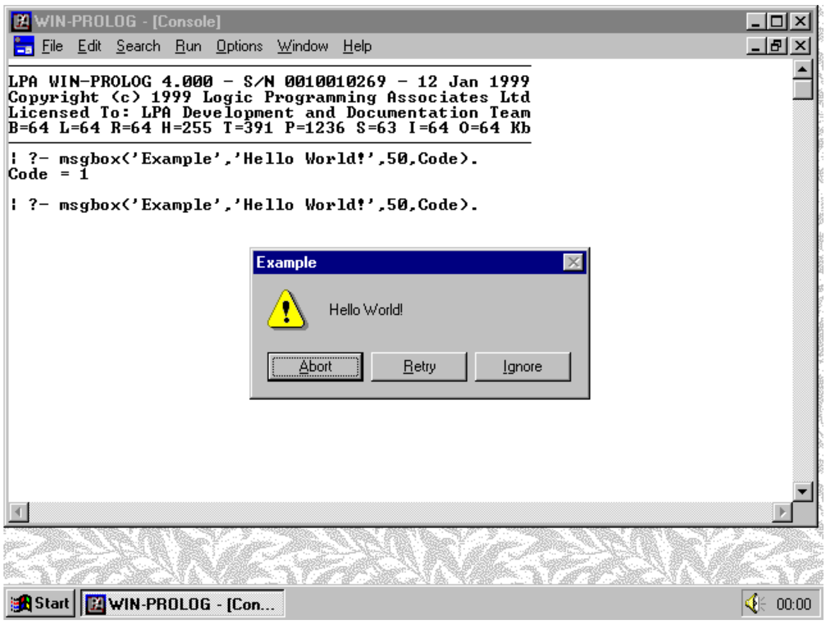
Fig 4.1 - A message box displayed by msgbox/4

mouse over "48" on the command you have just entered, change the value to "50", and then press <*enter*>. Your edited command line will be copied down to the bottom of the console window, and will result in the message box shown in *Fig 4.3*. Each of the three buttons has its own code: if you click on "Retry", for example, the returned code will be "4", as shown in *Fig 4.4*.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',48,Code).
Code = 1

| ?- |
```

Fig 4.2 - The code returned by the message box

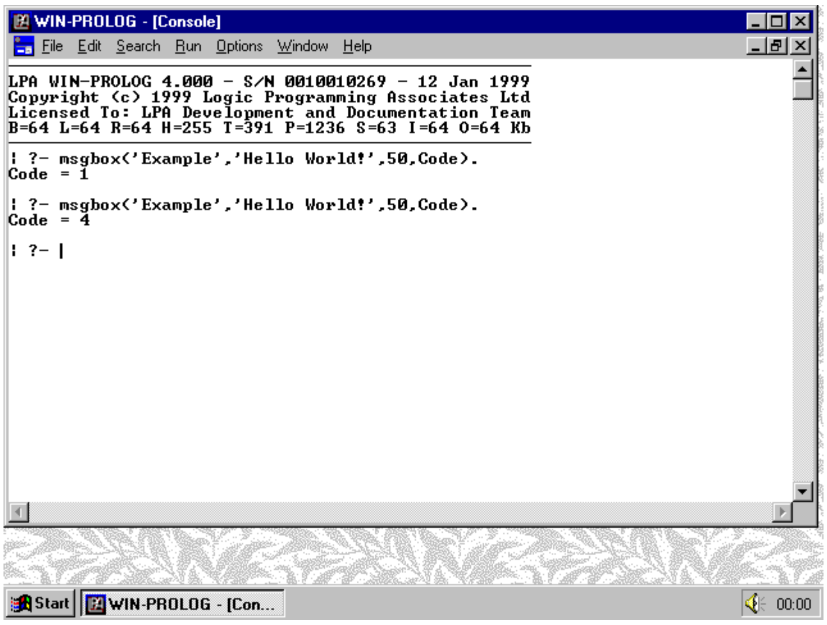


```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
```

Fig 4.3 - A message box with three buttons

Message Box Styles

As noted above, the message box style consists of a magic number defined by Microsoft Windows. If you have access to the Windows SDK, you will find the definitive list of styles listed in one of the header files for your version of Windows; otherwise, see the *msgbox/4* entry in the WIN-PROLOG *Technical Reference* for a listing of the message box styles supported by Windows.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- |
```

Fig 4.4 - The return code after selecting the retry button

When experimenting, please be aware of one potential problem: not all values for the message box values include buttons, and if you accidentally manage to create a message box without a button, there is no way to close the box and return to Prolog, other than by leaving Windows or using "End Task"!

The least significant 4 bits of the style dictate the button combination used by the message box. The next most significant 3 bits of the style govern which of the standard Windows icons is used, if any, in the message box. Remaining bits, provide other controls over the message box' behaviour, but should be left at zero for most purposes.

The About Box

Message boxes are among the only truly standard dialogs in Windows: most other dialogs are created by individual applications, which explains why no two Windows programs ever look the same or operate in quite the same manner. One dialog, the "about" box, which is used by an application to display information about itself, is perhaps more individualistic than any other. In WIN-PROLOG, the about box is really a special type of message box which can display text in a choice of two fonts and two styles, and with or without the WIN-PROLOG logo displayed in the background. In applications, you can replace this logo with one of your own design.

To show one style of about box, use the command:

```
?- abtbox('My about box','I did this one myself!',0).        <enter>
```

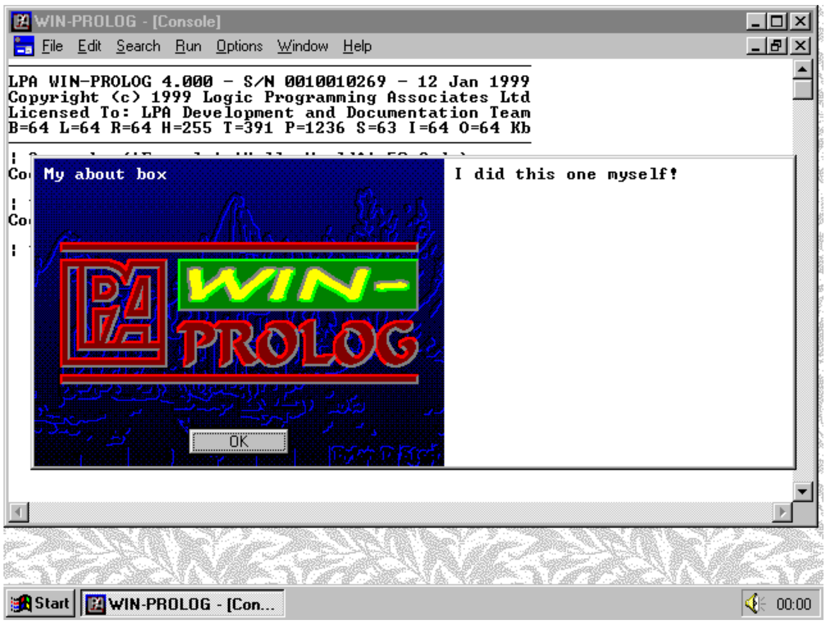
This will display the about box using the OEM character set and the WIN-PROLOG bitmap, as shown in *Fig 4.5*. Like *msgbox/4*, the parameters given to *abtbox/3* include the box title, message and style. Unlike the message box, however, the styles relate purely to the font which is used to display the text in the about box.

Because the about box always has just one button, "OK", it does not return a button code: instead, *abtbox/3* succeeds if "OK" is clicked with the mouse or *<enter>* is pressed, or fails if *<escape>* is pressed.

Please note that, unlike in the previous Win16 versions of WIN-PROLOG, the code to display the about box is contained in a separate file, PRO386W.DLL. If this file is not available at run time, the *abtbox/3* predicate simply displays a standard Windows message box.

The Open and Save Boxes

A pair of general purpose "common" dialogs, *opnbox/5* and *savbox/5*, provides WIN-PROLOG with a semi-standardised interface for selecting files and directories, and are used to implement commands such as "open", "save" and "save as". Note



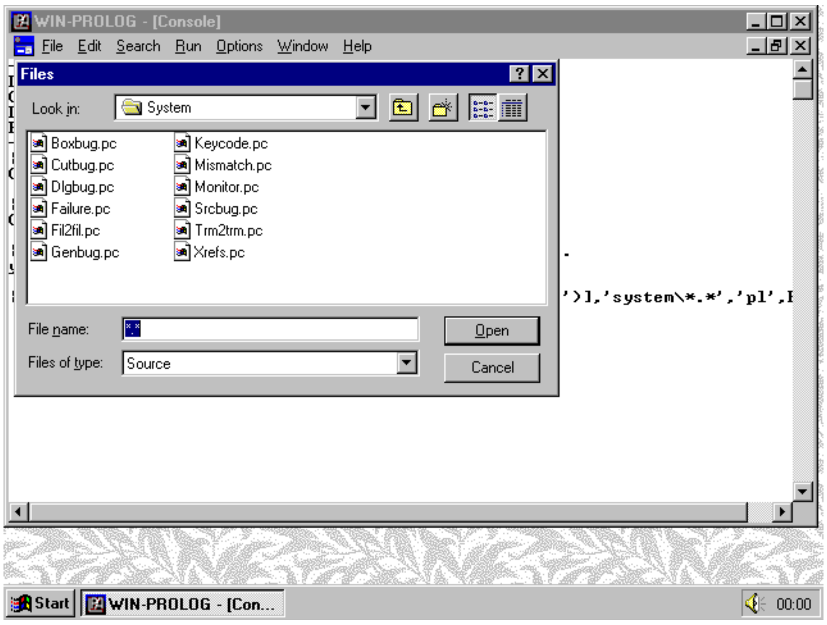
```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
```

```
My about box
I did this one myself!
```

Fig 4.5 - An about box displayed by abtbox/3

that these dialogs replace the former *dirbox/4*, which is now provided in a source code library file for backward compatibility. These dialogs have one unexpected side-effect: they can change the current working directory even if you eventually cancel the dialog. For example, type the command:

```
?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',F).
                                                              <enter>
```



```
Look in: System

Boxbug.pc      Keycode.pc
Cutbug.pc      Mismatch.pc
Dlgbug.pc      Monitor.pc
Failure.pc     Srcbug.pc
Fil2fil.pc     Trm2trm.pc
Genbug.pc      Xrefs.pc

File name:     *.*
Files of type: Source
```

Fig 4.6 - An open box displayed by opnbox/5

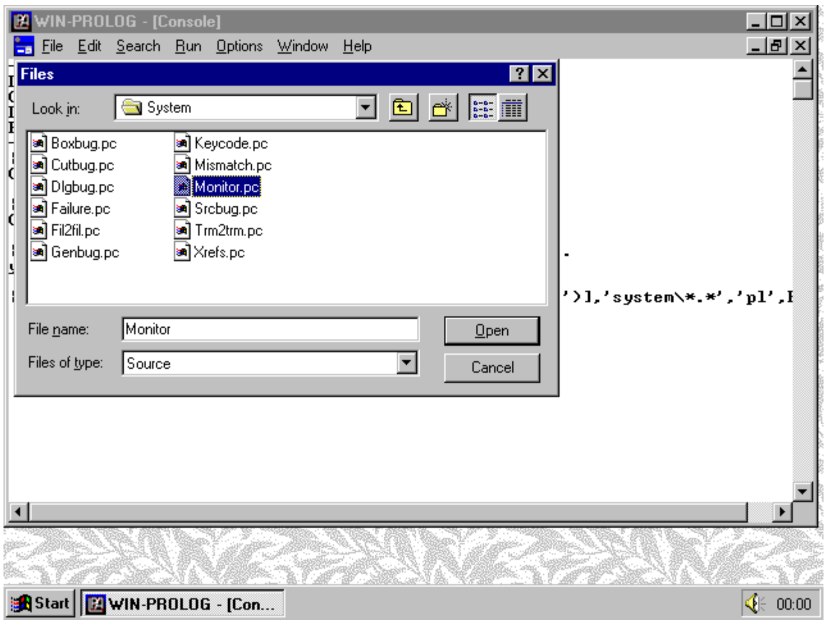


Fig 4.7 - Selecting files from the open box file list

A file open directory box will be displayed, with a list of ".PL" files from your current directory, as shown in *Fig 4.6*. You can use the mouse or the <*tab*> key to move between the fields in the dialog, and the mouse or space bar to select items from either of the list boxes. As you move the selection cursor down the "File Name" list, names are copied into the edit field, as shown in *Fig 4.7*. You can type in file names, paths, or wildcard patterns.

If you click "OK" or press <*enter*> while a wildcard pattern is in the edit box, a new selection is shown in the list box. For example, use the mouse or <*tab*> key to return to the edit box, and type in "..\library*.*", and then click "OK"

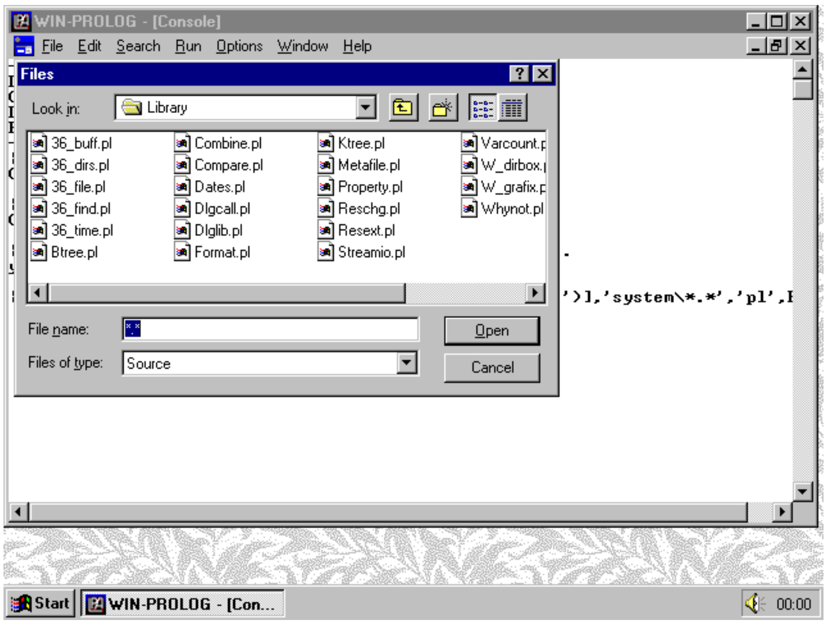
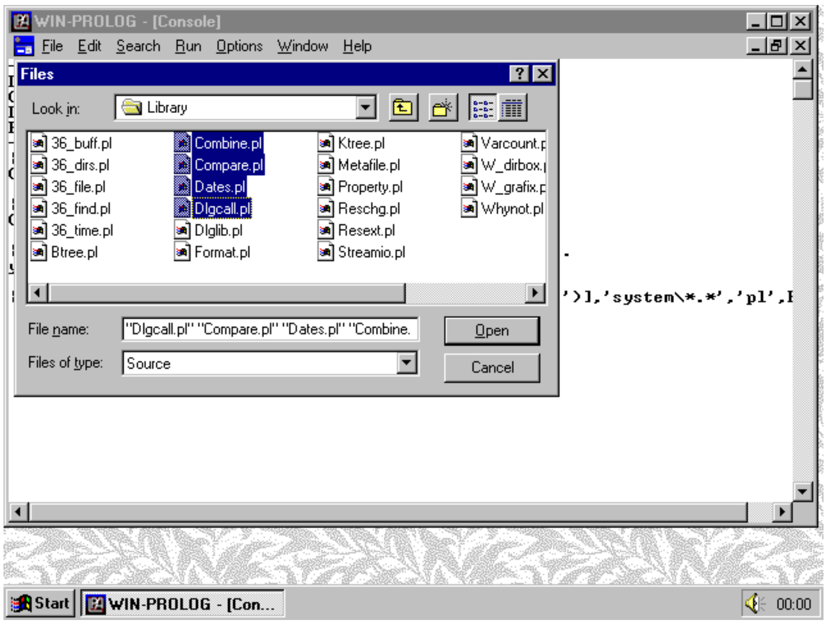


Fig 4.8 - Selecting files with a wildcard file specification



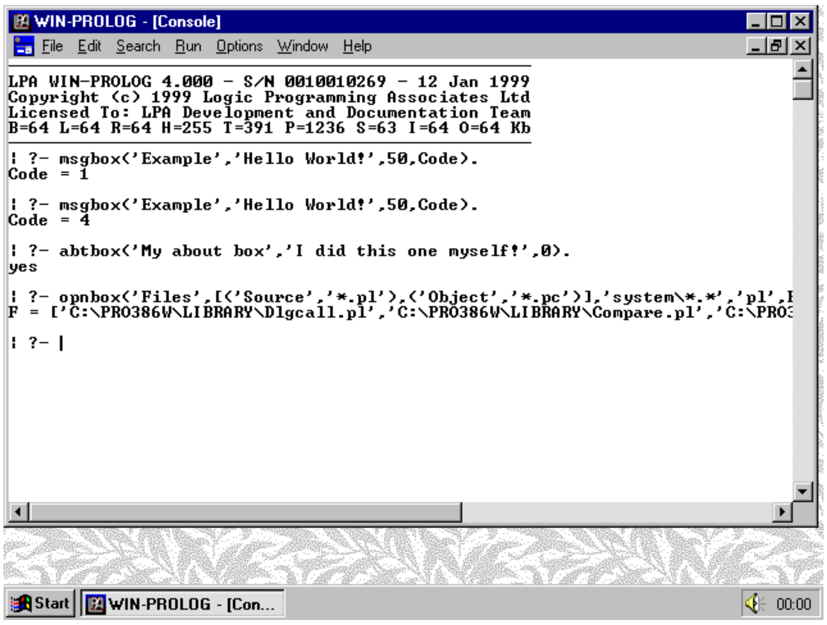
File name: "Dlgcall.pl" "Compare.pl" "Dates.pl" "Combine.

Files of type: Source

Fig 4.9 - Choosing some files from the open box file list

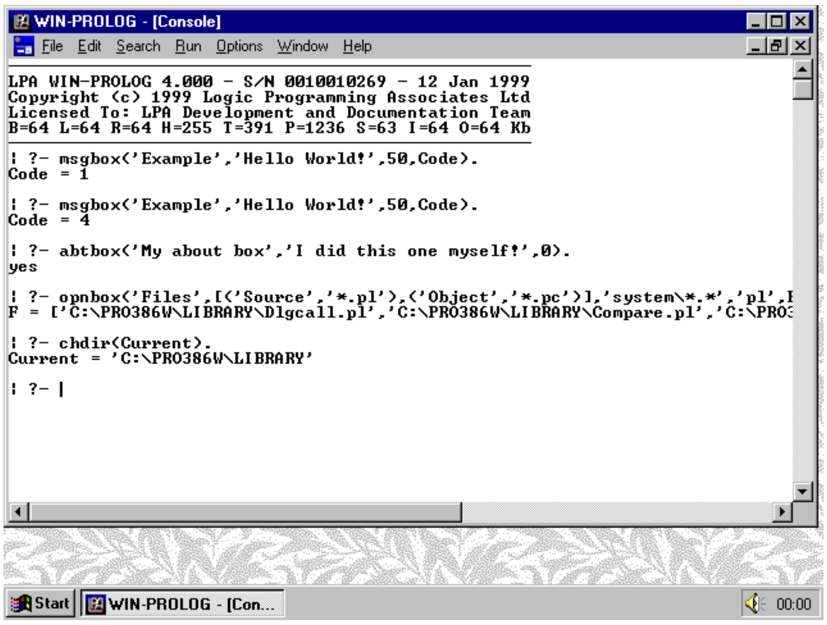
or press <*enter*>. The files menu will now contain a list of library files, as shown in *Fig 4.8*. Use the mouse or <*tab*> and cursor keys to select some files, as shown in *Fig 4.9*, and then click "OK" or press <*enter*>. The full path names of all the selected files will be returned as a list of atoms, as shown in *Fig 4.10*.

The *savbox/5* predicate is virtually identical to opnbox/5, except that it can only return a single file name, and it includes logic to prompt the user in the event of an existing file name being chosen. Any paths you select during either dialog will change your directory, even if you eventually cancel the dialog. To prove this, type the command:



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',F
F = ['C:\PRO386W\LIBRARY\Dlgcall.pl','C:\PRO386W\LIBRARY\Compare.pl','C:\PRO3

| ?- |
```

Fig 4.10 - Returning files name from the open box



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',F
F = ['C:\PRO386W\LIBRARY\Dlgcall.pl','C:\PRO386W\LIBRARY\Compare.pl','C:\PRO3

| ?- chdir(Current).
Current = 'C:\PRO386W\LIBRARY'

| ?- |
```

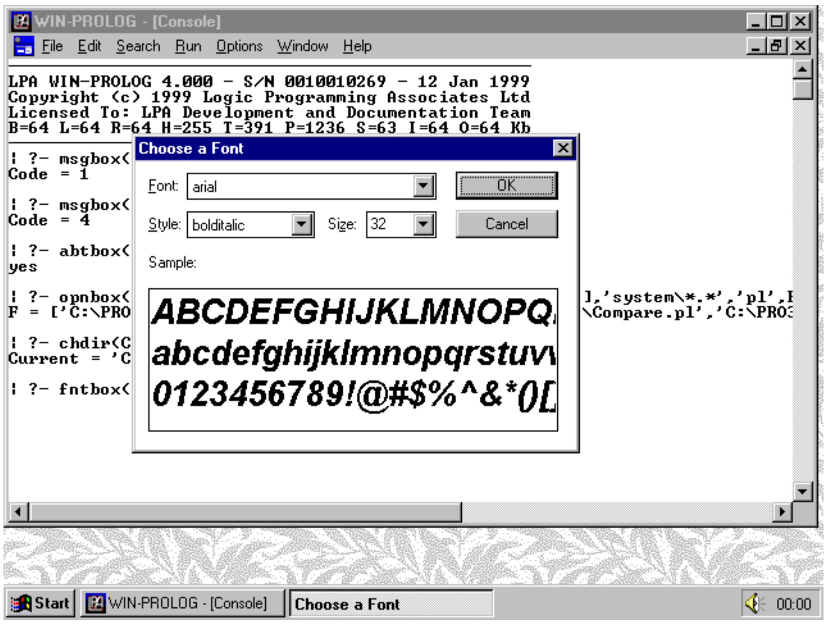
Fig 4.11 - Confirming that the directory has changed

?- **chdir(Current).** *<enter>*

This returns the name of the current working directory, as shown in *Fig 4.11*.

The Font Box

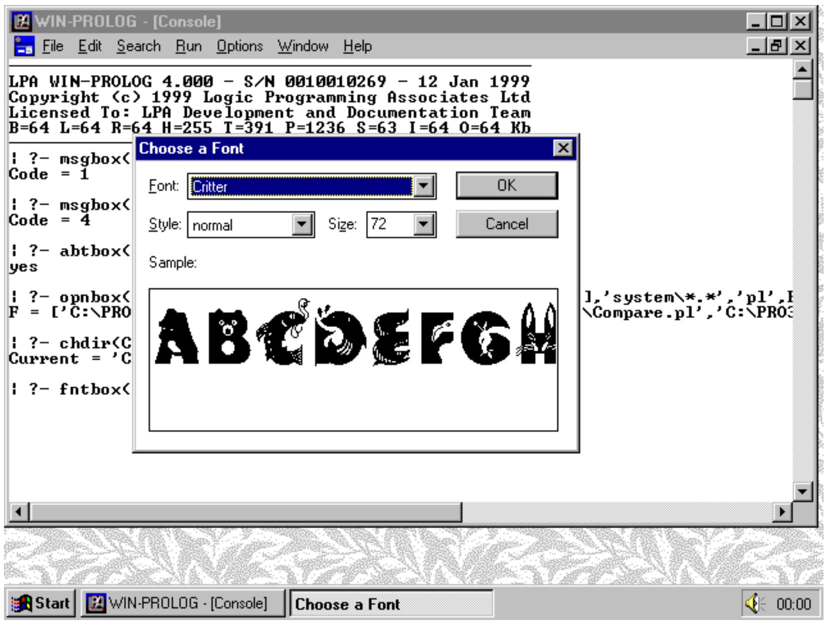
The *fntbox/3* predicate provides **WIN-PROLOG** with a simple method for selecting from the many fonts supported by Windows. It takes a title and initial font description as input, displays some sample text and then allows the user to select other fonts, returning the final selection. For example, type the command:



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox(
Code = 1

| ?- msgbox(
Code = 4

| ?- abtbox(
yes

| ?- opnbox(                                  ],'system\*.*','pl',F
F = ['C:\PRO                                 \Compare.pl','C:\PRO3

| ?- chdir(C
Current = 'C

| ?- fntbox(
```

Font: arial
Style: bolditalic
Size: 32

Fig 4.12 - A font box displayed by fntbox/3

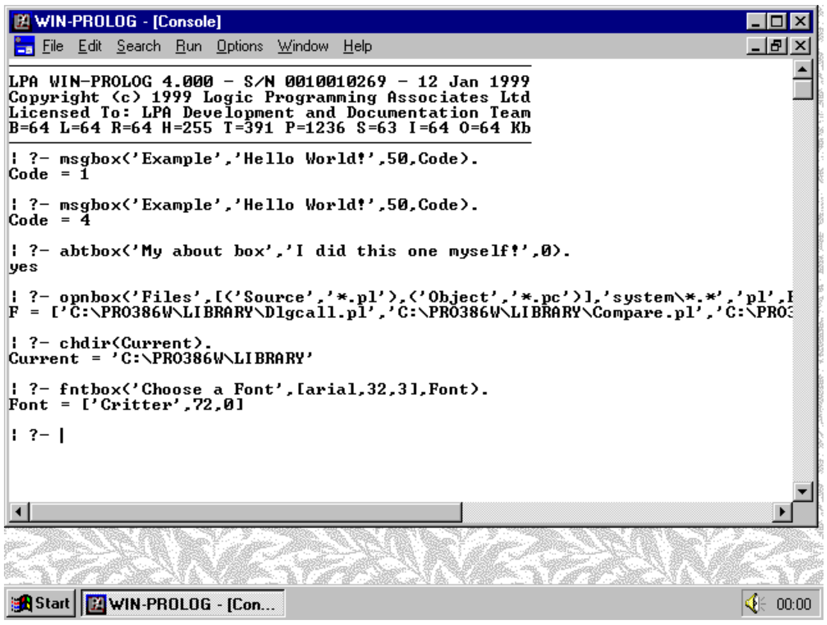


```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
```

Fig 4.13 - Choosing other font names in the font dialog

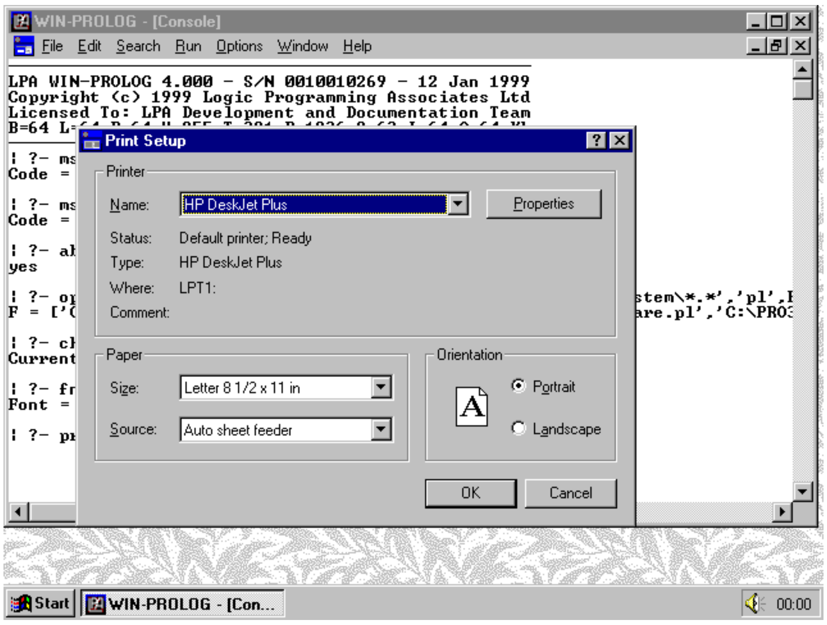
?- **fntbox('Choose a Font',[arial,32,3],Font).** *<enter>*

A font box will be displayed as shown in *Fig 4.12*. Select other sizes, styles and font names from the list, as shown in *Fig 4.13*, and then press "OK", at which point your final selection will be returned as a list, as shown in *Fig 4.14*. You will find out more about typefaces and fonts in *Chapter 7*.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',F
F = ['C:\PRO386W\LIBRARY\Dlgcall.pl','C:\PRO386W\LIBRARY\Compare.pl','C:\PRO3

| ?- chdir(Current).
Current = 'C:\PRO386W\LIBRARY'

| ?- fntbox('Choose a Font',[arial,32,3],Font).
Font = ['Critter',72,0]

| ?- |
```

Fig 4.14 - The chosen font name, size and style returned



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
```

Printer
Name: HP DeskJet Plus
Status: Default printer; Ready
Type: HP DeskJet Plus
Where: LPT1:
Comment:

Paper
Size: Letter 8 1/2 x 11 in
Source: Auto sheet feeder

Orientation
[✓ Portrait] Landscape

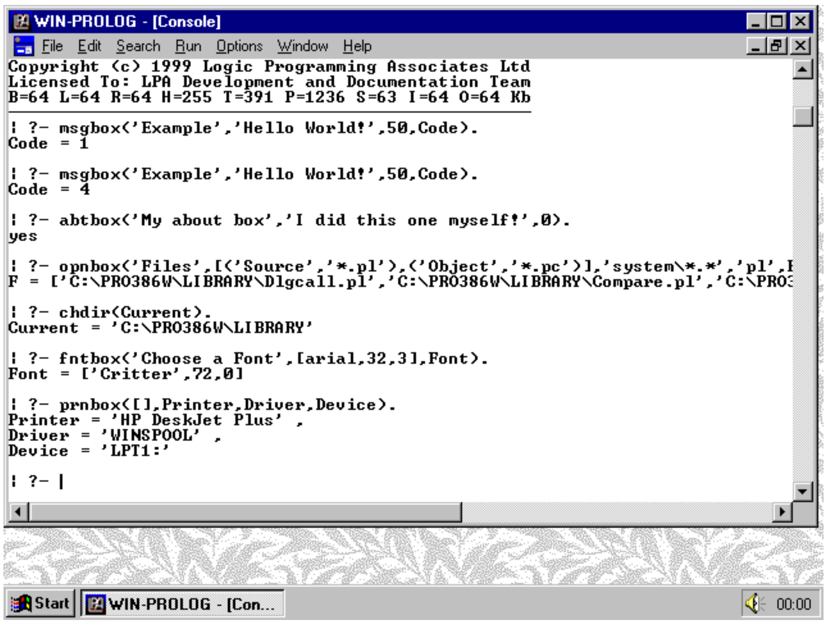
Fig 4.15 - The print setup dialog displayed by prnbox/4

The Print and Print Setup Boxes

The *prnbox/4* predicate gives **WIN-PROLOG** access to two related dialogs, one for initialisating the printer device context at the start of a print job, and the other for performing setup operations on the printer. The former dialog includes a "setup" option that links to the latter dialog. For example, type the command:

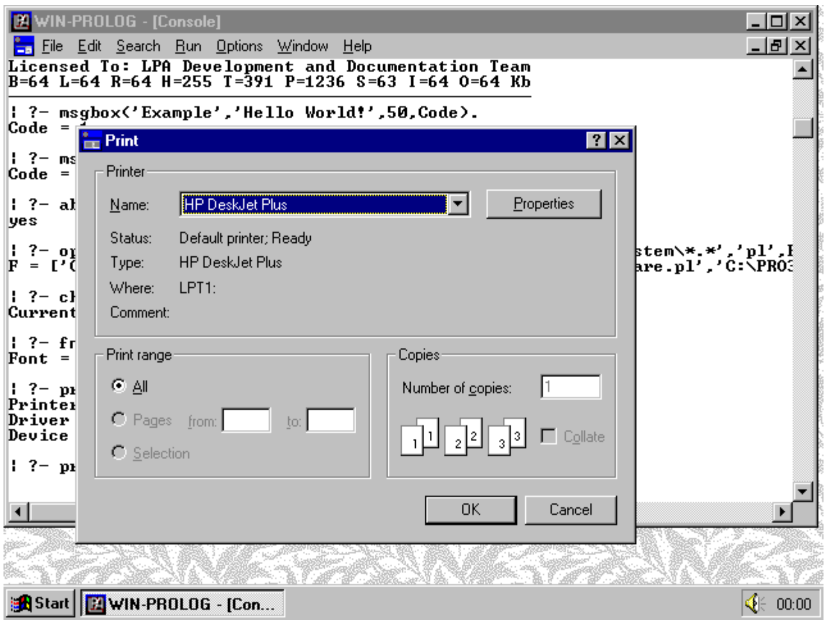
?- **prnbox([],Printer,Driver,Device).** *<enter>*

This will display the print setup dialog, as shown in *Fig 4.15*. If you simply click



```
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',F
F = ['C:\PRO386W\LIBRARY\Dlgcall.pl','C:\PRO386W\LIBRARY\Compare.pl','C:\PRO3

| ?- chdir(Current).
Current = 'C:\PRO386W\LIBRARY'

| ?- fntbox('Choose a Font',[arial,32,3],Font).
Font = ['Critter',72,0]

| ?- prnbox([],Printer,Driver,Device).
Printer = 'HP DeskJet Plus' ,
Driver = 'WINSPOOL' ,
Device = 'LPT1:'

| ?- |
```

Fig 4.16 - Printer information returned by prnbox/4



```
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- msgbox('Example','Hello World!',50,Code).
Code = 1
```

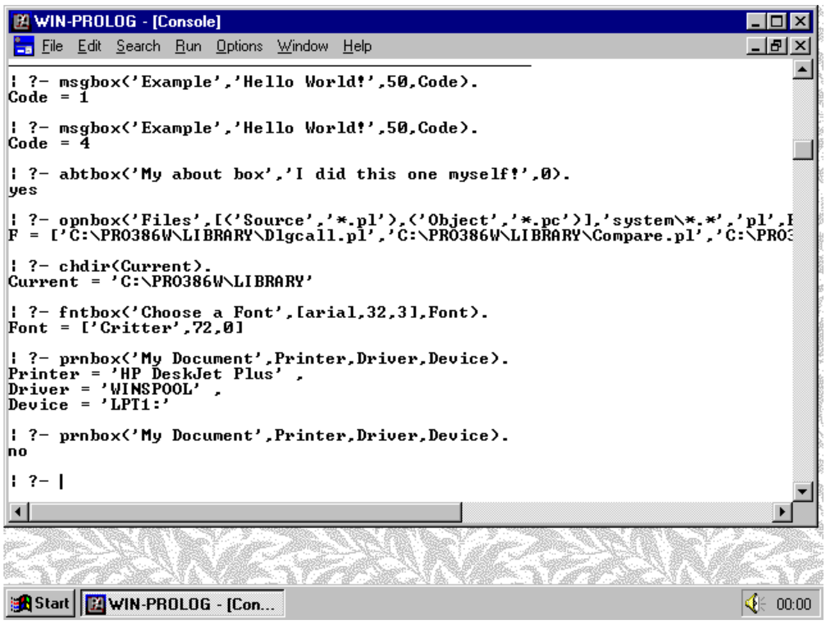
Name: HP DeskJet Plus
Status: Default printer; Ready
Type: HP DeskJet Plus
Where: LPT1:
Comment:
Number of copies: 1

Fig 4.17 - The print dialog displayed by prnbox/4

on "OK", the current printer and driver information will be returned, as shown in *Fig 4.16*. Note that to initiate a print job, you simply supply a document name (an atom) as the first argument to *prnbox/4*, rather than an empty list as above. For example, edit the previous command to include a document name:

?- **prnbox('My Document',Printer,Driver,Device).** *<enter>*

This will show the print dialog, as shown in *Fig 4.17*. It is best to select "Cancel" right now, as shown in *Fig 4.18*, because otherwise you will commence a print job, and printing is not discussed in detail until *Chapter 12*.



```
| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',F
F = ['C:\PRO386W\LIBRARY\Dlgcall.pl','C:\PRO386W\LIBRARY\Compare.pl','C:\PRO3

| ?- chdir(Current).
Current = 'C:\PRO386W\LIBRARY'

| ?- fntbox('Choose a Font',[arial,32,3],Font).
Font = ['Critter',72,0]

| ?- prnbox('My Document',Printer,Driver,Device).
Printer = 'HP DeskJet Plus' ,
Driver = 'WINSPOOL' ,
Device = 'LPT1:'

| ?- prnbox('My Document',Printer,Driver,Device).
no

| ?- |
```

Fig 4.18 - Result of cancelling the print dialog

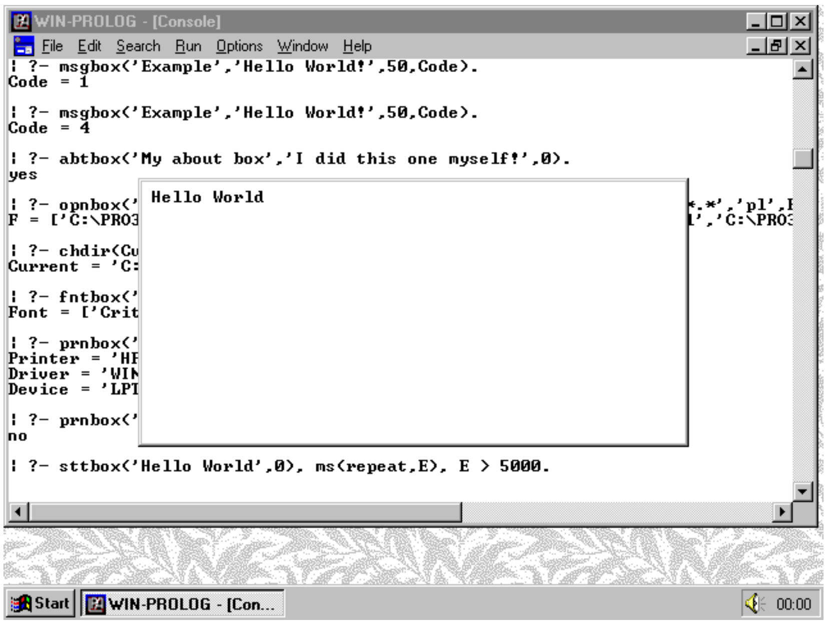
The Status Box

This is a dialog which differs from the others described in this chapter in that it is semi-modeless. While the status box is displayed, all **WIN-PROLOG**'s other windows and dialogs remain active (unless explicitly disabled), and **WIN-PROLOG** continues processing unhindered, but the status box is automatically destroyed whenever the system finishes processing a command and returns to the console prompt. The status box can be called any number of times in a computation: once on display, only its contents are updated on successive calls. The status box is typically used to display information about the progress of a long computation; for example, type the following command:

?- **sttbox('Hello World',0), ms(repeat,E), E > 5000.** *<enter>*

The reason for the call to *ms/2*, and checking the resulting time is greater than 5000ms, is that returning to the console prompt after completing a command automatically destroys the status box. During the 5-second delay caused by the *repeat/0* calls within *ms/2*, the status box will be displayed as shown in *Fig 4.19*. The value "0" requests a status box with the fixed "OEM" font and no **WIN-PROLOG** bitmap. As with the *abtbox/3* predicate, the style argument is simply the handle of the font to use to display text. A special "font handle" of 65535 is used to select the OEM font and display the **WIN-PROLOG** logo. While the status box is displayed, the underlying process can be interrupted by clicking on the main window and pressing *<ctrl-break>* as usual.

Please note that the code to display the status box is contained in a separate file, PRO386W.DLL. If this file is not available at run time, the *sttbox/2* predicate simply succeeds quietly without displaying anything.



```
| ?- msgbox('Example','Hello World!',50,Code).
Code = 1

| ?- msgbox('Example','Hello World!',50,Code).
Code = 4

| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('                                          *.*','pl',I
F = ['C:\PRO3                                          l','C:\PRO3

| ?- chdir(Cu
Current = 'C:

| ?- fntbox('
Font = ['Crit

| ?- prnbox('
Printer = 'HI
Driver = 'WIN
Device = 'LPT

| ?- prnbox('
no

| ?- sttbox('Hello World',0), ms(repeat,E), E > 5000.
```

```
Hello World
```

Fig 4.19 - A status box displayed by sttbox/2

The Busy (Hourglass) Cursor

Whenever a Windows program is busy, it typically disables input to its windows, and indicates this state by showing the cursor as an "hourglass" shape. You can set this cursor yourself, using the *busy/1* predicate. It takes a single argument, which is either "0" (not busy) or "1" (busy); this predicate can be used in conjunction with the status box, or by itself during shorter computations.

When the hourglass cursor is displayed, all mouse and keyboard input is disabled, preventing the user from performing any input functions. Like the status box, this state is automatically cleared by certain system events such as input from the console window, and once again, the underlying process can be interrupted by pressing <*ctrl-break*>.

Find and Change Boxes

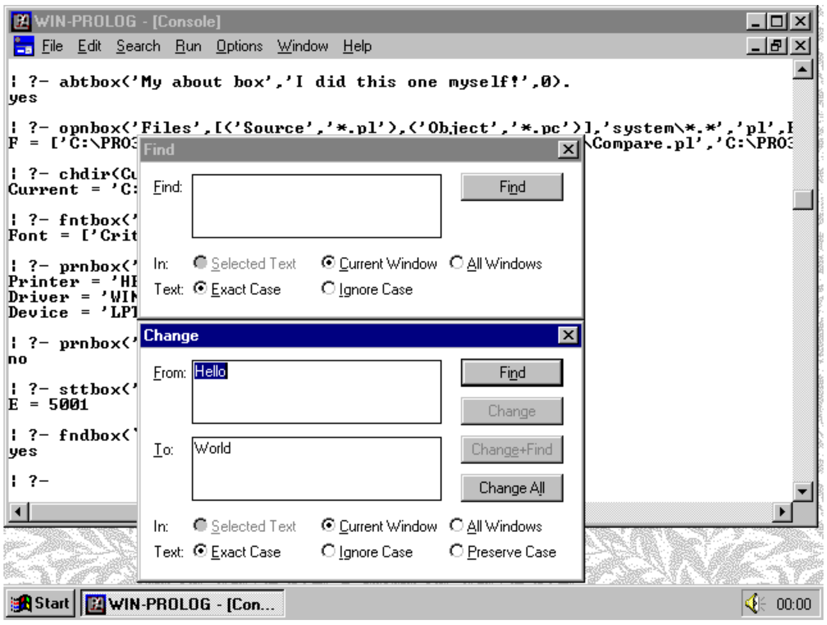
The two remaining dialogs are fundamentally different from the others described in this chapter in that they are truly modeless, and signal results to**WIN-PROLOG** using messages (see *Chapter 10* for a detailed discussion of messaging). Both are special, rather than general purpose, and are related to editing functions. One is used to return a find string (*fndbox/2*) and the other to return find/replace string pair (*chgbox/3*) for use in conjunction with these editing options. Because of their single purposes, these dialogs have fixed titles and prompts.

Because the dialogs are modeless, the predicates*fndbox/2* and*chgbox/3* return control immediately: their functions are to hide, display, or return information from the relevant boxes. Either one, neither or both of these boxes may be present on the screen at any one time, and the user may enter information or move between these boxes and other windows at will.

When one of the action buttons is clicked, or <*enter*> is pressed, the find or change box is disabled, preventing further interaction with the box, and a message is sent to Prolog. In response, the message handler should call the relevant find or change box predicate with variables to return the contents, perform the action requested, and then reenable or hide the box as demanded.

The messages returned by both boxes are dependent upon their operating mode: both can be used in a "find first" or "find next" state. The find box looks identical in both cases, but the change box has two buttons which are greyed out ("Change" and "Change+Find") in the "find first" state. Even if these dialogs are enabled in the "find next" state, certain user actions, such as clicking the mouse elsewhere on the screen, or changing the contents of the edit boxes or radio buttons, will revert the dialogs to the "find first" state. The only other difference between these states is that in one the "Find" button sends a "find first" message, and in the other it sends "find next". Type in the following command:

```
?- fndbox(``,1), chgbox(`Hello`,`World`,1).            <enter>
```



```
| ?- abtbox('My about box','I did this one myself!',0).
yes

| ?- opnbox('Files',[('Source','*.pl'),('Object','*.pc')],'system\*.*','pl',I
F = ['C:\PRO3                                    \Compare.pl','C:\PRO3

| ?- chdir(Cu
Current = 'C:

| ?- fntbox('
Font = ['Crit

| ?- prnbox('
Printer = 'HI
Driver = 'WIN
Device = 'LPT

| ?- prnbox('
no

| ?- sttbox('
E = 5001

| ?- fndbox('
yes

| ?-
```

Find:
In: Selected Text [✓ Current Window] All Windows
Text: [✓ Exact Case] Ignore Case

From: Hello
To: World
In: Selected Text [✓ Current Window] All Windows
Text: [✓ Exact Case] Ignore Case Preserve Case

Fig 4.20 - Dialogs displayed by fndbox/2 and chgbox/3

Note that the text parameters are all strings, not atoms: both these dialogs require strings. Normally you should use empty strings, because these leave the contents of the relevant edit control unchanged, which is desirable for modeless dialogs.

Notice that control returns to Prolog immediately, as evidenced by the fact that both boxes are shown on at once, and the "yes" and "?-" prompts have already appeared. Move the change box to reveal the find box beneath it, as shown in *Fig 4.20*. You will not be able to do anything particularly useful with these two dialogs until you have read *Chapter 10*, which discusses messaging.

Chapter 5 - User Windows and Dialogs

This chapter describes the "user" and "dialog" windows of WIN-PROLOG, covering their creation, manipulation and low level programming. These windows provide the framework for building complex and dynamic dialogs which go well beyond the capabilities of the standard dialogs.

User and Dialog Windows

In *Chapter 4*, we looked at a collection of standard dialogs, which provide easy to use functions for a number of common system events. Apart from being able to change text fields and (in some cases) the styles of these dialogs, while they are useful in certain circumstances, they are limited in their range of applications.

WIN-PROLOG provides two types of user-defined window in which any arbitrary dialog can be built. The first kind, "user" windows, are similar to "text" windows (see *Chapter 3*) in that they form part of the Multiple Document Interface ("MDI"). The second kind, "dialog" windows, occur outside the MDI, and exist on the desktop alongside WIN-PROLOG's "main" window and other Windows applications.

User and dialog windows share an important common feature which distinguishes them from text windows: when they are created, they contain no predefined controls (these are added subsequently).

Creating a User Window

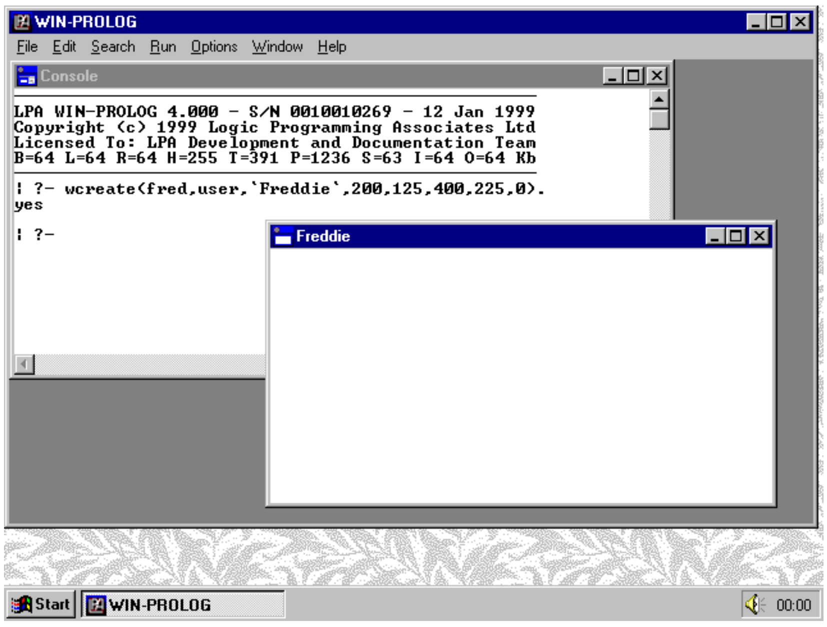
User windows are created with the *wcreate/8* predicate. Its arguments give the window its name, type, title, size and style. Let's try an example. If you are already in WIN-PROLOG, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start WIN-PROLOG up, and type the following command:

```
?- wcreate(fred,user,`Freddie`,200,125,400,225,0).        <enter>
```

In this, and all other examples, only type the characters in**bold** letters, and press the named key for anything bracketed in <*italics*>.

This example will create a window known to Prolog as the atom "fred", and will create it as a user window with the title "Freddie". The initial position will be at (200,125) relative to the main window, and its size will be (400*225), as shown in *Fig 5.1*. The style (0) is a dummy parameter here: in user windows, the style argument is ignored.

Note the use of backwards quotes (`...`) around the title, denoting a string rather



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

¦ ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

¦ ?-
```

Fig 5.1 - A user window created by wcreate/8

than an atom. Throughout the Windows predicates, strings are used for general text parameters. If you are not familiar with strings, and their relationships with atoms and char lists, you should refer to *Appendix B* for further information.

Lack of Control

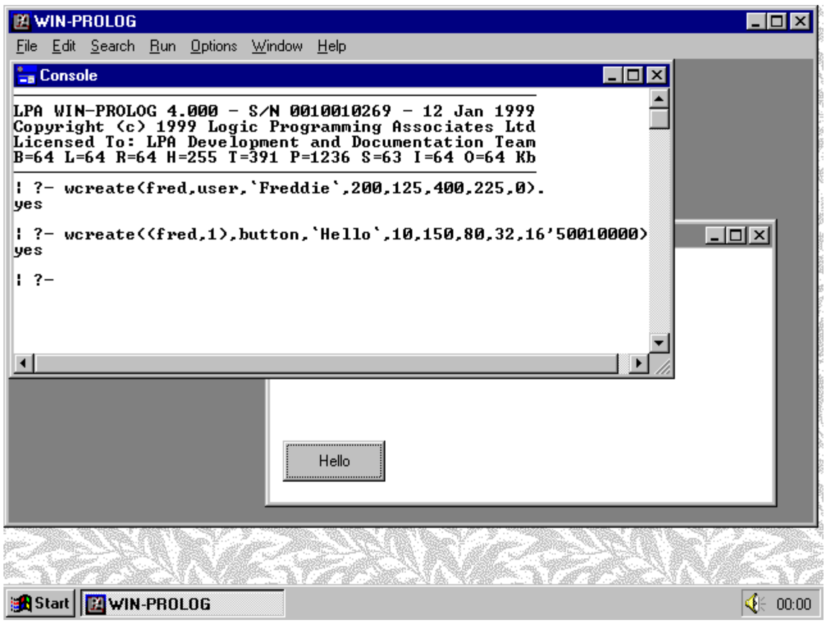
Because user windows do not contain any predefined controls, you will not be able to type anything into your new window. Instead, click back on the console window, and type the following command (**bold** letters only):

?- **wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000).**
<enter>

This will create a single button in your user window, with an ID of "1", and the label "Hello", at position (10,10) and of size (80*32), as shown in *Fig 5.2*. The style parameter has been given in hexadecimal notation (16'....), because this makes it easier to compute complex styles. Do not worry about the meaning of this number for now: styles are explained in greater detail in *Appendix D*; controls will be examined in detail in *Chapter 6*.

Creating a Dialog Window

As you will probably have guessed by now, dialog windows are also created with the *wcreate/8* predicate. Two main differences exist between the creation of MDI (text and user) and dialog windows: firstly, the given position is absolute, and not relative to the main window, and secondly, the style parameter is significant. As an example, type the following command:

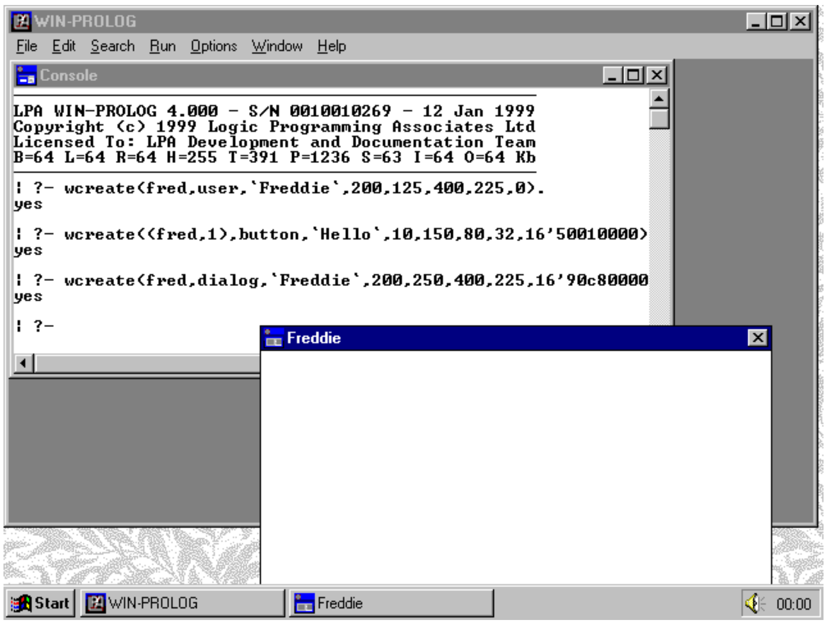


```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?-
```

Fig 5.2 - A button added to a user window by wcreate/8

?- **wcreate(fred,dialog,`Freddie`,200,250,400,225,16'90c80000).**
<enter>

Your existing user window called "fred" will be deleted, and replaced by a top level dialog window called "fred", as shown in *Fig 5.3*. Again, the style has been shown in hexadecimal notation, and once again, do not worry about its meaning for now.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'90c80000
yes

| ?-
```

Fig 5.3 - A dialog window created by wcreate/8

Just as before, the new (dialog) window will have gained focus, but since it has no controls, any attempt to type on the keyboard will be unsuccessful. Normally you would create a dialog "invisible", add controls to it, and only display and set focus to the finished result when ready to use it (see below for an example of how to do this): in this mode of use, the inability to perform keyboard input will not arise. For now, click back on the console, and add a button by typing the following command:

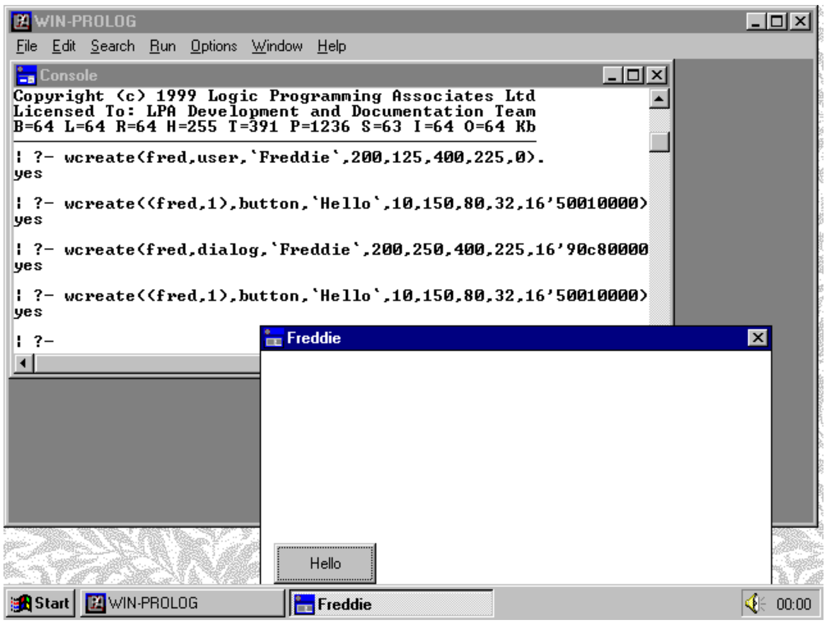
```
?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000).
                                                        <enter>
```

Just as before, this will create a single button labelled "Hello", as shown in *Fig 5.4*. As you can see, user and dialog windows are superficially very similar.

Differences between User and Dialog Windows

At this point, we should say a few words about user and dialog windows. As was briefly outlined above, the main difference between user and dialog windows is that the former exist in the **WIN-PROLOG** MDI environment, vying for space with text windows and the console within the main window, while the latter exist on the top level desktop, and are for most intents and purposes independent of **WIN-PROLOG**. Another difference between user and dialog windows is that the former are fixed in style, while the latter is not.

A more subtle, but very important difference between these two types of window is in the processing of keyboard input. Certain keys, such as *<tab>* and *<return>* behave differently in controls contained in user and dialog windows. For example, if you were to create a pair of edit controls in a user window, and



```
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'90c80000
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?-
```

Fig 5.4 - A button added to a dialog window by wcreate/8

then clicked on one of them, typing <*tab*> would insert a tab character into the edit buffer of whichever control you were typing into; in a dialog window, the <*tab*> key would cause you to switch between controls. Similarly, <*return*>, if pressed in a control contained in a user window, inserts a new line character, while in a dialog it causes the default push button (if present) to be activated.

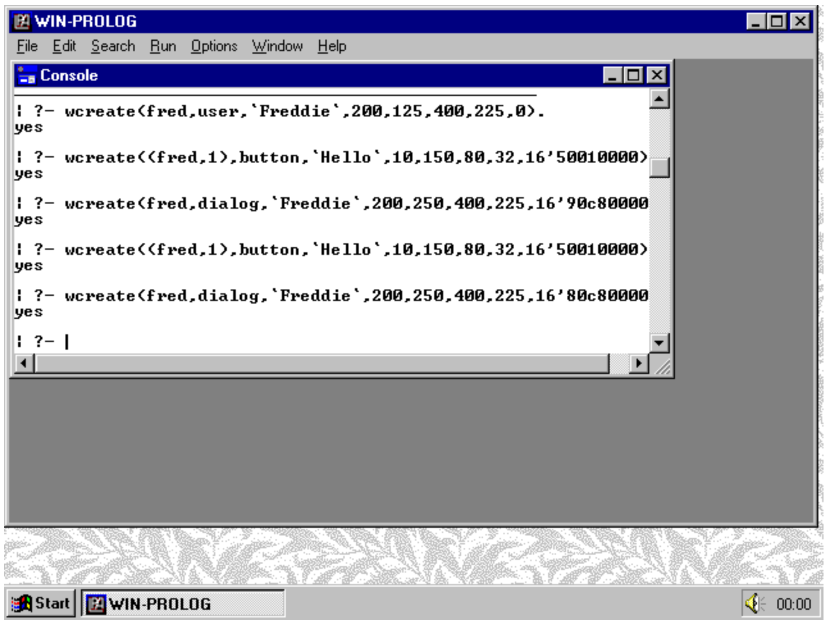
In general, normal dialog behaviour is exhibited only by dialog windows. You can put any number of controls, of any kind, into both user and dialog windows, but in user windows you might need to use the mouse to move between controls rather than the normal keyboard shortcuts. User windows are actually intended as placeholders for graphics and other output features, so in general you should use dialog windows when defining dialogs.

Creating Invisible Dialogs

Normally, you would not want users to see the individual steps used in building a dialog, but would prefer them just to see the finished result. Apart from some strange screen behaviour which can occur when you create a dialog which is initially visible (see above), it simply looks messy to have a dialog appear, and then add its controls visibly, one by one. The simple trick is to create an invisible dialog, fill it with controls, and then show it only when you want to use it. To see how to do this, type the following command:

```
?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'80c80000).
                                                        <enter>
```

Note that the hexadecimal parameter has changed from the previous example (16'8... rather than 16'9...): this is important, as the missing bit is the visibility



```
| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'90c80000
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'80c80000
yes

| ?- |
```

Fig 5.5 - An invisible dialog created by wcreate/8

flag. Because you are reusing the name "fred", the existing dialog will be destroyed before the new one is created. Since the new one is invisible, you will simply be left in the console window, as shown in *Fig 5.5*. Now create the usual button in the dialog:

```
?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000).
                                                        <enter>
```

The hexadecimal parameter here (16'5...) includes the visibility flag, but since "fred" itself is invisible, you won't see anything new on the display. Visibility is always with respect to a parent window, so a button could be invisible even if its parent window were visible, but not vice versa. Now "turn on" the dialog with the following command:

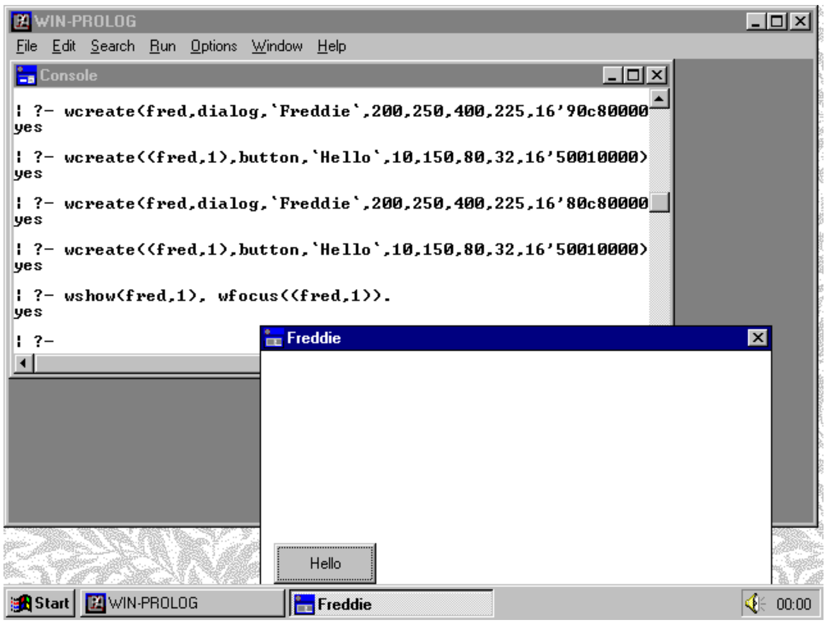
```
?- wshow(fred,1), wfocus((fred,1)).                     <enter>
```

This causes the dialog "fred" to be shown, and then sets focus to its button, as shown in *Fig 5.6*.

Note the double parentheses on the second call: the focus predicate, *wfocus/1*, has only one argument, the name of the window which is to gain focus. The show predicate, *wshow/2*, has two arguments, the second of which allows you to hide(0), show(1), minimise(2) or maximise(3) any window.

Manipulating Dialogs

So far, you have seen how to create dialogs, but not how to use them. Simply having a dialog called "fred" on your screen does not enable you to perform any



```
| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'90c80000
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'80c80000
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wshow(fred,1), wfocus((fred,1)).
yes

| ?-
```

Fig 5.6 - The finished dialog displayed and set into focus

actions. We will look at the use of dialogs fully in *Chapter 12*: for now, we will concentrate on their programmatic manipulation.

You have already seen how to create and recreate dialogs, and how to add controls to them (controls are fully discussed in *Chapter 6*); you have also seen how to display a hidden dialog and switch focus to one of its controls. There are many more things you can do to a dialog, including getting or setting its title, size, position or status: some of these are described in the following sections.

Changing Window Titles

When you create a dialog (or text window, user window or control), one of the parameters you supply to *wcreate/8* is the title. Once the window is created, you can retrieve or even change the title, using the *wtext/2* predicate. Click back on the console window, and type the following command:

```
?- wtext(fred,X).                                    <enter>
```

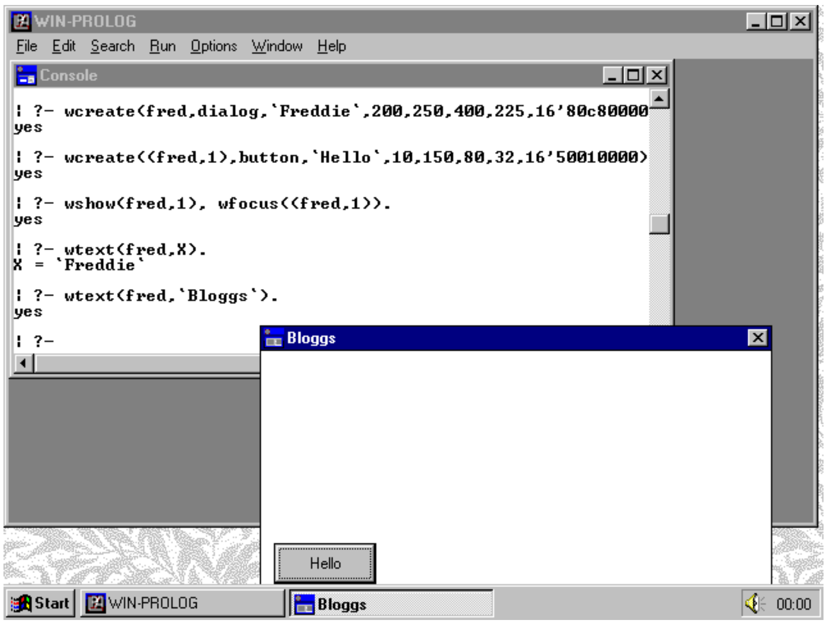
The "text" of a dialog window is its title, so Prolog will return the answer:

```
X = `Freddie`
```

Note that the parameter is returned as a string, using the backward quote characters (`...`): this is the normal data type used for general text parameters in ***WIN-PROLOG***. Now type the command:

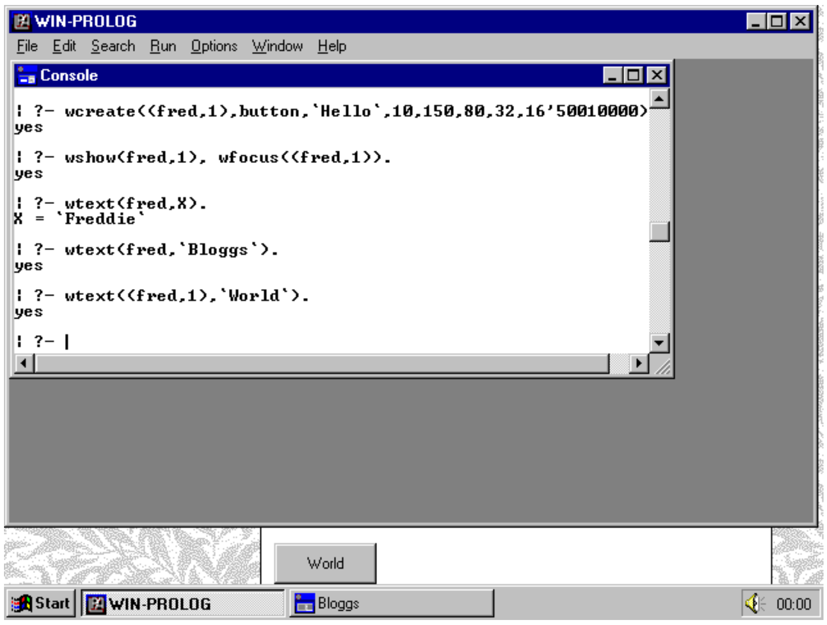
```
?- wtext(fred,`Bloggs`).                             <enter>
```

Because ***WIN-PROLOG***'s main window is on top of the dialog, you will not see



```
| ?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'80c80000
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wshow(fred,1), wfocus((fred,1)).
yes

| ?- wtext(fred,X).
X = `Freddie`

| ?- wtext(fred,`Bloggs`).
yes

| ?-
```

Fig 5.7 - Renaming a dialog with the wtext/2 predicate



```
| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wshow(fred,1), wfocus((fred,1)).
yes

| ?- wtext(fred,X).
X = `Freddie`

| ?- wtext(fred,`Bloggs`).
yes

| ?- wtext((fred,1),`World`).
yes

| ?- |
```

Fig 5.8 - Renaming a button with the wtext/2 predicate

the result immediately: now click on the "fred" dialog, and you will see that the title has changed, as shown in *Fig 5.7*. You can also change the text in controls using the same predicate. Click back on the console window, and type:

```
?- wtext((fred,1),`World`).                          <enter>
```

Instantly, the text of the button "(fred,1)" will change from "Hello" to "World", as shown in *Fig 5.8*. The *wtext/2* predicate can be used to change the text of any window, including the main and console windows. Try the command:

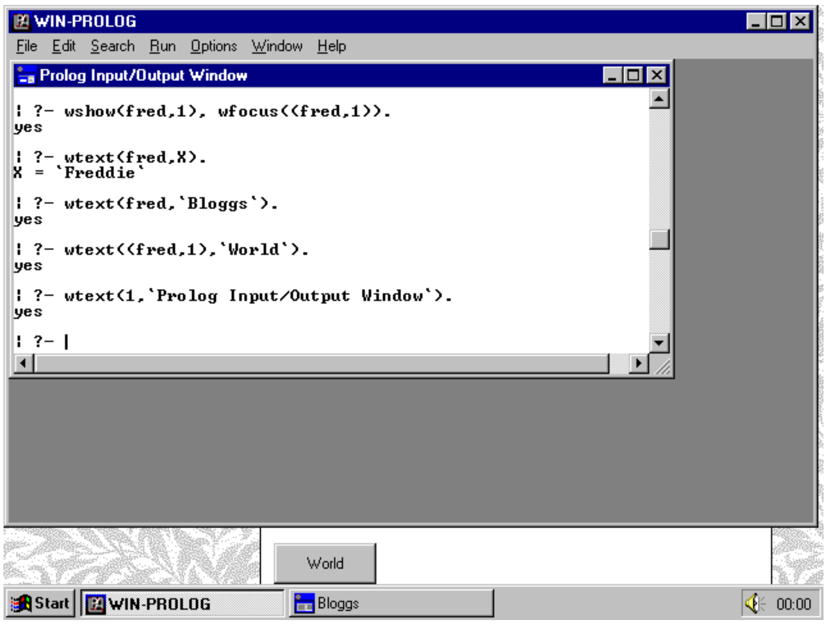
```
?- wtext(1,`Prolog Input/Output Window`).            <enter>
```

The console window has no name as such, but is referred to as the integer "1": this command will instantly change the title of the WIN-PROLOG console, as shown in *Fig 5.9*. If you want to change the main window, use the integer "0" in place of "1".

Resizing Windows

Normally it is considered bad practice to resize windows programmatically: part of the Windows philosophy is that the user alone should dictate the layout of the desktop. There are times, however, when you will want to check or set the sizes of windows, perhaps when implementing a "cascade" or "tile" command, or when restoring a screen layout saved in a previous session. The *wsize/5* predicate allows you to check or set all four position and size parameters for a given window. Type the command:

```
?- wsize(fred,L,T,W,D).                              <enter>
```

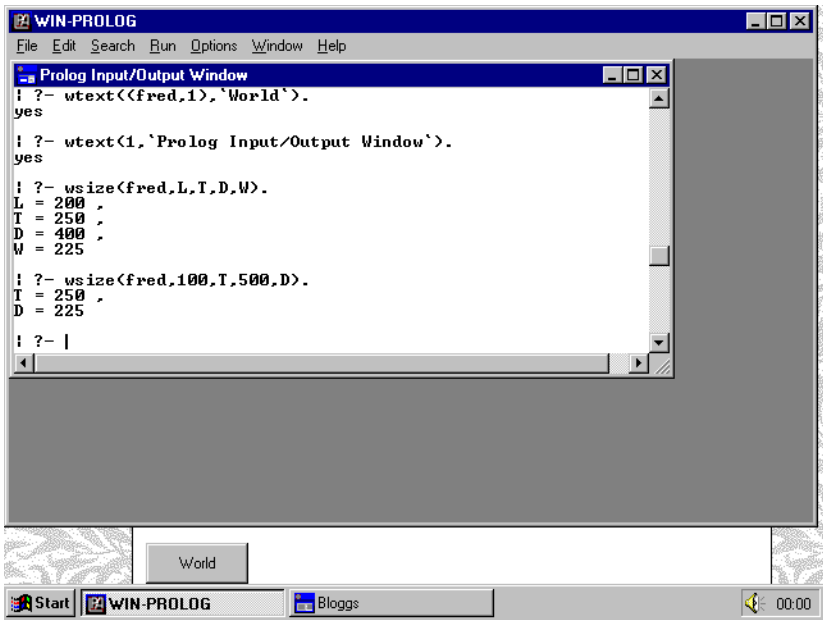

WIN-PROLOG
File Edit Search Run Options Window Help
Prolog Input/Output Window
World
Start
WIN-PROLOG
Bloggs
00:00

```
| ?- wshow(fred,1), wfocus((fred,1)).
yes

| ?- wtext(fred,X).
X = `Freddie`

| ?- wtext(fred,`Bloggs`).
yes

| ?- wtext((fred,1),`World`).
yes

| ?- wtext(1,`Prolog Input/Output Window`).
yes

| ?- |
```

Fig 5.9 - Renaming the console with the wtext/2 predicate

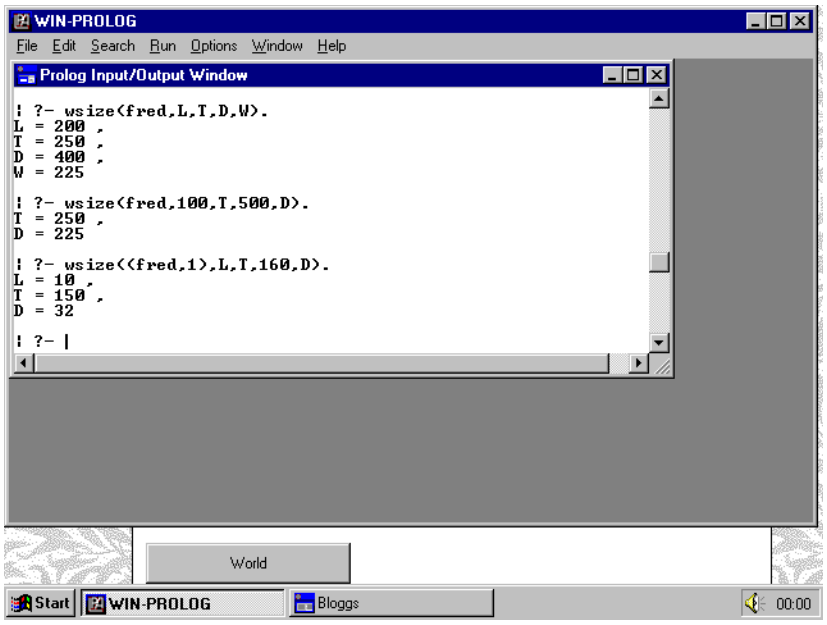
Where variables are given as the parameters, WIN-PROLOG will simply return the current values:

```
L = 200
T = 250
W = 400
D = 225
```


WIN-PROLOG
File Edit Search Run Options Window Help
Prolog Input/Output Window
World
Start
WIN-PROLOG
Bloggs
00:00

```
| ?- wtext((fred,1),`World`).
yes

| ?- wtext(1,`Prolog Input/Output Window`).
yes

| ?- wsize(fred,L,T,D,W).
L = 200 ,
T = 250 ,
D = 400 ,
W = 225

| ?- wsize(fred,100,T,500,D).
T = 250 ,
D = 225

| ?- |
```

Fig 5.10 - Resizing a dialog with the wsize/5 predicate


WIN-PROLOG
File Edit Search Run Options Window Help
Prolog Input/Output Window
World
Start
WIN-PROLOG
Bloggs
00:00

```
| ?- wsize(fred,L,T,D,W).
L = 200 ,
T = 250 ,
D = 400 ,
W = 225

| ?- wsize(fred,100,T,500,D).
T = 250 ,
D = 225

| ?- wsize((fred,1),L,T,160,D).
L = 10 ,
T = 150 ,
D = 32

| ?- |
```

Fig 5.11 - Resizing a button with the wsize/5 predicate

You can change one or more of the parameters simply by including your desired value in the call. For example, type:

?- **wsize(fred,100,T,500,D).** *<enter>*

In this call, you have set new values for the left position and width of the dialog, while returning the existing top position and height, as shown in *Fig 5.10*. As with *wtext/2*, you can apply *wsize/5* to any window. Type the command:

?- **wsize((fred,1),L,T,160,D).** *<enter>*

This will double the width of the button "(fred,1)", and return its other three size and position parameters, as shown in *Fig 5.11*.

Chapter 6 - Control Windows

This chapter describes the use of "control" windows in **WIN-PROLOG**, covering their creation, manipulation and low level programming. These windows provide the working elements of complex and dynamic dialogs which go well beyond the capabilities of the standard dialogs.

Control Windows

In *Chapter 5*, we looked at how to create "dialog" windows, which provide the outer shell of all dialogs used in **WIN-PROLOG**, but (apart from a simple example) we did not address the creation and manipulation of their buttons, list boxes, edit and other controls.

In the Windows environment, all controls are windows in their own right, and this leads to some very powerful properties. For a start, a great many Windows functions can be applied equally to top level dialogs and to their controls, considerably reducing the number of built-in predicates needed to implement a full dialog management system. It also means that it is relatively easy to define new, custom controls to augment the standard Windows set: one such extension is built into **WIN-PROLOG**, but others could easily be defined in C/C++ using the standard Windows SDK.

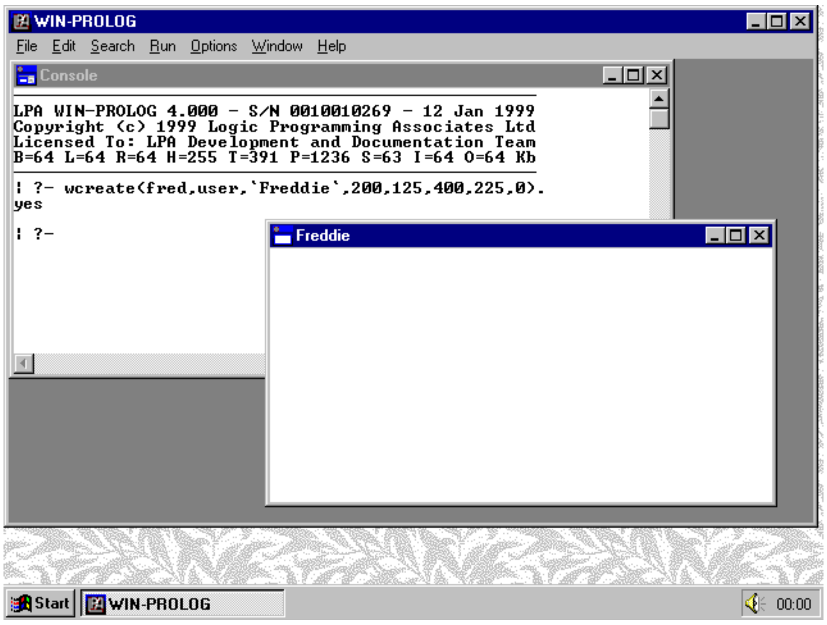
Control Classes

Windows provides several control "classes", each of which defines a family of related control types. For example, the "button" class includes push buttons, radio buttons, check boxes, and even group boxes, while the "edit" class includes all kinds of single or multi line, scrollable or non-scrollable edit boxes. In this chapter we will examine each class in turn; a formal list of the available classes is given in *Appendix E*.

In order to look at controls, we need to create a dialog into which to put them. For most purposes, "user" windows behave in the same manner as dialogs, but have the advantage of being slightly easier to create and view, since they form part of the MDI environment (see *Chapter 5*). Because of this, we will use a user window rather than a dialog to demonstrate each of the individual controls. If you are already in **WIN-PROLOG**, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start **WIN-PROLOG** up, and type the following command:

```
?- wcreate(fred,user,`Freddie`,200,125,400,225,0).        <enter>
```

In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in <*italics*>. This will create a user window, ready for experimentation, as shown in *Fig 6.1*.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?-
```

Fig 6.1 - A user window ready for experimentation

The Button Class

In *Chapter 5* we briefly looked at the creation of a simple pushbutton while experimenting with dialogs: now we will examine buttons further. Begin by clicking back on the console window, and then type the following command to create a standard pushbutton in the user window "fred" (**bold** letters only):

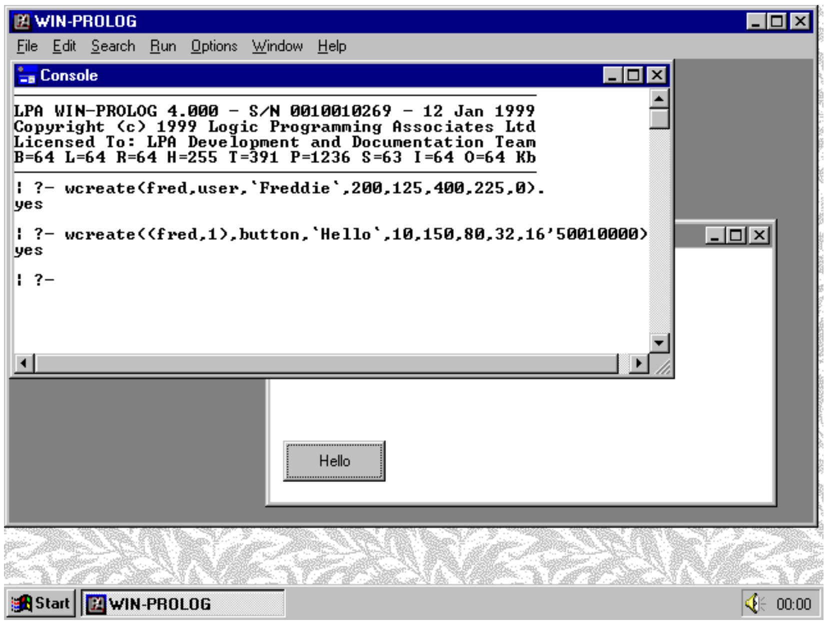
```
?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000).
                                                        <enter>
```

This will create a single button in your user window, with an ID of "1", and the label "Hello", at position (10,10) and of size (80*32), as shown in *Fig 6.2*. The style parameter has been given in hexadecimal notation (16'....), because this makes it easier to compute complex styles. For now, all you need to know is that the "5" digit includes two bits which mean "child window" and "visible", and the "1" digit makes the control part of the "tab stop" cycle used to move between controls in a dialog. Styles are explained in fully in *Appendix D*.

The least significant hexadecimal digit of the button style dictates the type of button, as we shall now see. Now type the following command:

```
?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010003).
                                                        <enter>
```

This time, we are creating a button with an ID of "2" (if we had reused "1", then the existing button would have been destroyed before creating the new one). Because the least significant digit of the style is "3", rather than "0", the new button is created as an automatic checkbox, as shown in *Fig 6.3*. While we are here, create two more buttons with the commands:



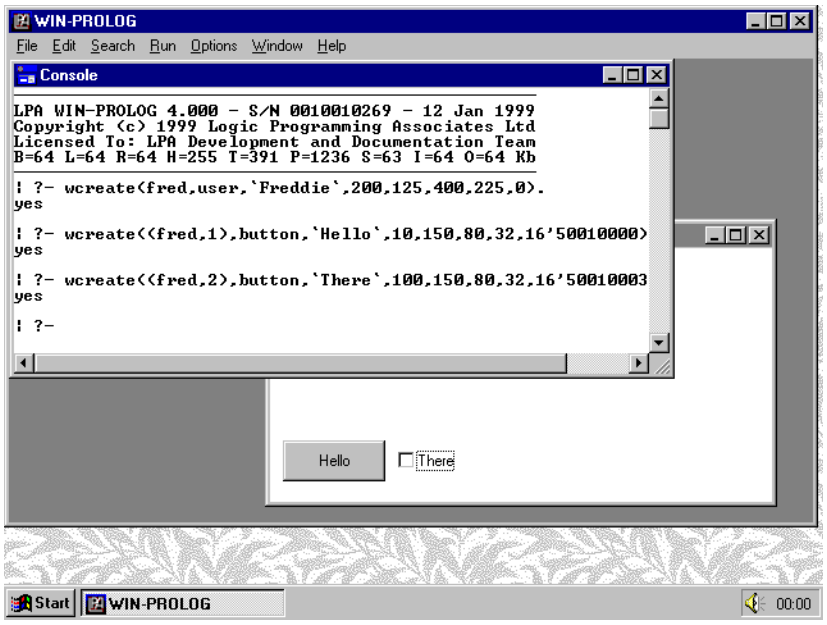
```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?-
```

Fig 6.2 - A push button added to the user window

```
?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010009).
```
<enter>

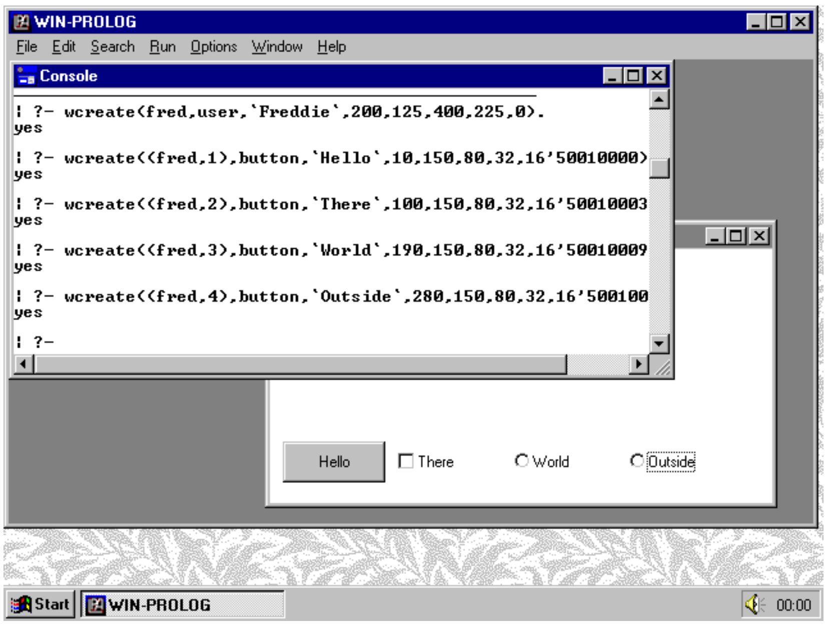
```
?- wcreate((fred,4),button,`Outside`,280,150,80,32,16'50010009).
```
<enter>

The two new buttons have been created with IDs of "3" and "4" respectively, and their style includes a least significant digit of "9", which creates them as



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010003
yes

| ?-
```

Fig 6.3 - A checkbox button added to the user window



```
| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010003
yes

| ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010009
yes

| ?- wcreate((fred,4),button,`Outside`,280,150,80,32,16'500100
yes

| ?-
```

Fig 6.4 - Two radio buttons added to the user window

automatic radio buttons, as shown in *Fig 6.4*. If you click on the user window, you can experiment with pressing the buttons to see how they behave. Notice how the checkbox alternates between a checked and unchecked state whenever you click it, and how whenever you click on one of the radio buttons to select it, the other is deselected.

Programming Button Controls

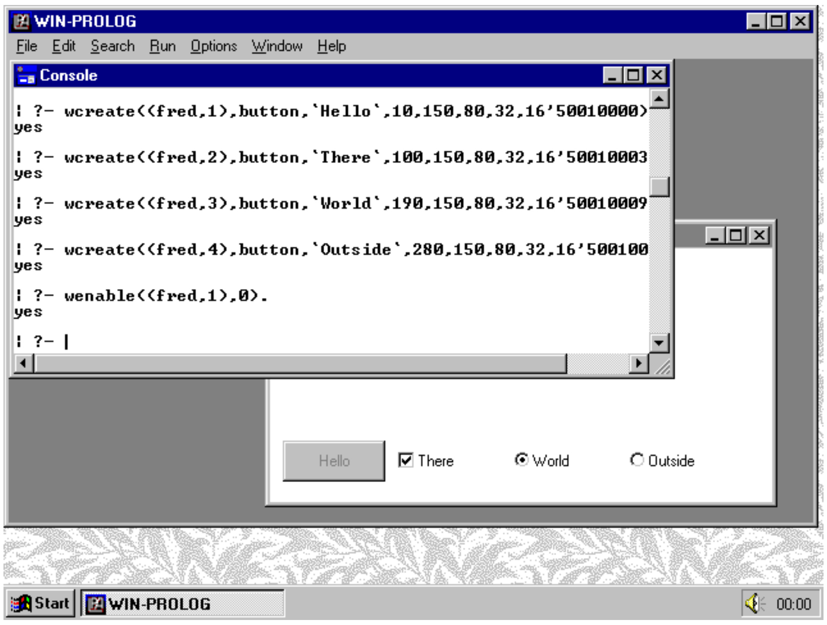
You can use generic window handling predicates to manipulate buttons: for example, *wtext/2* allows you to retrieve or change a button's label, and *wsize/5* allows you to get or set its size and position (see *Chapter 5*). Two more predicates, one generic and one specific, are useful in handling buttons.

It is frequently desirable to disable buttons (and other controls) so that they cannot be clicked or otherwise selected, for example when a given operation is not possible at some particular moment. The *wenable/2* predicate allows you to get or set a button's enable status. Click back on the console window, and type in the command:

```
?- wenable((fred,1),0).                                    <enter>
```

This sets the enable status of "(fred,1)", the pushbutton currently labelled "Hello", to "0" (false). The result is that the button is greyed out, as shown in *Fig 6.5*, and can no longer be selected. You can reenable a button using "1" (true) as the second argument.

All types of window can be enabled or disabled with *wenable/2*, but this is not true of another predicate, *wbtnsel/2*, which is specific to buttons. This predicate



```
¦ ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

¦ ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010003
yes

¦ ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010009
yes

¦ ?- wcreate((fred,4),button,`Outside`,280,150,80,32,16'500100
yes

¦ ?- wenable((fred,1),0).
yes

¦ ?- |
```

Fig 6.5 - Disabling a button with the wenable/2 predicate

allows you to get or set the selection status of a button. Type the command:

?- **wbtnsel((fred,2),S).** *<enter>*

This will return the result "S = 1" or "S = 0", depending upon whether the checkbox "(fred,2)" is checked or unchecked respectively. You can force a new selection state by supplying the second argument; for example, type:

?- **wbtnsel((fred,3),0), wbtnsel((fred,4),0).** *<enter>*

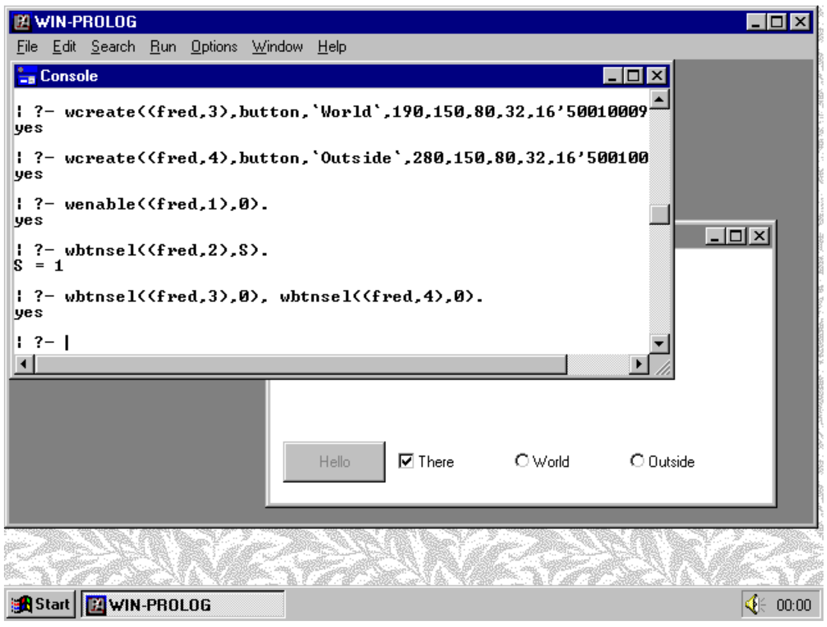
This will force both radio buttons, "(fred,3)" and "(fred,4)", to be simultaneously deselected, as shown in *Fig 6.6*. A more surprising result will occur if you type:

?- **wbtnsel((fred,3),1), wbtnsel((fred,4),1).** *<enter>*

This forces both radio buttons to be simultaneously selected, as shown in *Fig 6.7*, something which is impossible using the keyboard or mouse. Notice how the "automatic" behaviour of radio buttons is bypassed by the *wbtnsel/2* predicate.

The Edit Class

The "edit" class is every bit as important as the button class, supporting as it does all forms of typed keyboard input. Whenever you are able to type anything directly into a Windows dialog, you are using an "edit" control window. Edit controls process the keyboard in a standard way, and support both cut/paste editing and single level undo, as well as straight typing. Just as with buttons, the style parameter dictates in which of many types and variations a given edit



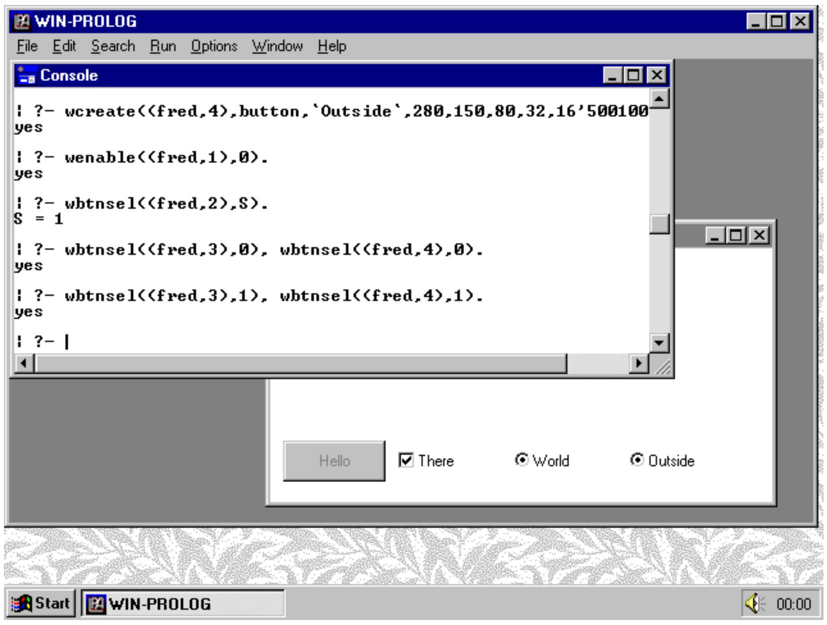
```
| ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010009
yes

| ?- wcreate((fred,4),button,`Outside`,280,150,80,32,16'500100
yes

| ?- wenable((fred,1),0).
yes

| ?- wbtnsel((fred,2),S).
S = 1

| ?- wbtnsel((fred,3),0), wbtnsel((fred,4),0).
yes

| ?- |
```

Fig 6.6 - Deselecting two radio buttons with wbtnsel/2

control will be created. To clear up the button examples before we commence, recreate the "fred" user window by typing the command:

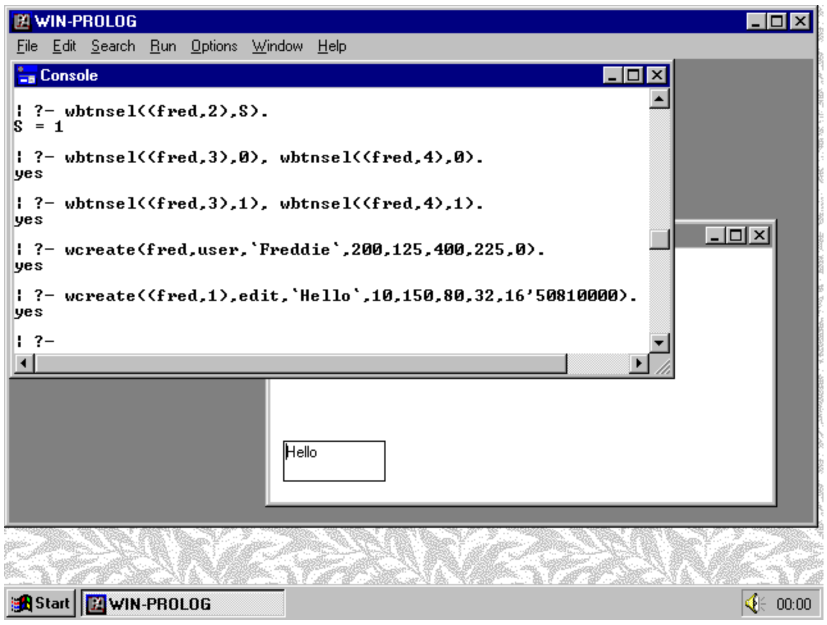
?- **wcreate(fred,user,`Freddie`,200,125,400,225,0).** *<enter>*

Because you are reusing the name "fred", this will delete the existing user window and all of its controls, once again leaving you with an empty user window. Click back on the console, and type the command:



```
| ?- wcreate((fred,4),button,`Outside`,280,150,80,32,16'500100
yes

| ?- wenable((fred,1),0).
yes

| ?- wbtnsel((fred,2),S).
S = 1

| ?- wbtnsel((fred,3),0), wbtnsel((fred,4),0).
yes

| ?- wbtnsel((fred,3),1), wbtnsel((fred,4),1).
yes

| ?- |
```

Fig 6.7 - Selecting two radio buttons with wbtnsel/2



```
| ?- wbtnsel((fred,2),S).
S = 1

| ?- wbtnsel((fred,3),0), wbtnsel((fred,4),0).
yes

| ?- wbtnsel((fred,3),1), wbtnsel((fred,4),1).
yes

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),edit,`Hello`,10,150,80,32,16'50810000).
yes

| ?-
```

Fig 6.8 - An edit control added to a new user window

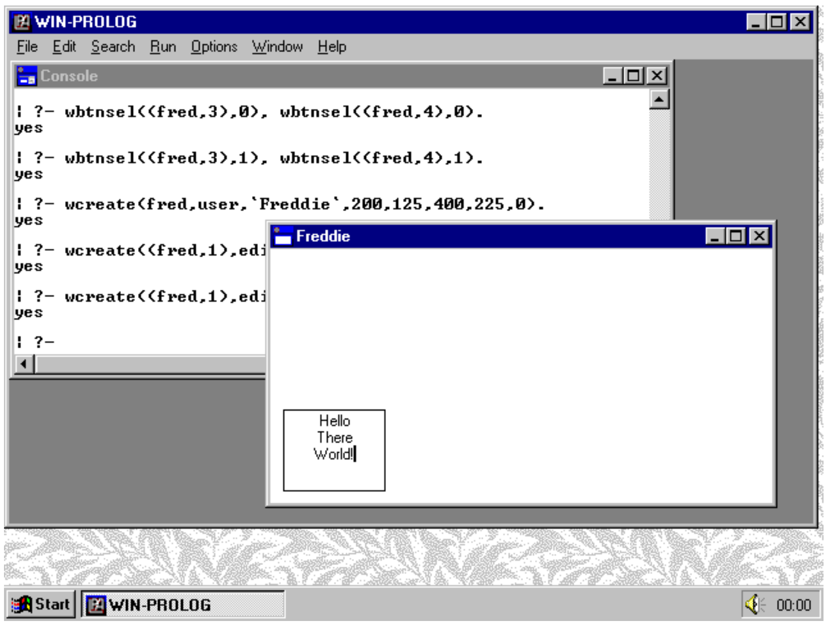
```
?- wcreate((fred,1),edit,`Hello`,10,150,80,32,16'50810000).
                                                    <enter>
```

This will create an edit control with an ID of "1", and the initial contents of "Hello", at the bottom left hand corner of "fred", as shown in*Fig 6.8*. Notice how the text argument denotes the contents of an edit window, since it has no title. The style used here includes the same "5" as before, meaning "child window" and "visible", as well as the same "1", meaning "tab stop". The extra "8" digit simply puts a border around the edit control: without this, all you would see would be the text "Hello", without any indication of the size of the field.

If you click on the new edit window, you will find that you can delete, insert into, or append to the text in the window, up to but not exceeding the full width of the window. Just as with buttons, the lower digits of the edit control style can be used to change its behaviour. For example, replace the existing edit control by clicking back on the console, and typing the command:

```
?- wcreate((fred,1),edit,`Hello`,10,125,80,64,16'50810005).
                                                    <enter>
```

This will create a slightly larger edit control, with two additional properties. The least significant "5" digit contains two bits which mean "multiline" and "centred". Click on the edit control and type into it. No one line may exceed its width, and you will only be allowed to type up to three lines, as shown in *Fig 6.9*. Other style bits allow you to enable horizontal and/or vertical scrolling, with or without attendant scroll bars, to declare the window read-only, and more beside: see *Appendix D* for a complete description of the available styles.



```
| ?- wbtnsel((fred,3),0), wbtnsel((fred,4),0).
yes

| ?- wbtnsel((fred,3),1), wbtnsel((fred,4),1).
yes

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),edi
yes

| ?- wcreate((fred,1),edi
yes

| ?-
```

```
Hello
There
World!
```

Fig 6.9 - An edit control with three lines of centred text

Programming Edit Controls

The predefined controls in text windows are in fact edit controls, their only additional special property being that each one's size automatically tracks that of its parent window. This simple acknowledgement means that you already know how to program edit controls: see the detailed descriptions in *Chapter 3*. We need not reexamine all the features here: suffice it to say that you can use *wedtfnd/6* to search for text in an edit control, *wedtsel/3* to retrieve or make selections, and *wedttxt/2* to retrieve or replace selected text. Other functions, such as *wedtlin/4*, *wedtpxy/4* and *wcount/4* allow you to perform computations on the contents of an edit control.

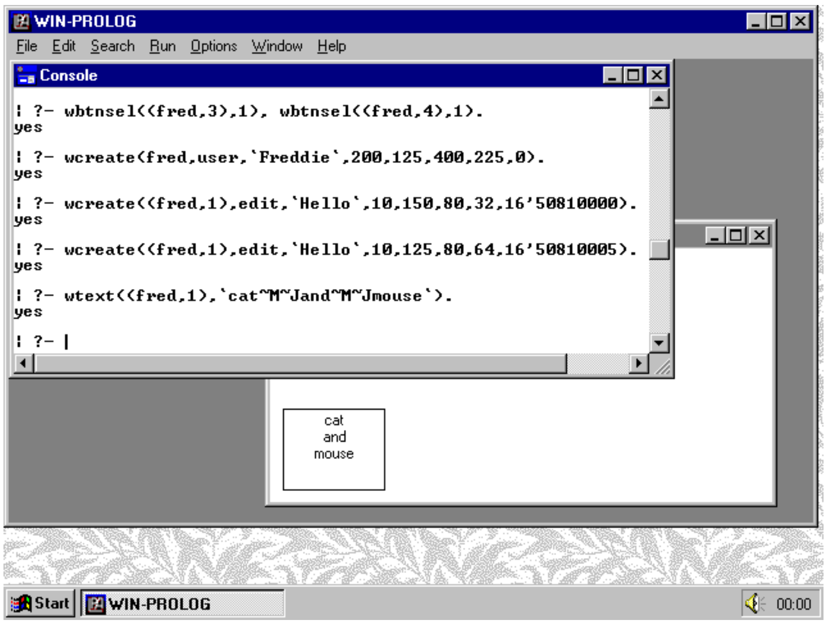
Just as with other windows, you can read or write the entire contents of an edit control with *wtext/2*. For example, click on the console window and type the command:

?- **wtext((fred,1),`cat~M~Jand~M~Jmouse`).** *<enter>*

This will replace the entire contents of the edit window "(fred,1)" with the three lines given, as shown in *Fig 6.10*. Note you could have performed the same replacement with the pair of calls:

?- **wedtsel((fred,1),0,64000),**
wedttxt((fred,1),`cat~M~Jand~M~Jmouse`). *<enter>*

The power of *wedttxt/2* in conjunction with *wedtsel/3* is that it allows you to read or write portions of an edit window, rather than the entire contents as in the case of *wtext/2*.



```
| ?- wbtnsel((fred,3),1), wbtnsel((fred,4),1).
yes

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),edit,`Hello`,10,150,80,32,16'50810000).
yes

| ?- wcreate((fred,1),edit,`Hello`,10,125,80,64,16'50810005).
yes

| ?- wtext((fred,1),`cat~M~Jand~M~Jmouse`).
yes

| ?- |
```

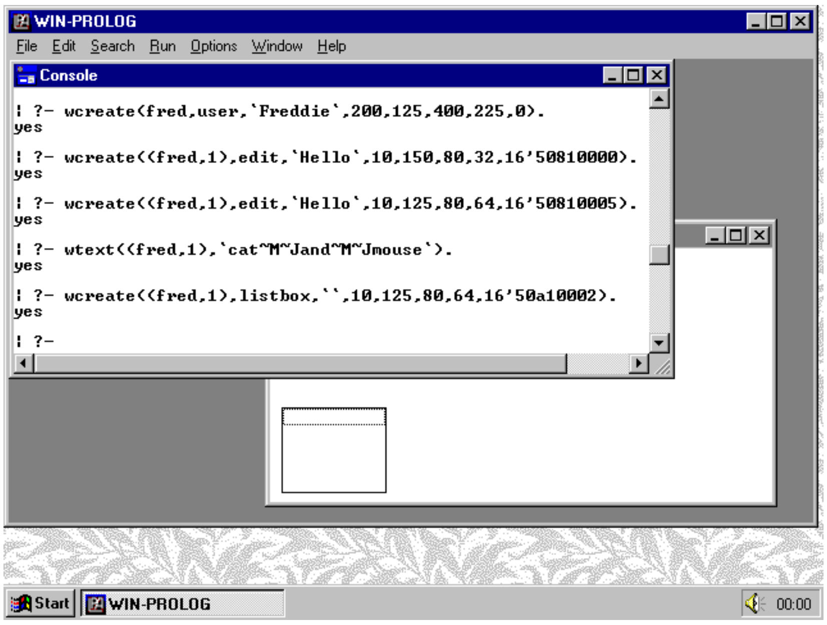
```
cat
and
mouse
```

Fig 6.10 - Replacing text in an edit control with wtext/2

The Listbox Class

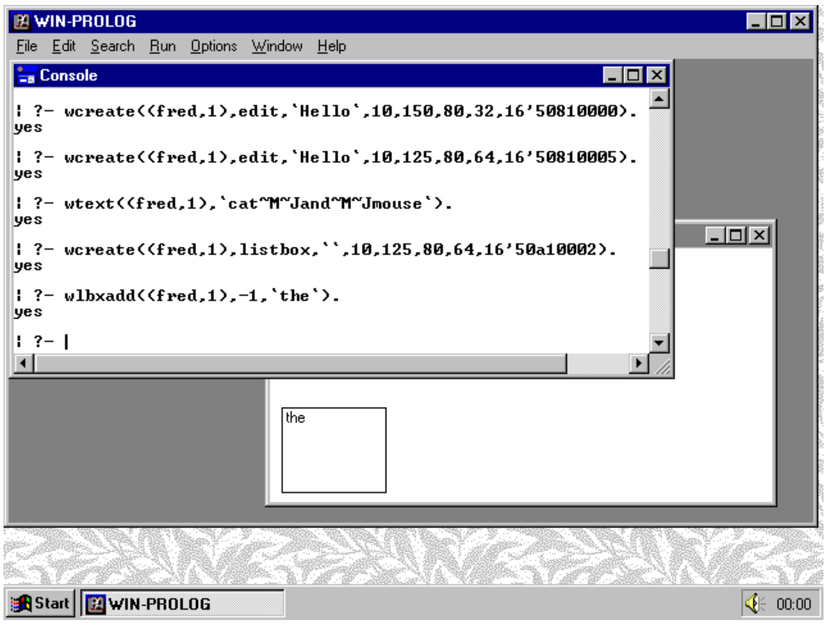
Another widely used type of control is the list box, in which the user selects one or more items, depending upon the style, simply by clicking on them with the mouse, or by using the keyboard. Unlike edit controls, the user cannot actually change the contents of a list box, but only its selection. Replace the existing edit control with a list box by typing the command:

?- **wcreate((fred,1),listbox,``,10,125,80,64,16'50a10002).** *<enter>*



```
| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),edit,`Hello`,10,150,80,32,16'50810000).
yes

| ?- wcreate((fred,1),edit,`Hello`,10,125,80,64,16'50810005).
yes

| ?- wtext((fred,1),`cat~M~Jand~M~Jmouse`).
yes

| ?- wcreate((fred,1),listbox,``,10,125,80,64,16'50a10002).
yes

| ?-
```

Fig 6.11 - A listbox control added to the user window



```
| ?- wcreate((fred,1),edit,`Hello`,10,150,80,32,16'50810000).
yes

| ?- wcreate((fred,1),edit,`Hello`,10,125,80,64,16'50810005).
yes

| ?- wtext((fred,1),`cat~M~Jand~M~Jmouse`).
yes

| ?- wcreate((fred,1),listbox,``,10,125,80,64,16'50a10002).
yes

| ?- wlbxadd((fred,1),-1,`the`).
yes

| ?- |
```

Fig 6.12 - An item added to a listbox by wlbxadd/3

Because you have reused the ID code 1, the existing control "(fred,1)" is replaced by an empty list box, as shown in *Fig 6.11*. The hexadecimal style here denotes a visible child ("5") window, with a vertical scroll bar and border ("a"), which is part of the tab stop cycle ("1") and which automatically sorts its entries ("2"). Once again, refer to *Appendix D* for a full description of the available styles.

Notice that the vertical scroll bar requested in the style is not actually visible: list boxes only display this when they contain too many items to show without scrolling. Notice also that the text parameter is given as the empty string: list boxes are one of the few types of window which do not have a meaningful text property.

Programming Listbox Controls

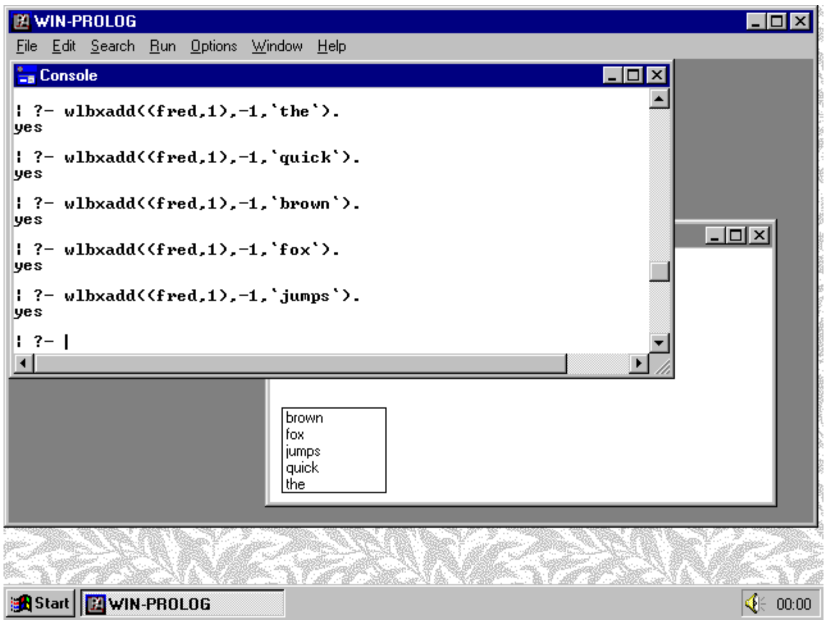
List boxes have a similar set of programming predicates to those provided for edit controls. Whereas the latter are all named *wedt???/n*, the list box controls use the convention *wlbx???/n*. To add items to a list box, you use the *wlbxadd/3*: this predicate allows you to insert an item at a specific location in the list box, or to insert it at the default position. Type the following command:

?- **wlbxadd((fred,1),-1,`the`).** *<enter>*

This will place the string "the" at the default location (denoted by "-1") in the list box "(fred,1)", as shown in *Fig 6.12*. Once again, strings (`...`) are used for the text argument, rather than atoms. Add four more strings by typing:

?- **wlbxadd((fred,1),-1,`quick`).** *<enter>*

?- **wlbxadd((fred,1),-1,`brown`).** *<enter>*



```
| ?- wlbxadd((fred,1),-1,`the`).
yes

| ?- wlbxadd((fred,1),-1,`quick`).
yes

| ?- wlbxadd((fred,1),-1,`brown`).
yes

| ?- wlbxadd((fred,1),-1,`fox`).
yes

| ?- wlbxadd((fred,1),-1,`jumps`).
yes

| ?- |
```

```
brown
fox
jumps
quick
the
```

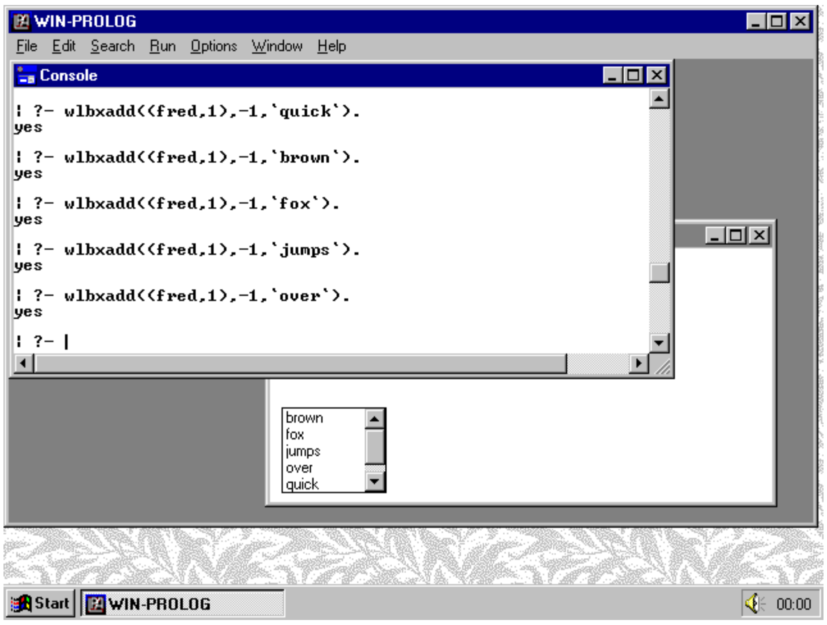
Fig 6.13 - Three more items added to the list box

?- **wlbxadd((fred,1),-1,`fox`).** *<enter>*

?- **wlbxadd((fred,1),-1,`jumps`).** *<enter>*

The list box will now contain five entries, given in alphabetical order, as shown in *Fig 6.13*. Now add a sixth string by typing:

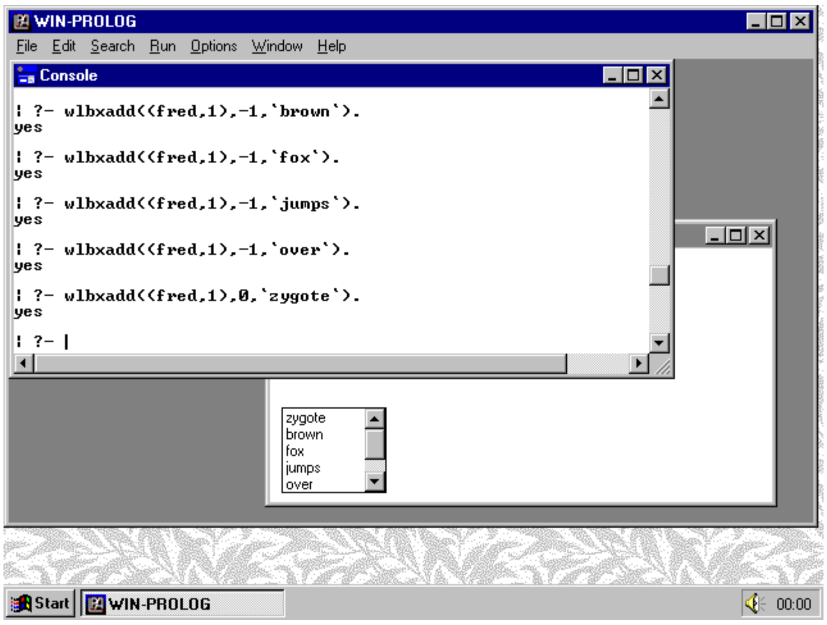
?- **wlbxadd((fred,1),-1,`over`).** *<enter>*



```
| ?- wlbxadd((fred,1),-1,`quick`).
yes

| ?- wlbxadd((fred,1),-1,`brown`).
yes

| ?- wlbxadd((fred,1),-1,`fox`).
yes

| ?- wlbxadd((fred,1),-1,`jumps`).
yes

| ?- wlbxadd((fred,1),-1,`over`).
yes

| ?- |
```

```
brown
fox
jumps
over
quick
```

Fig 6.14 - A scrollbar shown automatically when needed



```
| ?- wlbxadd((fred,1),-1,`brown`).
yes

| ?- wlbxadd((fred,1),-1,`fox`).
yes

| ?- wlbxadd((fred,1),-1,`jumps`).
yes

| ?- wlbxadd((fred,1),-1,`over`).
yes

| ?- wlbxadd((fred,1),0,`zygote`).
yes

| ?- |
```

```
zygote
brown
fox
jumps
over
```

Fig 6.15 - An item put at a specific location by wlbxadd/3

Notice how a scroll bar has suddenly appeared, as shown in *Fig 6.14*: this is because you have now added too many items for the list box to display all at once. Add one final string:

?- **wlbxadd((fred,1),0,`zygote`).** *<enter>*

This time, rather than inserting your string at the correct alphabetic position, it is been placed at the beginning of the list box, as shown in *Fig 6.15*. You can explicitly place any item anywhere, even in a sorted list box, by giving the position (starting at 0) in which you want it. The special "position" of -1 means simply "store at default position", which is either at the end, or in alphabetical order, depending upon the list box style.

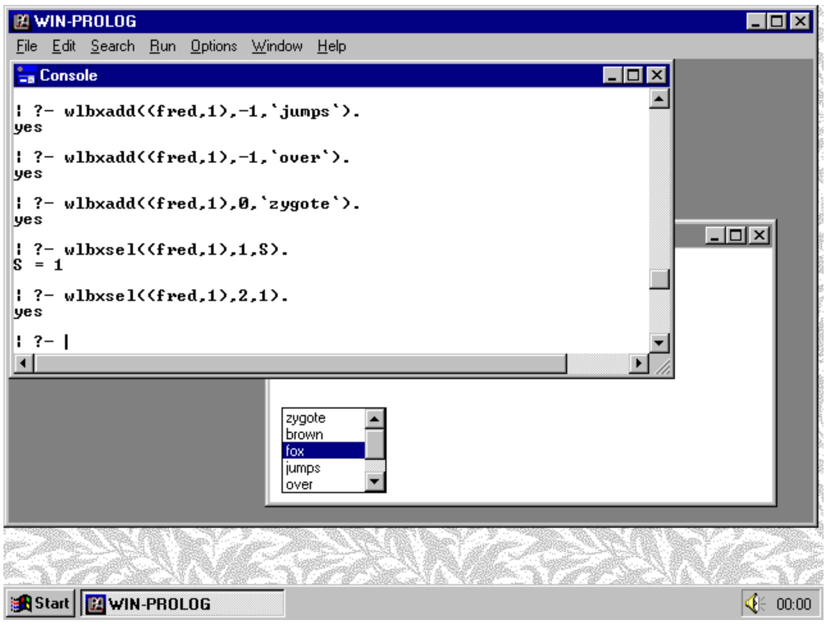
If you click on the list box, you can check that it works correctly in response to mouse and keyboard selections. Having made a selection, say of the word "brown", click back on the console window, and type the command:

?- **wlbxsel((fred,1),1,S).** *<enter>*

This will return the value "1" to indicate that item 1 (the second item) in list box "(fred,1)" is currently selected. The value "0" is returned for a list box item which is not selected. Now type the command:

?- **wlbxsel((fred,1),2,1).** *<enter>*

The selection in "(fred,1)" will change to highlight the third entry, as shown in *Fig 6.16*. You can remove the highlight altogether from "(fred,1") with the following command:



```
| ?- wlbxadd((fred,1),-1,`jumps`).
yes

| ?- wlbxadd((fred,1),-1,`over`).
yes

| ?- wlbxadd((fred,1),0,`zygote`).
yes

| ?- wlbxsel((fred,1),1,S).
S = 1

| ?- wlbxsel((fred,1),2,1).
yes

| ?- |
```

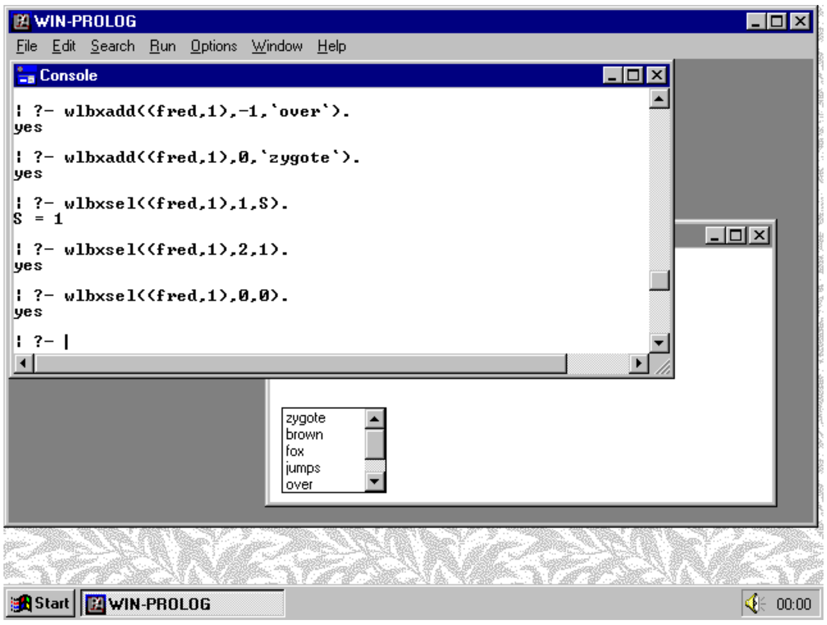
```
zygote
brown
fox
jumps
over
```

Fig 6.16 - Selecting an item with the wlbxsel/3 predicate

?- **wlbxsel((fred,1),0,0).** *<enter>*

This "selects" the first item of the list box, but with an "unselected" status, and effectively removes selection from the entire list box, as shown in *Fig 6.17*.

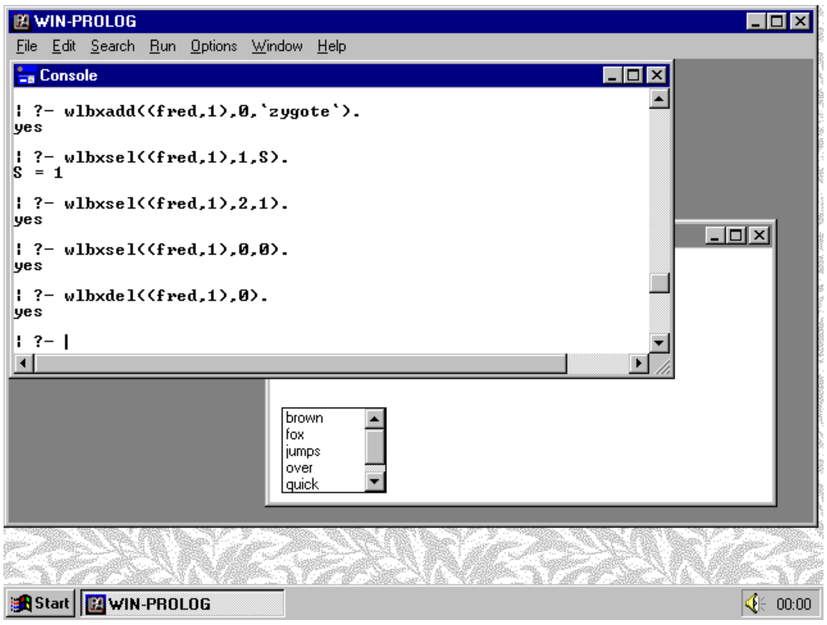
Further items can be added to, and existing ones removed from a list box at any time. Type the following command:



```
| ?- wlbxadd((fred,1),-1,`over`).
yes

| ?- wlbxadd((fred,1),0,`zygote`).
yes

| ?- wlbxsel((fred,1),1,S).
S = 1

| ?- wlbxsel((fred,1),2,1).
yes

| ?- wlbxsel((fred,1),0,0).
yes

| ?- |
```

```
zygote
brown
fox
jumps
over
```

Fig 6.17 - The selection cleared with wlbxsel/3



```
| ?- wlbxadd((fred,1),0,`zygote`).
yes

| ?- wlbxsel((fred,1),1,S).
S = 1

| ?- wlbxsel((fred,1),2,1).
yes

| ?- wlbxsel((fred,1),0,0).
yes

| ?- wlbxdel((fred,1),0).
yes

| ?- |
```

Fig 6.18 - An item deleted with the lbxdel/2 predicate

?- **wlbxdel((fred,1),0).** *<enter>*

This removes the entry "0" (the first item) from this list box, as shown in *Fig 6.18*. If you want to retrieve the text of a list box entry, use a command such as:

?- **wlbxget((fred,1),1,T).** *<enter>*

This will return the string "`fox`", which is the second item in the list box. A final list box predicate, *wlbxfnd/4*, can be used to search the entries for a list box for one matching the given string. Type the command:

?- **wlbxfnd((fred,1),-1,`qui`,P).** *<enter>*

This will search the entire list box for the first occurrence of an entry beginning with the characters "qui", and will return its position (in this case, "4"). Note that *wlbxfnd/4* always begins the search from one place after the position you give, which is why this example specifies "-1" (ie, start searching at "0"). This feature means that, having found an entry, you can search for the next by directly using the existing index as the new starting point.

List boxes can be created in a variety of styles, as enumerated in *Appendix D*, and some of these styles allow multiple choice selections. All of the predicates described in this section function identically in multi-choice list boxes, except for *wlbxsel/3*. When you select or deselect an entry using this predicate in a single choice list box, any existing selection is cancelled: this is not the case in a multi-choice list box. There is a quick way of selecting or deselecting all entries in such a list box: using a position of "-1" causes the given selection status to be applied to all entries in a multi-choice list box.

The Combobox Class

Combo boxes are very similar to list boxes, being comprised of a single line edit control and a single choice list box. Once again, there are many different styles within this control class (see *Appendix D*). Replace the existing list box control with a combo box by typing the command:

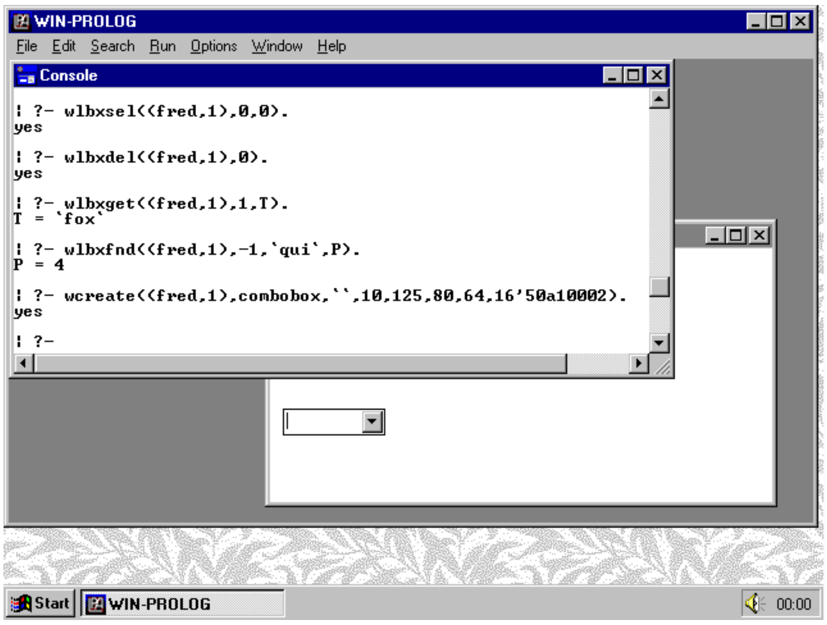
```
?- wcreate((fred,1),combobox,``,10,125,80,64,16'50a10002).
                                                    <enter>
```

Once again, you have reused the ID code 1, and the existing control "(fred,1)" is replaced by your new combo box, as shown in *Fig 6.19*. The hexadecimal style here denotes a visible child ("5") window, with a vertical scroll bar and border ("a"), which is part of the tab stop cycle ("1") and which has a drop down list box ("2").

Programming Combobox Controls

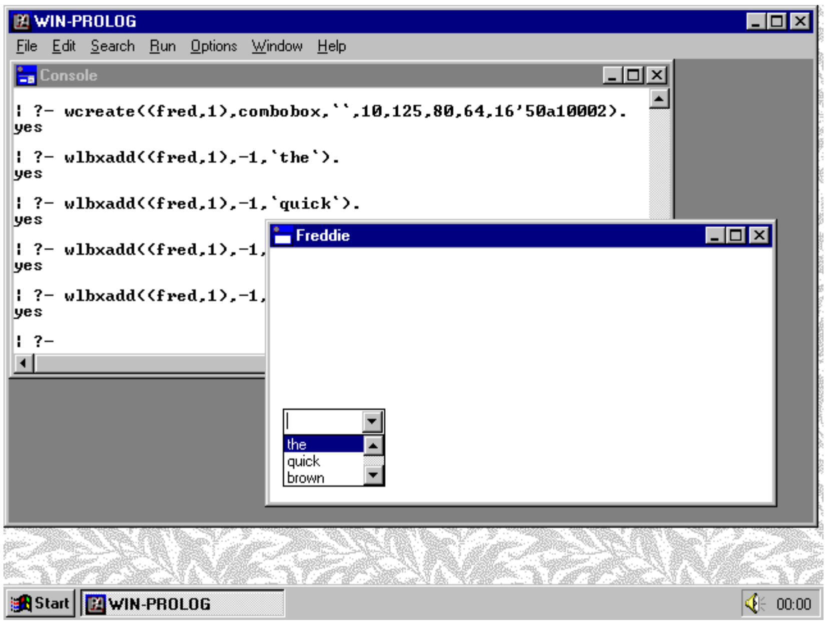
The similarity between combo boxes and list boxes extends to their programming. The list box component of a combo box can be directly programmed with the *wlbx???/n* predicates. For example, type the commands:

```
?- wlbxadd((fred,1),-1,`the`).                  <enter>

?- wlbxadd((fred,1),-1,`quick`).                <enter>

?- wlbxadd((fred,1),-1,`brown`).                <enter>
```



```
| ?- wlbxsel((fred,1),0,0).
yes

| ?- wlbxdel((fred,1),0).
yes

| ?- wlbxget((fred,1),1,T).
T = `fox`

| ?- wlbxfnd((fred,1),-1,`qui`,P).
P = 4

| ?- wcreate((fred,1),combobox,``,10,125,80,64,16'50a10002).
yes

| ?-
```

Fig 6.19 - A combobox added to the user window



```
| ?- wcreate((fred,1),combobox,``,10,125,80,64,16'50a10002).
yes

| ?- wlbxadd((fred,1),-1,`the`).
yes

| ?- wlbxadd((fred,1),-1,`quick`).
yes

| ?- wlbxadd((fred,1),-1,
yes

| ?- wlbxadd((fred,1),-1,
yes

| ?-
```

Fig 6.20 - Items added to the combobox by wlbxadd/3

?- **wlbxadd((fred,1),-1,`fox`).** *<enter>*

You won't actually see anything yet, because the list box component of "(fred,1)" is currently concealed. Now click on the drop-down icon of your list box, and you will see the entries, as shown in *Fig 6.20*. Notice that they have not been sorted this time: we did not request sorting in the style parameter when we created this combo box.

Apart from being able to use the full set of list box predicates on a combo box, you can use *wtext/2* to set or read the contents of its edit control. Click back on the console window, and type the command:

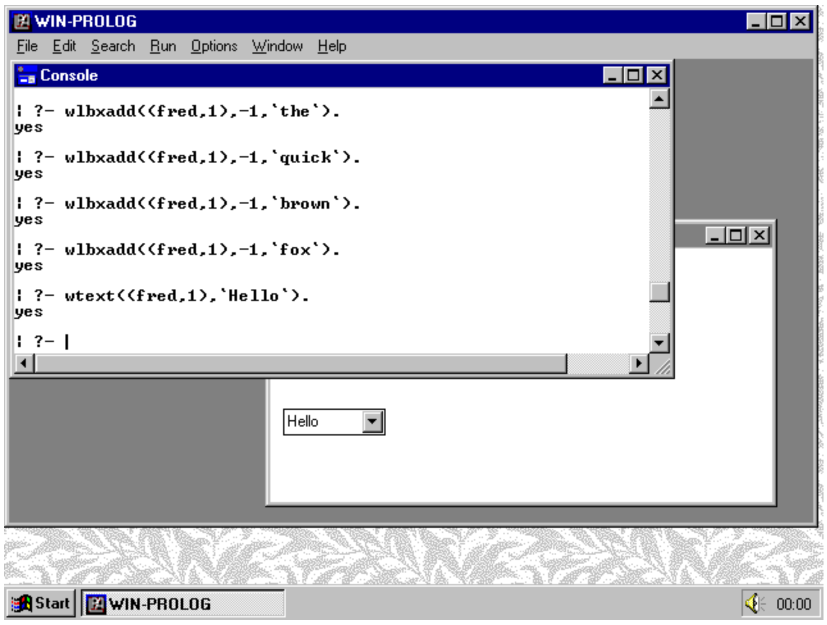
?- **wtext((fred,1),`Hello`).** *<enter>*

This will replace the contents of the edit control in "(fred,1)" with the text "Hello", as shown in *Fig 6.21*. Note, however, that you cannot use the *wedt???/n* predicates on a combo box: these are limited to true edit controls.

The Static Class

Static controls function mainly to provide labels and other non-editable information in dialogs. Despite their description, they need not remain static: you can alter their text, resize them, and otherwise manipulate them just like any other window. Replace the combo box with a static control using the command:

?-
wcreate((fred,1),static,`one~M~Jtwo~M~Jthree`,10,125,80,64,16'50800001).
<enter>



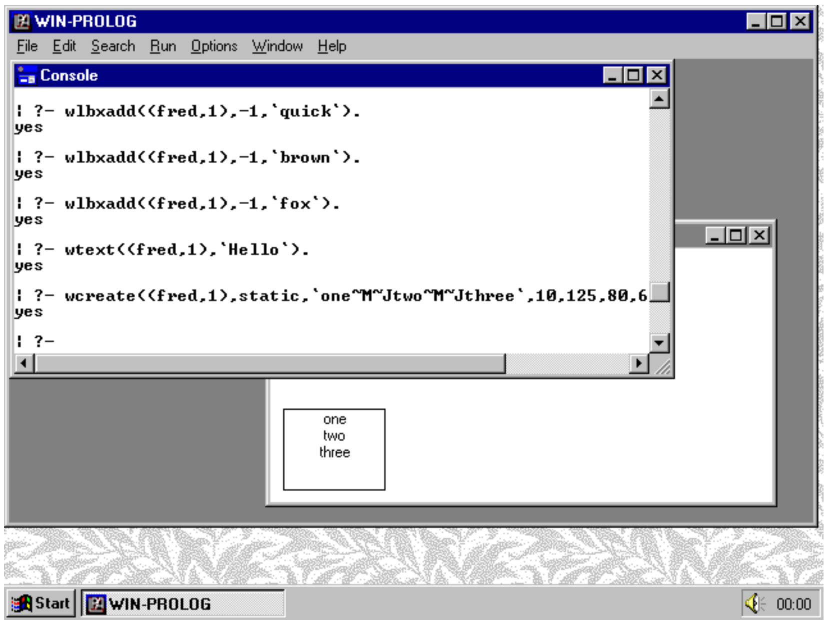
```
| ?- wlbxadd((fred,1),-1,`the`).
yes

| ?- wlbxadd((fred,1),-1,`quick`).
yes

| ?- wlbxadd((fred,1),-1,`brown`).
yes

| ?- wlbxadd((fred,1),-1,`fox`).
yes

| ?- wtext((fred,1),`Hello`).
yes

| ?- |
```

Fig 6.21 - The edit field of a combobox set by wtext/2

This will replace the existing "(fred,1)" control with a static window, as shown in *Fig 6.22*. The style here includes the visible and child bits ("5"), a border ("8"), and centred text ("1"). The other styles are listed in *Appendix D*.

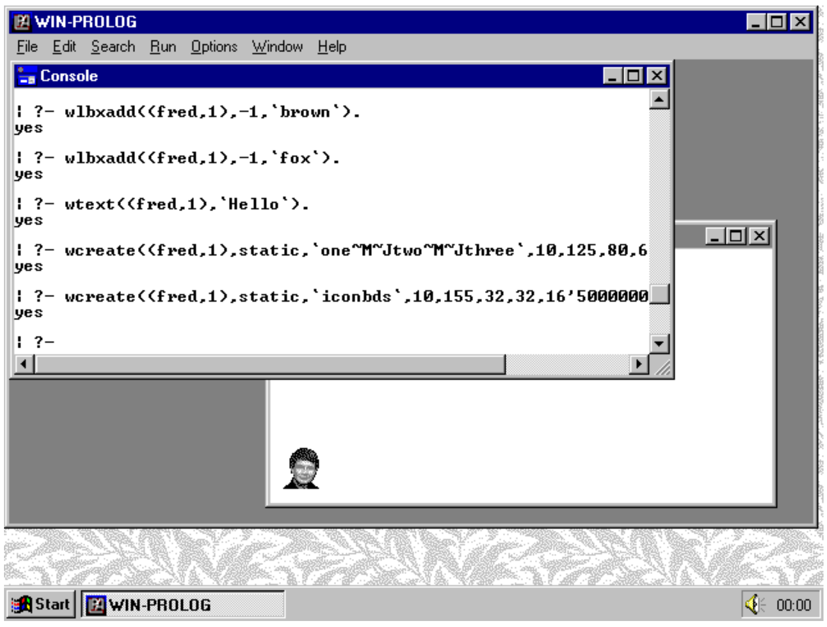
Programming Static Controls

Because of their simple text display functionality, there is not much to program with static controls. The *wtext/2* predicate can be used to read and write the entire contents of a static window, and that's about it. Apart from labelling neighbouring input fields, the main purpose of static controls is to provide



```
| ?- wlbxadd((fred,1),-1,`quick`).
yes

| ?- wlbxadd((fred,1),-1,`brown`).
yes

| ?- wlbxadd((fred,1),-1,`fox`).
yes

| ?- wtext((fred,1),`Hello`).
yes

| ?- wcreate((fred,1),static,`one~M~Jtwo~M~Jthree`,10,125,80,6
yes

| ?-
```

Fig 6.22 - A static control added to the user window



```
¦ ?- wlbxadd((fred,1),-1,`brown`).
yes

¦ ?- wlbxadd((fred,1),-1,`fox`).
yes

¦ ?- wtext((fred,1),`Hello`).
yes

¦ ?- wcreate((fred,1),static,`one~M~Jtwo~M~Jthree`,10,125,80,6
yes

¦ ?- wcreate((fred,1),static,`iconbds`,10,155,32,32,16'5000000
yes

¦ ?-
```

Fig 6.23 - A static control displaying an icon

system-updatable information in status boxes and similar dialogs.

One curious anomaly is a static control style which treats its initial text parameter as the name of an icon, rather than as text. Try the command:

```
?- wcreate((fred,1),static,`iconbds`,10,155,32,32,16'50000003).
                                                          <enter>
```

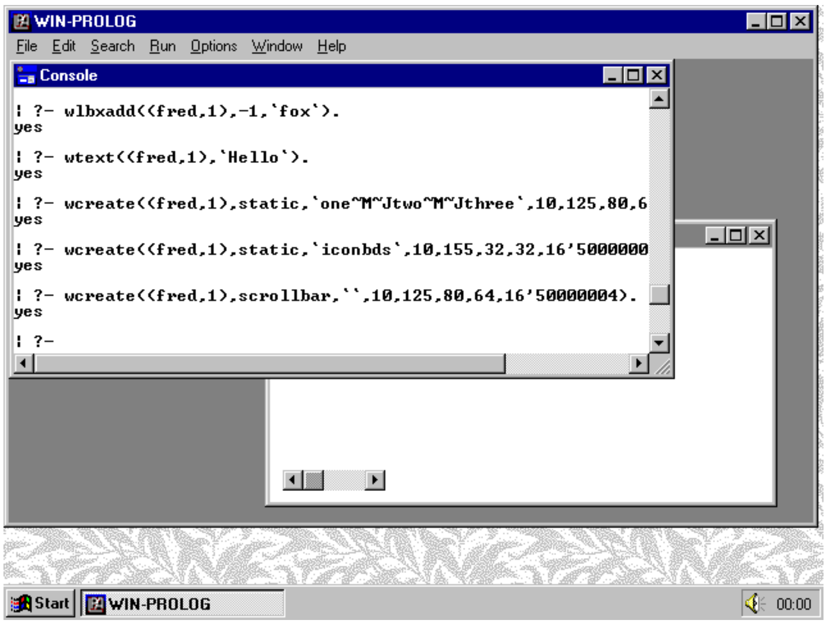
Because of the "3" digit in the style, this call results in a borderless display of the "iconbds" icon (a shameless portrait of the author!), as shown in *Fig 6.23*. By using a resource editor to add your own icons to *WIN-PROLOG*, you could easily display them in dialogs using this technique.

The Scrollbar Class

The final standard control class is something of a curiosity. Many of the other control class can include horizontal and/or vertical scroll bars, but in the "scrollbar" class, the window itself is the scroll bar! By permitting scroll bars to be created independently of other controls, Windows allows you to define dialogs with scroll bars in other than the standard positions. Type the following to replace the static control icon with a scroll bar:

```
?- wcreate((fred,1),scrollbar,``,10,125,80,64,16'50000004).
                                                          <enter>
```

As with list and combo boxes, scroll bars have no meaningful text parameter, so the empty string has been given. The style bits here mean visible child ("5") and horizontal bottom align ("4"), and the scroll bar is created as shown in *Fig*



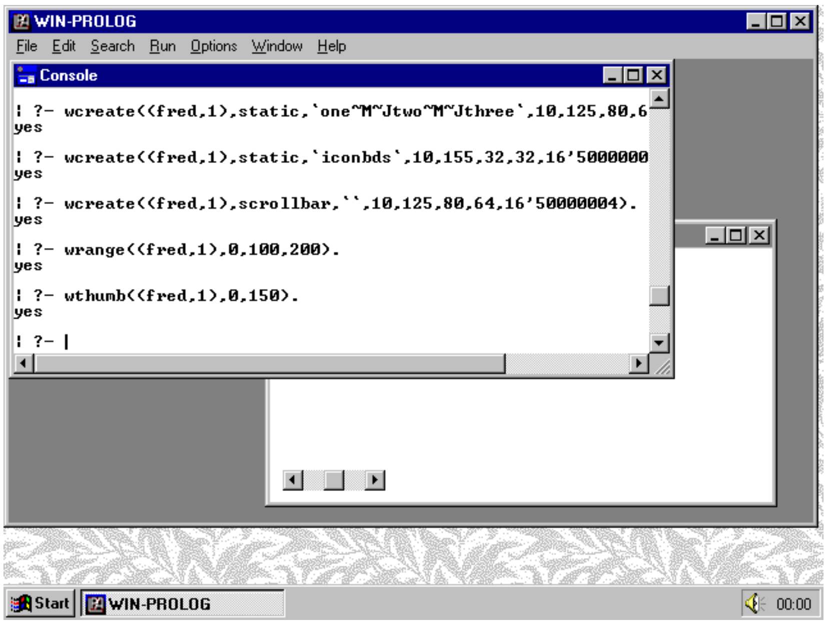
```
| ?- wlbxadd((fred,1),-1,`fox`).
yes

| ?- wtext((fred,1),`Hello`).
yes

| ?- wcreate((fred,1),static,`one^M^Jtwo^M^Jthree`,10,125,80,6
yes

| ?- wcreate((fred,1),static,`iconbds`,10,155,32,32,16'5000000
yes

| ?- wcreate((fred,1),scrollbar,``,10,125,80,64,16'50000004).
yes

| ?-
```

Fig 6.24 - A scrollbar added to the user window

6.24. The least significant digit of the style allows you to position scroll bars of standard width at any of the four sides of the given window rectangle, or to create scroll bars which fill the given area completely.

At this stage, any attempt to move the scroll bar "thumb" will fail, because the scroll bar is created with a "range" of 0..0. Before you can use a scroll bar, you must set its range with the *wrange/4* predicate. Type the command:

?- **wrange((fred,1),0,100,200).** *<enter>*



```
| ?- wcreate((fred,1),static,`one^M^Jtwo^M^Jthree`,10,125,80,6
yes

| ?- wcreate((fred,1),static,`iconbds`,10,155,32,32,16'5000000
yes

| ?- wcreate((fred,1),scrollbar,``,10,125,80,64,16'50000004).
yes

| ?- wrange((fred,1),0,100,200).
yes

| ?- wthumb((fred,1),0,150).
yes

| ?- |
```

Fig 6.25 - Setting the scroll bar range and thumb position

This will set the range of the scroll bar "(fred,1)" to 100..200 inclusive. Now you will find that if you click the scroll buttons or drag the thumb, it will move and remain where you leave it.

Note that unlike other control window predicates, this (and *wthumb/3*, which we are about to look at) are named in the generic window predicate style, and not as *wsbr.../n* or somesuch. This is because both these predicates work not only directly on scroll bar windows themselves (as in the above example), but also on the scroll bar components of other windows. This explains the "0" parameter in the call you have just made. The second argument of both *wrange/4* and *wthumb/3* is an integer which specifies whether the given window is a scroll bar itself ("0"), or whether you want the horizontal ("1") or vertical ("2") scroll bar within the given window.

You can read the position of a scroll bar, or set a new one, using the *wthumb/3* predicate. Type the command:

```
?- wthumb((fred,1),0,150).                        <enter>
```

This will move the thumb of "(fred,1)" to its midway position, as shown in *Fig 6.25*, because you have set a range of 100..200 for this scrollbar. Again, note the "0" parameter, which states that "(fred,1)" is a scroll bar in its own right, and not a window containing a scroll bar. To read a scroll bar, simply leave the third argument as a variable; for example, the call:

```
?- wthumb((fred,1),0,P).                          <enter>
```

will return the result P = 150 unless you have moved the thumb since the last example.

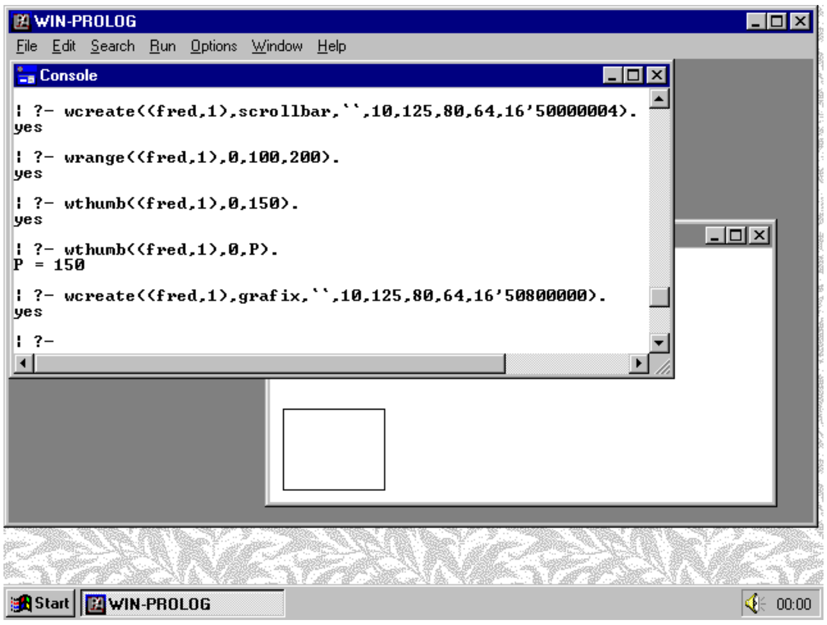
The Grafix Class

This special class differs from all the others described here in that it is defined by, and is unique to **WIN-PROLOG**. By itself, a grafix control looks just like a static control, except that its text parameter will be ignored: as its name suggests, this class provides support for graphics operations. Graphics are discussed in detail in *Chapter 12*, but to have a brief glimpse at the possibilities, type the command:

```
?- wcreate((fred,1),grafix,``,10,125,80,64,16'50800000). <enter>
```

As with list boxes, combo boxes and scroll bars, grafix windows have no meaningful text parameter, so the empty string has been given. The style bits here mean visible child ("5") and border ("8"), and the grafix control is created as shown in *Fig 6.26*. Now type the command:

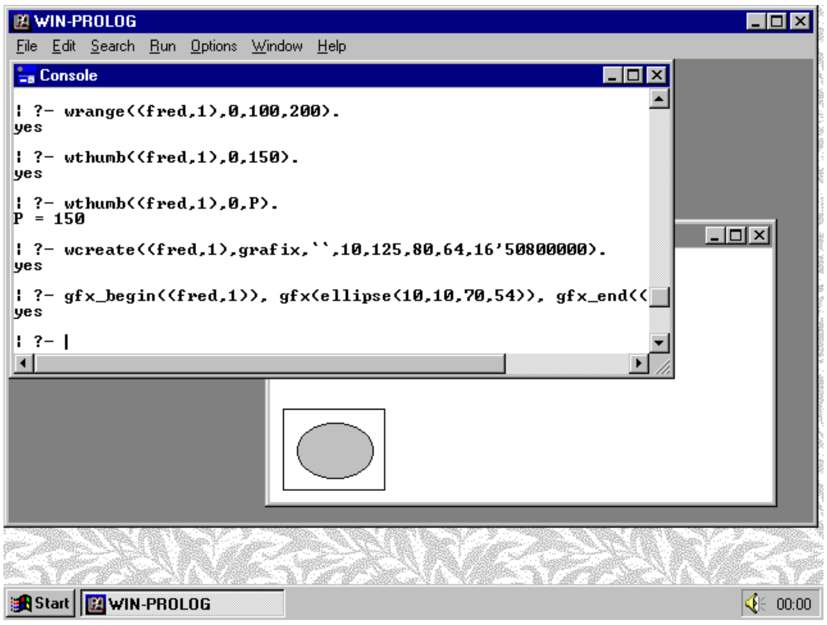
```
?- gfx_begin((fred,1)), gfx(ellipse(10,10,70,54)), gfx_end((fred,1)).
                                                  <enter>
```



```
| ?- wcreate((fred,1),scrollbar,``,10,125,80,64,16'50000004).
yes

| ?- wrange((fred,1),0,100,200).
yes

| ?- wthumb((fred,1),0,150).
yes

| ?- wthumb((fred,1),0,P).
P = 150

| ?- wcreate((fred,1),grafix,``,10,125,80,64,16'50800000).
yes

| ?-
```

Fig 6.26 - A grafix control added to the user window

This call begins graphics on the named window, draws an ellipse, and then finishes the graphics sequence, as shown in *Fig 6.27*. The *gfx/1* predicate is the hub of a family of around 50 predicates which support numerous graphics operations, including the drawing of ellipses, rectangles, polygons, lines, bezier curves, arcs, pie slices and segments, as well as text, metafiles, bitmaps and icons, in full colour. All special operations on grafix controls are carried out by predicates named *gfx*/n*, and together with *gfx/1* itself, these are discussed in detail in *Chapter 12*.



```
| ?- wrange((fred,1),0,100,200).
yes

| ?- wthumb((fred,1),0,150).
yes

| ?- wthumb((fred,1),0,P).
P = 150

| ?- wcreate((fred,1),grafix,``,10,125,80,64,16'50800000).
yes

| ?- gfx_begin((fred,1)), gfx(ellipse(10,10,70,54)), gfx_end((
yes

| ?- |
```

Fig 6.27 - An ellipse drawn in a grafix control

Chapter 7 - Typefaces and Fonts

This chapter describes the use of typefaces and fonts in WIN-PROLOG, covering their creation, uses and housekeeping. These features enable dialogs and other aspects of the user interface to be greatly customised for special effects or personal preference.

Typefaces versus Fonts

Before commencing upon this chapter, we should take a few moments to consider some of the concepts and terminology behind typefaces and fonts. Considerable confusion exists in this area, thanks mainly to the somewhat haphazard way in which type has been introduced to the mass world by Windows in general, and desktop publishing in particular.

So far as this manual is concerned, a "typeface" is the name given to a particular design of type, such as "Times Roman" or "Helvetica". A given typeface is usually part of a "family", where the other members include variations of the basic design at different "weights" (bold, heavy, light, etc) and "style" (italic, roman, etc). It is worth noting that an "italic" version of a typeface is not necessarily just a slanted (oblique) copy of the "roman" (normal) version: indeed, considerable differences normally exist between roman and italic styles.

A "font" is most simply defined as a given typeface, with a given weight and style, at a given size. Thus, "ITC Souvenir-Demi, 14 point" is a font, while "ITC Souvenir" is the generic name of a typeface family, of which "ITC Souvenir-Demi" is one particular typeface design.

Predefined Fonts

Now that the theory has been discussed, let's look at the practical side of things. In *Chapter 6*, we looked at how to create "control" windows, which form the building blocks of dialogs. With the exception of the "grafix" class, all text based controls use, by default, a font called "ANSI Var Font", or more correctly, "Helv 13 pt", which specifies its typeface name and size. For reasons of convenience and brevity, this font is referred to as the "ANSI" font throughout the rest of this manual, because it maps onto the ANSI character set.

By default, the predefined controls in the console and "text" windows, use another font, "Terminal 12 pt", which maps onto the IBM PC extended ("OEM") character set, and which is known as the "OEM" font throughout this manual.

As with all built-in objects, the OEM and ANSI fonts are known to Prolog programs by number rather than by name, with "0" being the OEM font, and "3" the ANSI font.

Creating and Using a Font

The OEM and ANSI fonts are all very well, but Windows supports fully scalable typefaces (TrueType), and comes with several of these as standard. The font handling predicates in WIN-PROLOG allow you to create fonts based on these typefaces, and then apply them to any control window. If you are already in WIN-PROLOG, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start WIN-PROLOG up, and type the following command:

```
?- wfcreate(fred,arial,16,0).                               <enter>
```

In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in *<italics>*. This will create a font known to WIN-PROLOG as "fred", consisting of the "Arial" typeface at a size of 16 point, and roman ("0") or "normal" style. The styles supported by WIN-PROLOG are roman ("0"), italic ("1"), bold ("2") and bold-italic ("3").

At this point, nothing will appear to have happened, apart from Prolog responding with the usual "yes". Now type the command (**bold** letters only):

```
?- wfont((1,1),fred).                                       <enter>
```

After a brief moment, while Windows computes a screen version of Arial 16pt, the console window "(1,1)" will change appearance to use the new font, as shown in *Fig 7.1*. You can switch back to the default, OEM font using the command:

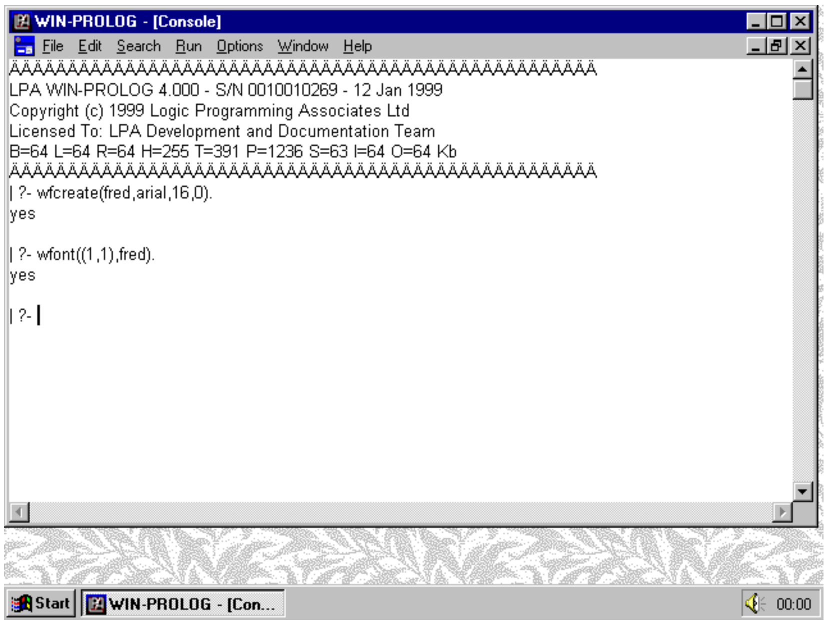
```
?- wfont((1,1),0).                                          <enter>
```

This will cause the console to revert to its normal state, as shown in *Fig 7.2*.

Care in Font Handling

You can create up to 64 named fonts at any one time, and can apply any one of them to any individual control window, creating the potential of a screen design consisting of huge numbers of different fonts. Apart from the stylistic nightmare, which would be reminiscent of the early days of Mac desktop publishing, there are some special considerations to bear in mind when creating and using fonts.

In Windows, fonts are global objects. When your application (WIN-PROLOG) creates a font, it uses up global (system-wide) memory resources, whether or not that font is being used. While a font is in use, such as when it is assigned to a control window, it must not be destroyed. Controls whose fonts have been deleted will not behave properly, and can even cause General Protection Faults.

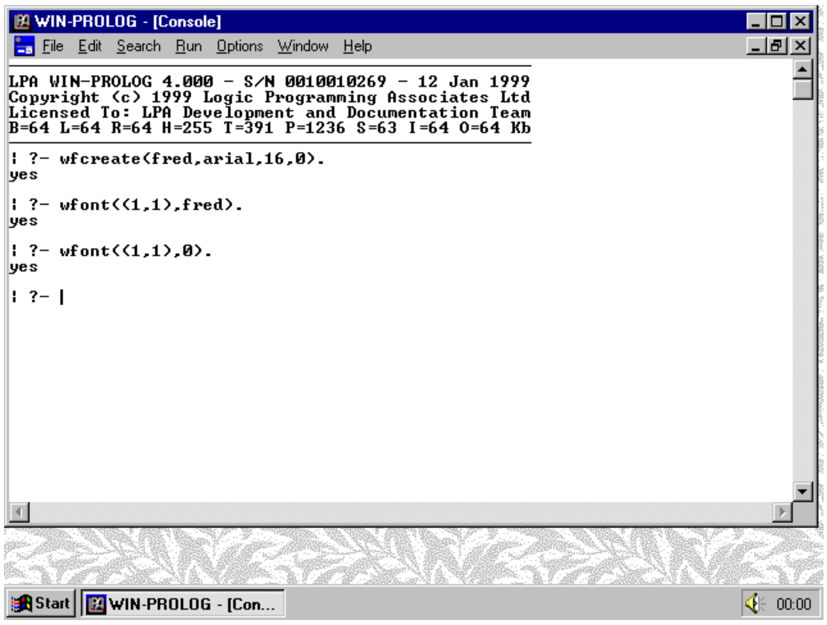


```
ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb
ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ
| ?- wfcreate(fred,arial,16,0).
yes

| ?- wfont((1,1),fred).
yes

| ?- |
```

Fig 7.1 - The console window set to use the Arial 16 font

Because fonts are global, and might have been exported from your application at run time, it is not always safe for **WIN-PROLOG** to destroy your fonts automatically on exit. These resources are not released when your application (**WIN-PROLOG**) terminates via the *exit/1* predicate, and will remain locked in memory until you exit from Windows itself. They are, however, closed down if you terminate via either the *halt/0* or *halt/1* predicate.

If you have created some fonts within **WIN-PROLOG**, and have assigned these to windows which will persist after the current session (such as those belonging



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wfcreate(fred,arial,16,0).
yes

| ?- wfont((1,1),fred).
yes

| ?- wfont((1,1),0).
yes

| ?- |
```

Fig 7.2 - The console window reset to the OEM font "0"

to other applications or DLLs), you should terminate your session with the *exit/1* predicate: do not use *halt/n*, the "File/eXit" menu option, *<alt-f4>* or any other trick to exit from Prolog.

To create a well behaved program, you should not allow fonts created within **WIN-PROLOG** to be assigned to other applications, and should remove these fonts during program termination. In this case, unless you have explicitly closed all fonts, do not use *exit/1* to quit, but use *halt/n*, the "File/eXit" menu option or *<alt-f4>* to complete your session.

Closing a Font

You can close, or destroy, any font that you have created, providing it is not currently in use (see above), by using the *wfclose/1* predicate. So long as the console window is once again using the OEM font, type the command:

```
        ?- wfclose(fred).                                    <enter>
```

This will remove the font "fred" (Arial 16pt) from memory, freeing up its global resources.

Note: if you create a new font with a name that is already being used for a font, the existing definition is first destroyed, to avoid leaving unreferenced fonts in Windows' global memory areas. The same care you take when closing a font must be taken when re-creating one of a given name, since the new one will not automatically be attached to any control that might have been using the previous font with that name.

Enumerating Typefaces

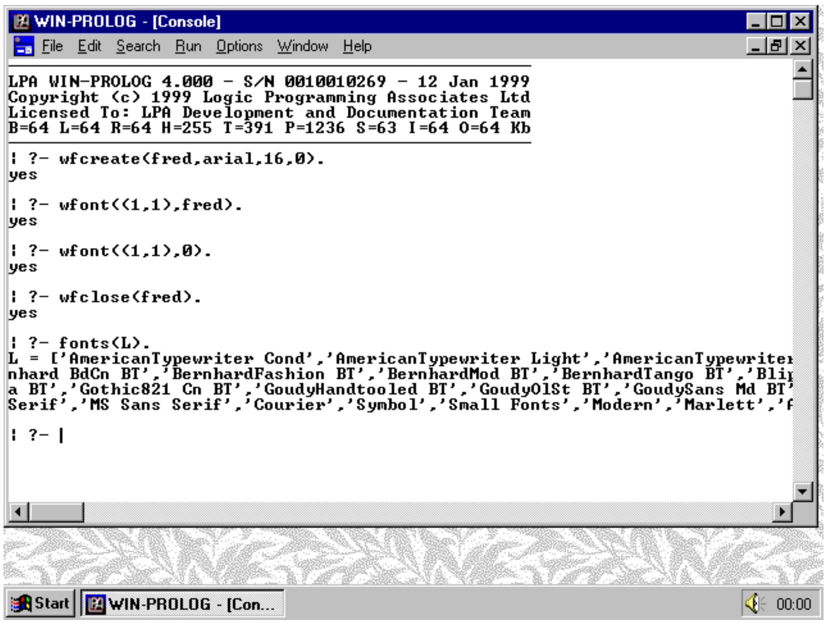
Because of the widespread availability of TrueType typefaces, as well as fonts for third-party type managers such as Adobe's "ATM" and Bitstream's "FaceLift", the typefaces available will vary considerably between individual Windows installations. A special built in predicate, misnamed *fonts/1*, returns the full list of available typefaces. Type the command:

```
        ?- fonts(L).                                         <enter>
```

A list of typeface names will be returned, as shown in *Fig 7.3* (the list will vary depending upon which typefaces are installed in your Windows system). Now prepare to create and show fonts based on some of these faces. Create a window by typing the following command:

```
        ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).   <enter>
```

This will create a user window, ready for experimentation, as shown in *Fig 7.4*. Click back on the console window, and type the following three commands to create three buttons:



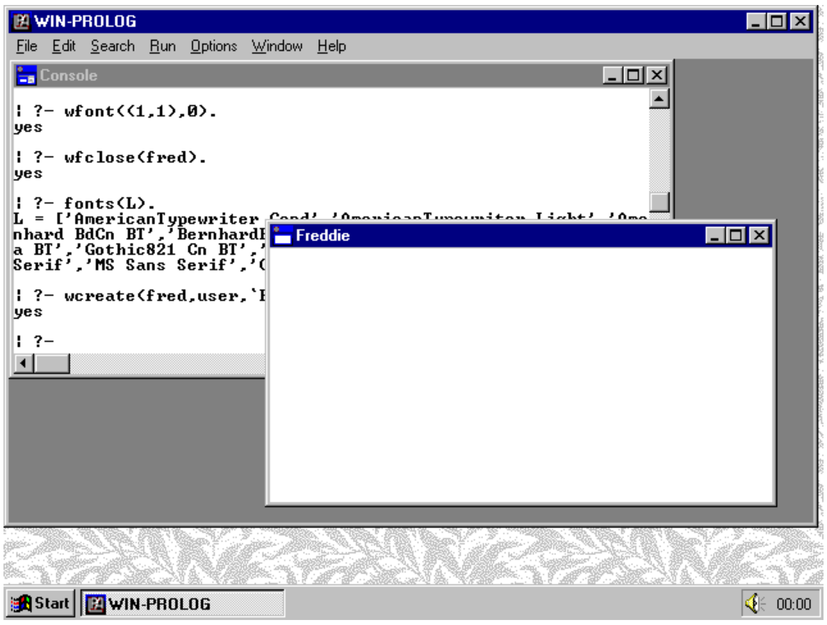
```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wfcreate(fred,arial,16,0).
yes

| ?- wfont((1,1),fred).
yes

| ?- wfont((1,1),0).
yes

| ?- wfclose(fred).
yes

| ?- fonts(L).
L = ['AmericanTypewriter Cond','AmericanTypewriter Light','AmericanTypewriter
nhard BdCn BT','BernhardFashion BT','BernhardMod BT','BernhardTango BT','Blip
a BT','Gothic821 Cn BT','GoudyHandtooled BT','GoudyOlSt BT','GoudySans Md BT'
Serif','MS Sans Serif','Courier','Symbol','Small Fonts','Modern','Marlett','F

| ?- |
```

Fig 7.3 - Returning typefaces with the fonts/1 predicate

?- **wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000).** *<enter>*

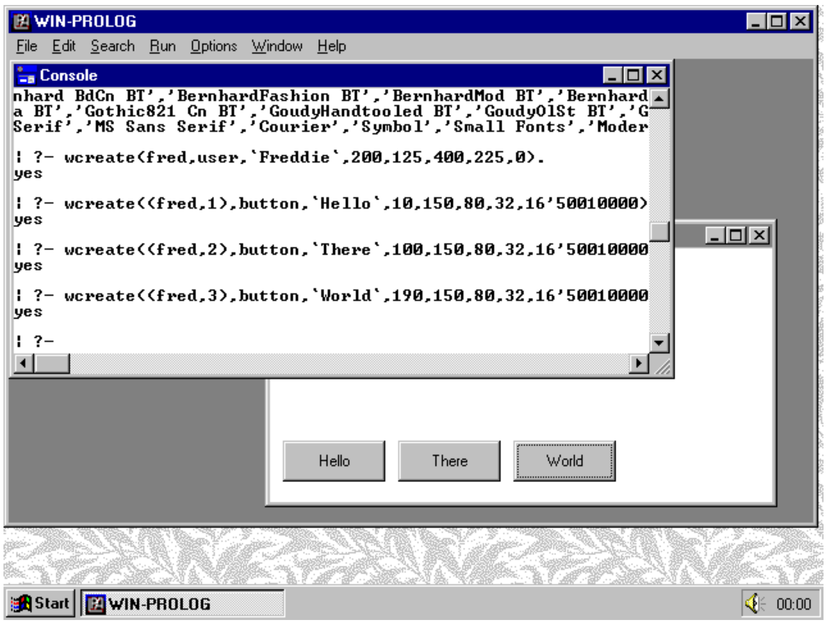
?- **wcreate((fred,2),button,`There`,100,150,80,32,16'50010000).** *<enter>*

?- **wcreate((fred,3),button,`World`,190,150,80,32,16'50010000).** *<enter>*



```
| ?- wfont((1,1),0).
yes

| ?- wfclose(fred).
yes

| ?- fonts(L).
L = ['AmericanTypewriter
nhard BdCn BT','BernhardF
a BT','Gothic821 Cn BT','
Serif','MS Sans Serif','C

| ?- wcreate(fred,user,`F
yes

| ?-
```

Fig 7.4 - A user window ready for experimentation



```
nhard BdCn BT','BernhardFashion BT','BernhardMod BT','Bernhard
a BT','Gothic821 Cn BT','GoudyHandtooled BT','GoudyOlSt BT','G
Serif','MS Sans Serif','Courier','Symbol','Small Fonts','Moder

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010000
yes

| ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010000
yes

| ?-
```

Fig 7.5 - Three buttons added to the user window

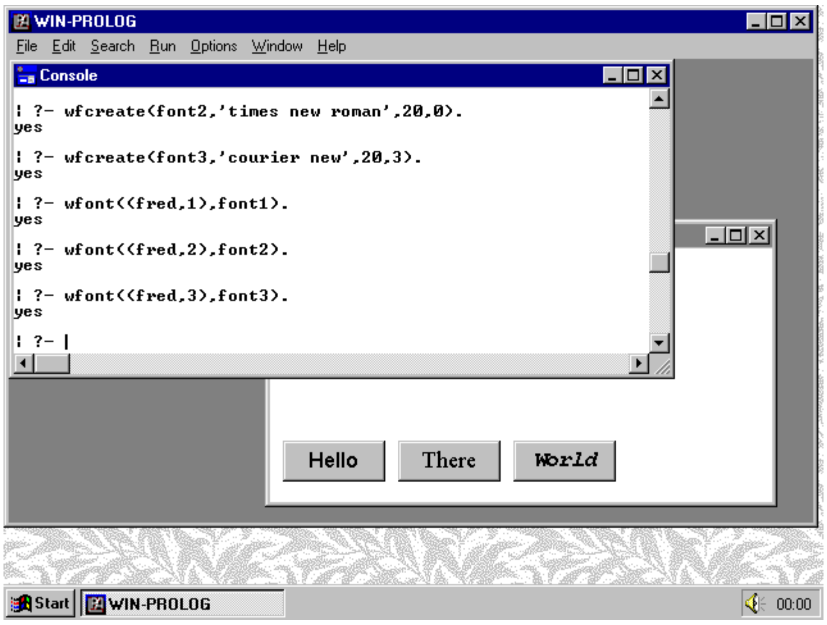
At the end of these three commands, you should have created three buttons in "fred", with IDs of "1", "2" and "3" respectively, as shown in *Fig 7.5*. Now select three typeface names from the list displayed by the *fonts/1* predicate above, and create fonts based on them, for example:

```
?- wfcreate(font1,arial,20,2).                    <enter>

?- wfcreate(font2,'times new roman',20,0).        <enter>

?- wfcreate(font3,'courier new',20,3).            <enter>
```

If the typeface names used in these examples are not present on your system, use names which are. Note that the font names are not case sensitive, so you can use any combination of upper and lower case letters when creating a font, but any embedded spaces must be included literally. Remember to quote the name if beginning with an uppercase letter or if it contains spaces, as in the "font2" and "font3" cases above. Now assign each of the new fonts to one of the buttons in "fred" with the following commands:

```
?- wfont((fred,1),font1).                         <enter>

?- wfont((fred,2),font2).                         <enter>

?- wfont((fred,3),font3).                         <enter>
```

At the end of these three commands, the buttons in "fred" will display with the new fonts, as shown in *Fig 7.6*. If any of the typefaces you requested were unavailable, Windows will supply a font it considers to be equivalent.



```
| ?- wfcreate(font2,'times new roman',20,0).
yes

| ?- wfcreate(font3,'courier new',20,3).
yes

| ?- wfont((fred,1),font1).
yes

| ?- wfont((fred,2),font2).
yes

| ?- wfont((fred,3),font3).
yes

| ?- |
```

Fig 7.6 - Three buttons each set to use a different font

Checking Font Data

You can retrieve the typeface, size and style data for any font that you have previously created using the *wfdata/5* predicate. For example, type the command:

?- **wfdata(0,N,P,S,B).** *<enter>*

This call will return the results "N = 'Terminal'", "P = 12", "S = 0" and "B = 10", to report that the OEM font ("0") is actually called "Terminal 12pt". The final parameter is the "baseline" offset, and is useful in graphics programming. This predicate is very useful for checking that you have created a font successfully. Assuming the font creation example in the previous section worked, the call:

?- **wfdata(font1,N,P,S,B).** *<enter>*

should return the results "N = 'Arial'", "P = 19", "S = 2" and "B = 15". Note that the point size, "19", is actually one less than you asked for: even with scaleable typefaces, Windows cannot always provide an exact match to your requirements. Now create a font based on a typeface which is not on your system, using the command:

?- **wfcreate(fred,loadsarubbish,20,0).** *<enter>*

The Windows call will apparently succeed, but you can test the actual result by typing the following call:

?- **wfdata(fred,N,P,S,B).** *<enter>*

This will return the expected values of P, S and B, but the name will be returned as "N = 'Times New Roman'" or some other default font, showing that Windows could not find "LoadsaRubbish" on the system.

You can also test the print width and height of a line of text using *wfsize/4*. Type the command:

```
?- wfsize(font1,`Hello World`,W,H).                    <enter>
```

This will return the results "W = 92" and "H = 19", to say that in the "Arial Bold 20pt" font ("font1"), the given string has a width of 92 units and a height of 19 units. One "unit" corresponds to one screen pixel, which in turn is roughly equivalent to one point.

Note that, as elsewhere in the WIN-PROLOG's Windows predicates, name parameters (such as the font and typeface name) are given as atoms, and general text parameters as strings (`...`).

The Font Dictionary

A final font handling predicate, *wfdict/1*, lets you return the list of fonts currently defined in WIN-PROLOG. The call:

```
?- wfdict(D).                                          <enter>
```

should return the list "D = [font1,font2,font3,fred]". If you are writing your own clean-up code, to remove fonts directly before quitting from an application, you can use this dictionary to remind you which fonts need closing.

Graphics Programming

Throughout this chapter, we have looked at fonts in connection with control windows: while it is nice to be able to use, say, "Arial 20pt Bold" in the WIN-PROLOG console window, this type of font handling is not particularly flexible. In particular, no one standard Windows control may contain more than a single font at any one time, and you are limited to black-on-white, or whatever system-wide colour combination is currently set for your Windows installation.

In *Chapter 12*, another use of fonts is discussed: a whole set of graphics predicates provides the ability to plot complex shapes, as well as metafiles, bitmaps, icons, and, of course, text strings. Fonts created as described in this chapter can be used, in any combination, and in any colour or mix of colours, to provide fully programmable text output in "grafix" control windows.

Chapter 8 - Menus

This chapter describes the use of menus in WIN-PROLOG, covering their creation, programming and housekeeping. These features enable complete customisation of the main menu bar, including the replacement of existing menus and items as well as the creation of new ones.

Predefined Menus

Despite appearances, there is only one truly predefined menu in WIN-PROLOG: the "Window" menu on the main menu bar. All other menus, including "File", "Edit", etc., are actually created and maintained using standard Prolog code. This means that, apart from "Window", you can create and maintain any number of menus on the menu bar.

The Window menu is programmed in C, and is maintained internally both by WIN-PROLOG and the Windows MDI environment. It is a standard component of a Windows MDI application, so its predefined behaviour should not be a problem in your own programs. Having said that, it is possible to add or remove entries from this menu, or even to remove the menu itself from the menu bar, but the results of doing this can be unpredictable, and this practice is not recommended.

Creating a Menu

Just as with windows, dialogs, controls and fonts, menus are created with a name which will be used thereafter to refer to the menu. If you are already in WIN-PROLOG, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start WIN-PROLOG up, and type the following command:

```
?- wmcreate(fred).                                    <enter>
```

In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in <*italics*>. This will create a menu known to WIN-PROLOG as "fred", and which will initially be empty: no other action will occur.

Adding Items to a Menu

Menus are built up using a series of calls to the *wmnuadd/4* predicate, in a fashion similar to the filling of a list or combo box (see *Chapter 6*). Add an item to your new menu by typing the command (**bold** letters only):

```
?- wmnuadd(fred,-1,`&Freddie`,1000).                  <enter>
```

This adds an item to the menu "fred", at the end ("-1"), using the string "Freddie", and a message code of "1000". You can specify the insertion position for any item you add to any menu, or use "-1" to append to the menu. Position numbering starts at 0, so use this value to place an item at the start of a menu.

As with all general text parameters in **WIN-PROLOG**, the item name is given as a string (`...`); just like the text labels of control windows (see *Chapter 6*), the ampersand ("&") character can be used to precede the character you want to use for keyboard selection within the menu.

The final parameter can be an integer, as shown above ("1000"), or the handle (Prolog name) of another menu: you will see an example of this in a moment; the integer can be any 16-bit value greater than 999, and is used to identify the menu command at run time. Now add two more items by typing:

```
?- wmnuadd(fred,-1,`&Maria`,1001).                <enter>

?- wmnuadd(fred,-1,`&David`,1002).                <enter>
```

So far, nothing has happened (apart from Prolog responding with "yes" prompts). To see (and use) your new menu, you must add it to the main **WIN-PROLOG** menu bar. Since a menu bar is simply a menu in its own right, you use *wmnuadd/4* here too. Type the command:

```
?- wmnuadd(0,3,`Fre&d`,fred).                     <enter>
```

Once again, the menu bar, which is a built-in feature, has a number ("0") rather than a name. Note that the fourth argument is an atom, and not an integer: this is the name of the menu you have been creating ("fred"). At this point, a new item will appear on the menu bar, as shown in *Fig 8.1*.

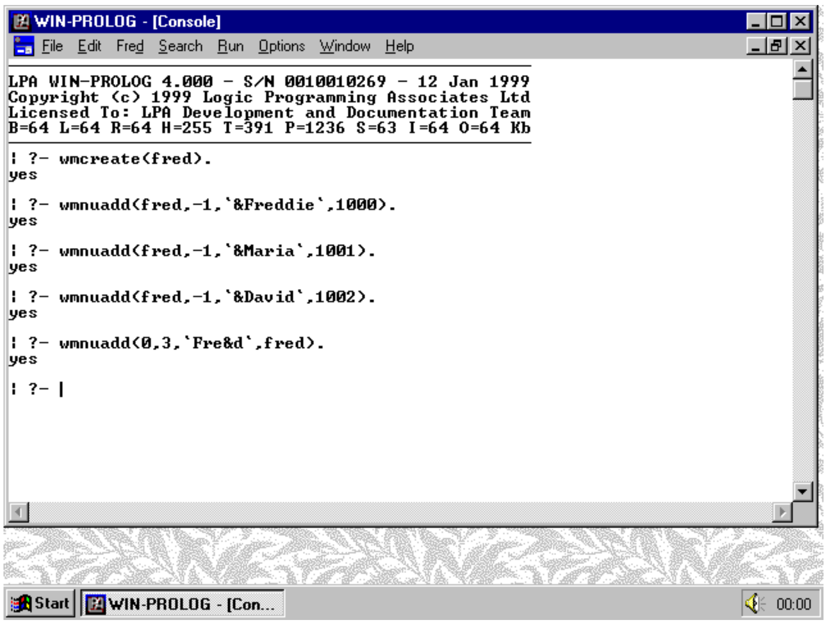
Note the placement of the "&" character in this example: by using "`Fre&d`" rather than "`&Fred`", we have chosen to use "<alt-D>" as the hotkey for this menu, not "<alt-F>", which is already in use by the "File" menu. Click on the "Fred" menu, and it will pull down, as shown in *Fig 8.2*.

Removing Items from a Menu

Just as you add menu items using *wmnuadd/4*, you can remove them using the *wmnudel/2* predicate. To remove your addition to the menu bar, type the command:

```
?- wmnudel(0,3).                                  <enter>
```

Your menu will disappear from the menu bar, as shown in *Fig 8.3*. You have not destroyed the menu "fred" itself, but have simply unhooked it from the menu bar.



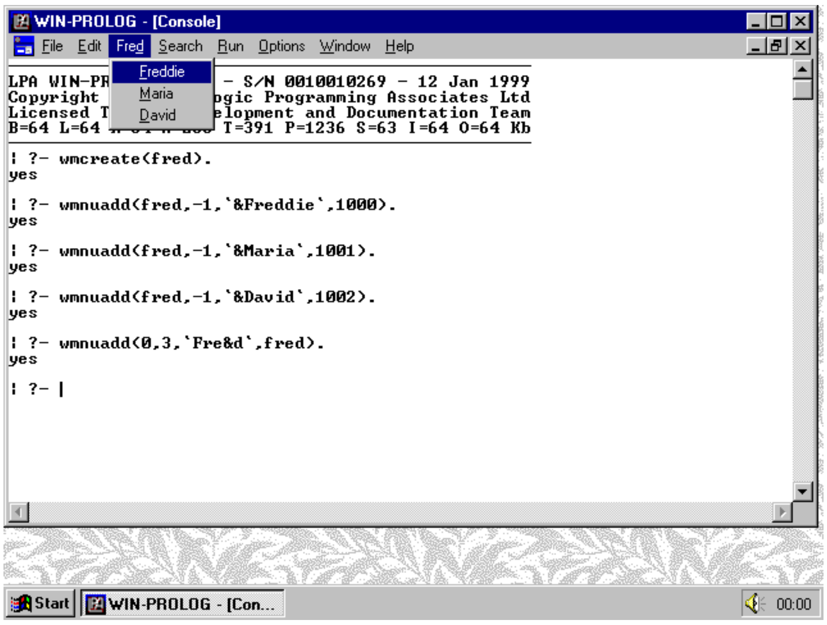
```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wmcreate(fred).
yes

| ?- wmnuadd(fred,-1,`&Freddie`,1000).
yes

| ?- wmnuadd(fred,-1,`&Maria`,1001).
yes

| ?- wmnuadd(fred,-1,`&David`,1002).
yes

| ?- wmnuadd(0,3,`Fre&d`,fred).
yes

| ?- |
```

Fig 8.1 - Creating a menu and adding it to the menu bar

Care in Menu Handling

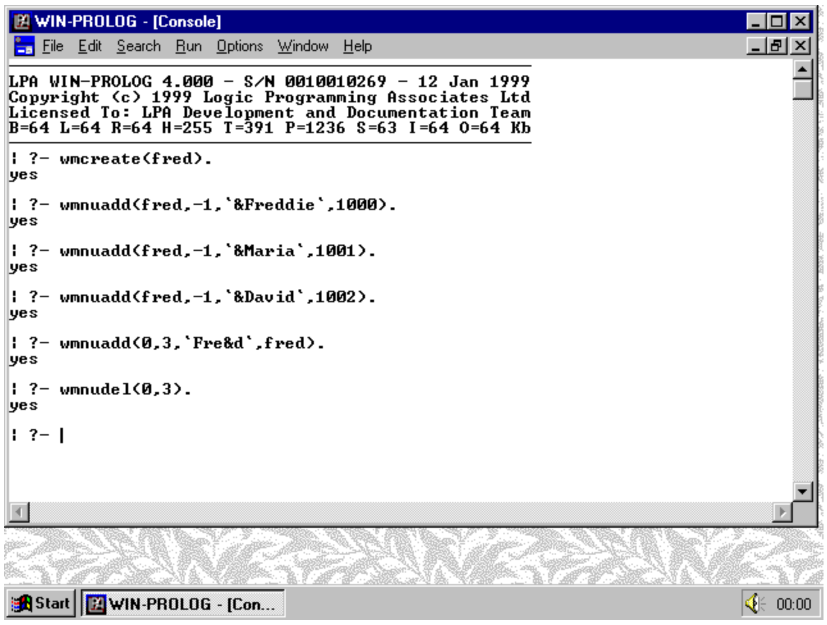
You can create an indefinite number of named menus at any one time, and can attach any of them to any other, creating deeply nested menus. As with fonts (see *Chapter 7*), there are some special considerations to bear in mind when creating and using menus.

In Windows, menus are global objects. When your application (*WIN*-**PROLOG**) creates a menu, it uses up global (system-wide) memory resources, whether or



```
LPA WIN-PR          - S/N 0010010269 - 12 Jan 1999
Copyright          ogic Programming Associates Ltd
Licensed T         elopment and Documentation Team
B=64 L=64           T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wmcreate(fred).
yes

| ?- wmnuadd(fred,-1,`&Freddie`,1000).
yes

| ?- wmnuadd(fred,-1,`&Maria`,1001).
yes

| ?- wmnuadd(fred,-1,`&David`,1002).
yes

| ?- wmnuadd(0,3,`Fre&d`,fred).
yes

| ?- |
```

Fig 8.2 - Pulling down the newly added menu



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wmcreate(fred).
yes

| ?- wmnuadd(fred,-1,`&Freddie`,1000).
yes

| ?- wmnuadd(fred,-1,`&Maria`,1001).
yes

| ?- wmnuadd(fred,-1,`&David`,1002).
yes

| ?- wmnuadd(0,3,`Fre&d`,fred).
yes

| ?- wmnudel(0,3).
yes

| ?- |
```

Fig 8.3 - Removing a menu from the menu bar

not that menu is being used. While a menu is in use, such as when it is attached to the WIN-PROLOG menu bar, it must not be destroyed. Menus whose submenus have been deleted will not behave properly, and can even cause General Protection Faults.

Another problem can arise when defining nested menus: if any circular reference are made, Windows will crash the first time the menu is activated. Menus can be nested quite deeply, but not infinitely so.

If you have created some menus within WIN-PROLOG, and have assigned these to windows which will persist after the current session (such as those belonging to other applications or DLLs), you should terminate your session with the *exit/1* predicate: do not use *halt/n*, the "File/eXit" menu option, *<alt-f4>* or any other trick to exit from Prolog.

To create a well behaved program, you should not allow menus created within WIN-PROLOG to be assigned to other applications, and should remove these menus during program termination. In this case, unless you have explicitly closed all menus, do not use *exit/1* to quit, but use *halt/n*, the "File/eXit" menu option or *<alt-f4>* to complete your session.

Closing a Menu

You can close, or destroy, any menu that you have created, providing it is not currently in use (see above), by using the *wmclose/1* predicate. So long as the menu bar has been restored to its initial state, type the command:

```
?- wmclose(fred).                                    <enter>
```

This will remove the menu "fred" from memory, freeing up its global resources.

Note: if you create a new menu with a name that is already being used for a menu, the existing definition is first destroyed, to avoid leaving unreferenced menus in Windows' global memory areas. The same care you take when closing a menu must be taken when re-creating one of a given name.

Checking Menu Entries

You can retrieve the definition of an entry in any menu that you have previously created using the *wmnuget/4* predicate. Before you can do this, create a couple of menus to experiment with by typing the following commands:

```
?- wmcreate(menu1).                                    <enter>

?- wmcreate(menu2).                                    <enter>
```

These commands will create two empty menus: now add some items to them with the commands:

```
?- wmnuadd(menu1,-1,`&Item1`,1000).                    <enter>

?- wmnuadd(menu2,-1,`&Item2`,2000).                    <enter>

?- wmnuadd(menu1,-1,`&More...`,menu2).                 <enter>
```

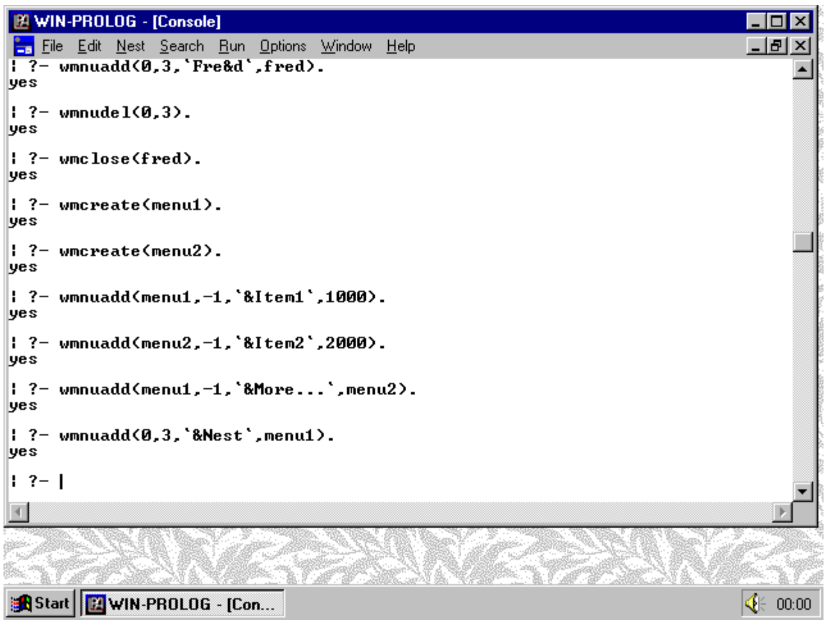
Note how the last command uses a menu handle, "menu2", in place of an integer message number: this links this menu to "menu1", creating a nested menu. Be careful not to add a menu to itself, or to add one menu to another and then the second back to the first: as noted above, any such circular reference can cause Windows to crash. If you have done this by mistake, simply recreate "menu1" and "menu2" following the example above more carefully. Now add this nested menu to the main window menu bar by typing the following command:

```
?- wmnuadd(0,3,`&Nest`,menu1).                         <enter>
```

Your new menu will appear, with "N" as its highlighted hotkey, as shown in *Fig 8.4*. If you click on it with the mouse, you will be able to follow it into the submenu, as shown in *Fig 8.5*. Now that you have checked that your menu is correctly defined, type the command:

```
?- wmnuget(menu1,0,S,I).                               <enter>
```

This will return the values "S = `&Item1`" and "I = 1000", which comprise the string and integer message number of the first entry (number "0") from the menu "menu1". Now type the command:

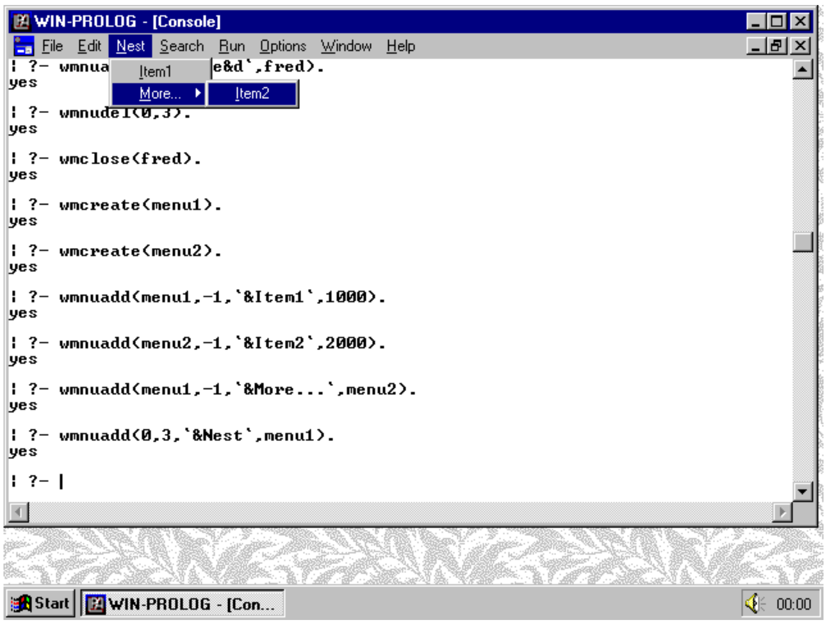


```
| ?- wmnuadd(0,3,`Fre&d`,fred).
yes

| ?- wmnudel(0,3).
yes

| ?- wmclose(fred).
yes

| ?- wmcreate(menu1).
yes

| ?- wmcreate(menu2).
yes

| ?- wmnuadd(menu1,-1,`&Item1`,1000).
yes

| ?- wmnuadd(menu2,-1,`&Item2`,2000).
yes

| ?- wmnuadd(menu1,-1,`&More...`,menu2).
yes

| ?- wmnuadd(0,3,`&Nest`,menu1).
yes
| ?- |
```

Fig 8.4 - Creating and installing a nested menu

?- **wmnuget(menu1,1,S,I).** *<enter>*

This will return the values "S = `&More...`" and "I = menu2", the second entry (number "1") from the menu. Because an atom has been returned in "I", rather than an integer, you can tell that this item is a menu handle, and refers to a submenu. By using *wmnuget/4* recursively on any menu handles that are returned, you can traverse the entire hierarchy of any nested menu.



```
| ?- wmnua         e&d`,fred).
yes

| ?- wmnudel(0,3).
yes

| ?- wmclose(fred).
yes

| ?- wmcreate(menu1).
yes

| ?- wmcreate(menu2).
yes

| ?- wmnuadd(menu1,-1,`&Item1`,1000).
yes

| ?- wmnuadd(menu2,-1,`&Item2`,2000).
yes

| ?- wmnuadd(menu1,-1,`&More...`,menu2).
yes

| ?- wmnuadd(0,3,`&Nest`,menu1).
yes

| ?- |
```

Fig 8.5 - Displaying the newly added nested menu

Separators, Checkmarks and Grey Menu Items

You can add various features to your menus to improve the layout, mark entries as "selected", or "disable" (grey out) inappropriate choices. Type the following command:

```
?- wmnuadd(menu1,1,``,0).                                   <enter>
```

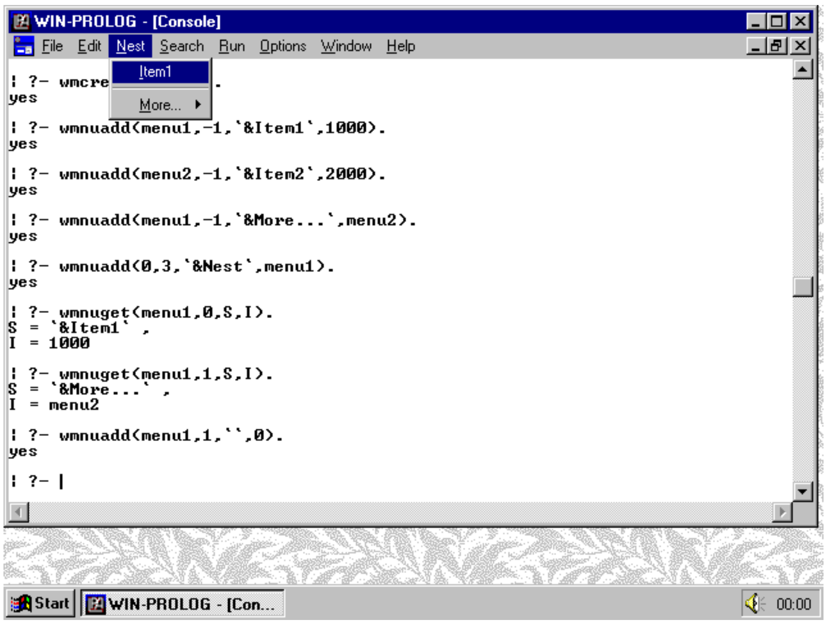
This adds a separator to position "1" on "menu1". Separators are denoted by a message number of zero, and the text parameter is ignored. Click on the "Nest" menu, and you will see the new separator, as shown in *Fig 8.6*. You can put as many separators as you like in a menu, and since you have added them with *wmnuadd/4*, you can delete them with *wmnudel/2*, just like any other menu item.

Checkmarks are the tick characters that are used to indicate the toggle states, and can be added to any non-separator menu item with the *wmnusel/3* predicate. Type the command:

```
?- wmnusel(menu1,0,1).                                      <enter>
```

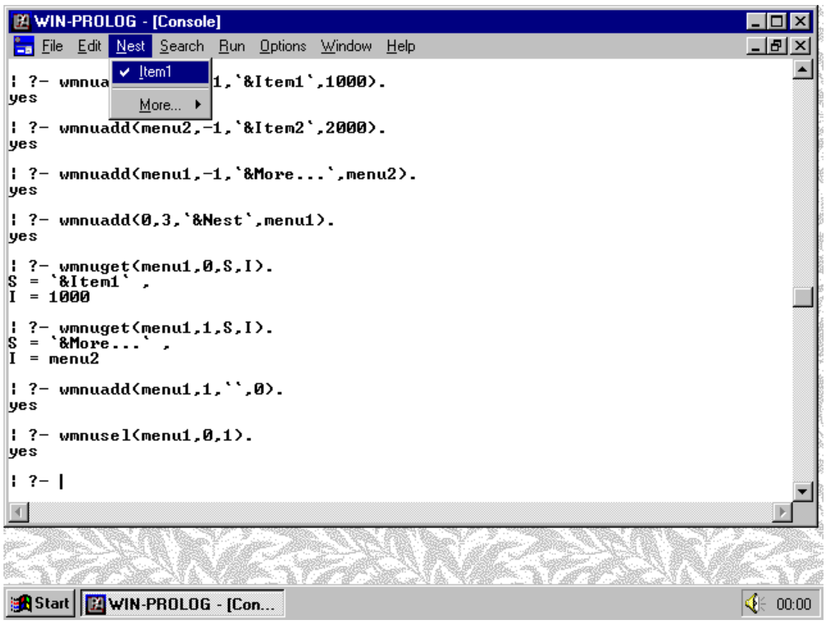
This will set the toggle state of the first item ("0") in "menu1" to "1", or enabled, showing a tick alongside the text string for this item. Click on the "Nest" menu, and you will see the checkmark alongside "Item1", as shown in *Fig 8.7*. You can clear a checkmark by giving "0" as the third argument, or simply return the current setting by calling *wmnusel/3* with a variable as the third argument.

Menu items can be disabled and enabled: when disabled, they are normally "greyed out", indicating that they cannot be used. Type the following command:



```
| ?- wmcre          .
yes

| ?- wmnuadd(menu1,-1,`&Item1`,1000).
yes

| ?- wmnuadd(menu2,-1,`&Item2`,2000).
yes

| ?- wmnuadd(menu1,-1,`&More...`,menu2).
yes

| ?- wmnuadd(0,3,`&Nest`,menu1).
yes

| ?- wmnuget(menu1,0,S,I).
S = `&Item1` ,
I = 1000

| ?- wmnuget(menu1,1,S,I).
S = `&More...` ,
I = menu2

| ?- wmnuadd(menu1,1,``,0).
yes

| ?- |
```

Fig 8.6 - Displaying the menu to show the new separator

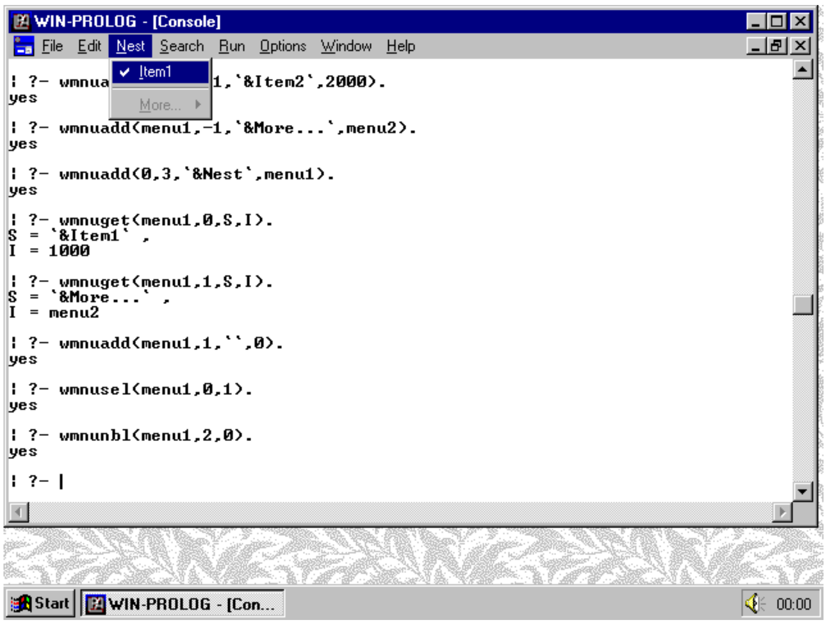


```
| ?- wmnua              1,`&Item1`,1000).
yes

| ?- wmnuadd(menu2,-1,`&Item2`,2000).
yes

| ?- wmnuadd(menu1,-1,`&More...`,menu2).
yes

| ?- wmnuadd(0,3,`&Nest`,menu1).
yes

| ?- wmnuget(menu1,0,S,I).
S = `&Item1` ,
I = 1000

| ?- wmnuget(menu1,1,S,I).
S = `&More...` ,
I = menu2

| ?- wmnuadd(menu1,1,``,0).
yes

| ?- wmnusel(menu1,0,1).
yes

| ?- |
```

Fig 8.7 - Displaying the menu to show a checked item

?- **wmnunbl(menu1,2,0).** *<enter>*

This sets the enable state of the third item ("2") in "menu1" to "0": note that separators count as menu items, so the "More..." submenu, which was the second item in the menu, became the third item when you added a separator. Click on the "Nest" menu, and you will see that the "More..." item has been greyed out, preventing you from selecting it, as shown in *Fig 8.8*. You can reenable a menu by giving "1" as the third argument, and even disable an entry without greying it by giving "2". To return the existing enable state, call *wmnunbl/3* with a variable as the third argument.



```
| ?- wmnua              1,`&Item2`,2000).
yes

| ?- wmnuadd(menu1,-1,`&More...`,menu2).
yes

| ?- wmnuadd(0,3,`&Nest`,menu1).
yes

| ?- wmnuget(menu1,0,S,I).
S = `&Item1` ,
I = 1000

| ?- wmnuget(menu1,1,S,I).
S = `&More...` ,
I = menu2

| ?- wmnuadd(menu1,1,``,0).
yes

| ?- wmnusel(menu1,0,1).
yes

| ?- wmnunbl(menu1,2,0).
yes

| ?- |
```

Fig 8.8 - Displaying the menu to show a greyed item

The Menu Dictionary

A final menu handling predicate, *wmdict/1*, lets you return the list of menus currently defined in **WIN-PROLOG**. The call:

```
?- wmdict(D).                                    <enter>
```

should return the list "D = [menu1,menu2]". If you are writing your own clean-up code, to remove menus directly before quitting from an application, you can use this dictionary to remind you which menus need closing.

Chapter 9 - General Window Handling

This chapter describes the general handling of windows in WIN-PROLOG, covering their handles, linking and styles. These features are applicable to all window types discussed so far, and provide considerable programming power.

Window Handles

Back in *Chapter 3* we looked briefly at window handles, in a section called "Windows and Controls". Throughout the subsequent chapters, use was freely made of window handles, but without ever fully explaining their structure and significance. This chapter, which is all about the low level linking and manipulation of windows, is the natural place in which to examine window handles fully.

Window handles in WIN-PROLOG programs can be in any one of three data types: atoms, integers and conjunctions. When you create a Multiple Document Interface (MDI) or dialog window, you assign to it an atom which is used as its identifier or handle, for example:

```
?- wcreate(fred,...
```

creates a window with the handle "fred". When you create a control window, you use a handle which is a conjunction of the parent window's handle and an integer ID, for example:

```
?- wcreate((fred,123),...
```

creates a control within the named window "fred", using an ID of 123. All windows other than those you have created yourselves have integer handles only: these includeWIN-PROLOG's predefined windows, as well as any windows belonging to other applications. There are two predefined integer handles in WIN-PROLOG, and these are listed in *Table 9.1*.

Table 9.1 - Predefined Window Handles

Integer	Meaning
0	The Main Window
1	The Console Window

The console window ("1") includes a predefined "edit" control, whose ID is also "1". This is the same as the ID used in "text" windows created as part of the MDI (see *Chapter 3*). Just as you would refer to a control of ID "123" in a window called "fred" using the conjunction "(fred,123)", so you should refer to the console window's edit control as "(1,1)".

External Windows

Windows belonging to WIN-PROLOG have names which are either atoms (MDI and dialog windows), integers (predefined windows) or conjunctions (controls). It is also possible to access and manipulate windows belonging to applications other than WIN-PROLOG, and this is done using the "raw" window handle for the window concerned. Raw handles are simply 32-bit integers which are passed directly on to Windows.

You can distinguish between integer handles belonging to external windows and those belonging to WIN-PROLOG's predefined windows by value: all raw window handles have values greater than "1", which is the highest numbered integer handle used for predefined windows.

Note that all external windows, whether top level, part of another application's MDI, or controls, are known simply by a raw integer: the conjunction data type is not used for external controls.

Logical and Raw Handles

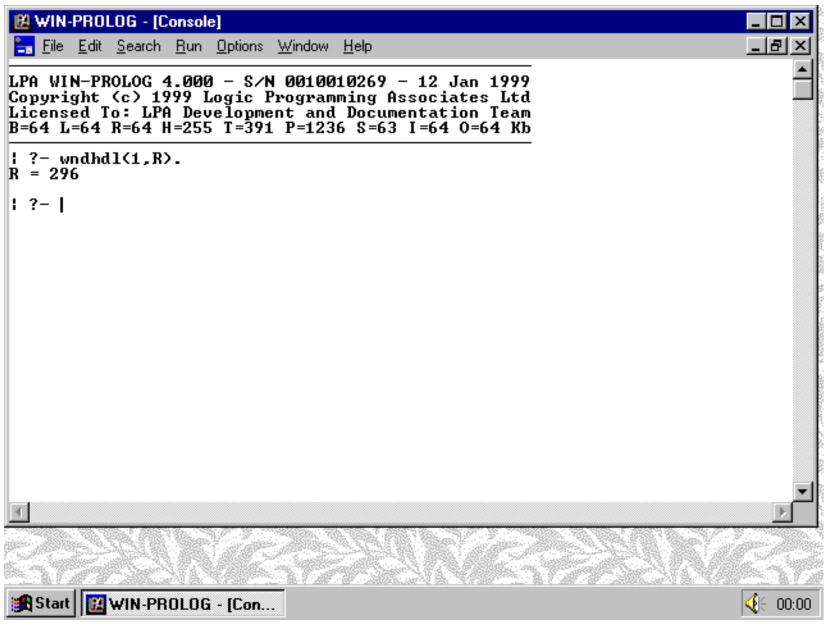
All windows, including those you create, are known to the underlying Windows operating environment by 32-bit, raw handles: the atom and conjunction handles that you assign when you create windows are merely "logical" handles that must be translated internally by WIN-PROLOG before being passed to Windows. Conversely, whenever Windows wants to pass a window handle back to WIN-PROLOG, an attempt is made to convert the raw handle back to its logical counterpart. If this attempt fails, the raw handle is returned directly. You can perform the conversion between logical and raw handles explicitly using the *wndhdl/2* predicate. If you are already in WIN-PROLOG, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start WIN-PROLOG up, and type the following command:

```
?- wndhdl(1,R).                                   <enter>
```

In this, and all other examples, only type the characters in**bold** letters, and press the named key for anything bracketed in <*italics*>. This returns the value "R" bound to an integer which is the raw handle of the console window ("1"), as shown in *Fig 9.1*. You can perform the reverse translation by typing the command (**bold** letters only):

```
?- wndhdl(L,1234).                                <enter>
```

where "1234" is the number that was returned by the first example; this will return the value "L = 1". As was mentioned above, WIN-PROLOG automatically returns logical window handles wherever possible, but there are times when you might want to perform the translation yourself: these include passing window handles to and from the *winapi/4* predicate (see *Appendix G*), where the logical



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wndhdl(1,R).
R = 296

| ?- |
```

Fig 9.1 - Returning the raw handle of the console window

name would be meaningless, as well as some sophisticated multi-nested window programming techniques.

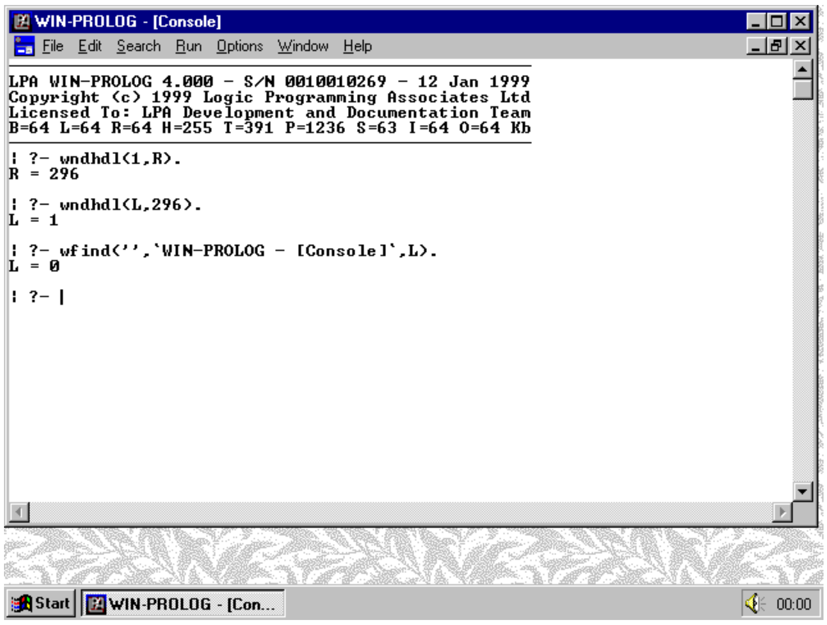
Finding an External Window Handle

There are several ways in which to find the handle of an external window. The first will occur from time to time by accident, using predicates such as *wfocus/1* which return window handles to **WIN-PROLOG**: if this predicate is called while another application is in focus, the handle returned will be the raw handle of the application, MDI window or control currently in focus. In such circumstances, your application will typically not wish to perform any processing on the resultant handle; indeed, in *Appendix C* you will see an example of some code which uses *wfocus/1* to suspend further processing until the user switches focus back to **WIN-PROLOG**.

Occasionally you might wish intentionally to access the handle of another window, and there are two main predicates which allow you to do this. The first, *wfind/3*, allows you to search the desktop for a window of the given name and class. For example, type the command:

```
?- wfind('',`WIN-PROLOG - [Console]`,L).          <enter>
```

This will search for a window labelled "WIN-PROLOG - [Console]", returning the handle "0" in "L", as shown in *Fig 9.2*. This is of course simply the **WIN-PROLOG** main window, but you can use this trick to find windows belonging to any currently running application. The first argument of *wfind/3* is an atom which is the name of the window class: if you don't know this name, you should give an empty atom, as here. The second is simply its title which, in common with all general text parameters, is a string (`...`).



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wndhdl(1,R).
R = 296

| ?- wndhdl(L,296).
L = 1

| ?- wfind('','WIN-PROLOG - [Console]',L).
L = 0

| ?- |
```

Fig 9.2 - Finding an application's top level window

The second window finding predicate, *wlink/3*, allows you to link to other windows from any given window handle. Click back on the console window and type the command:

?- **wlink(0,5,E).** *<enter>*

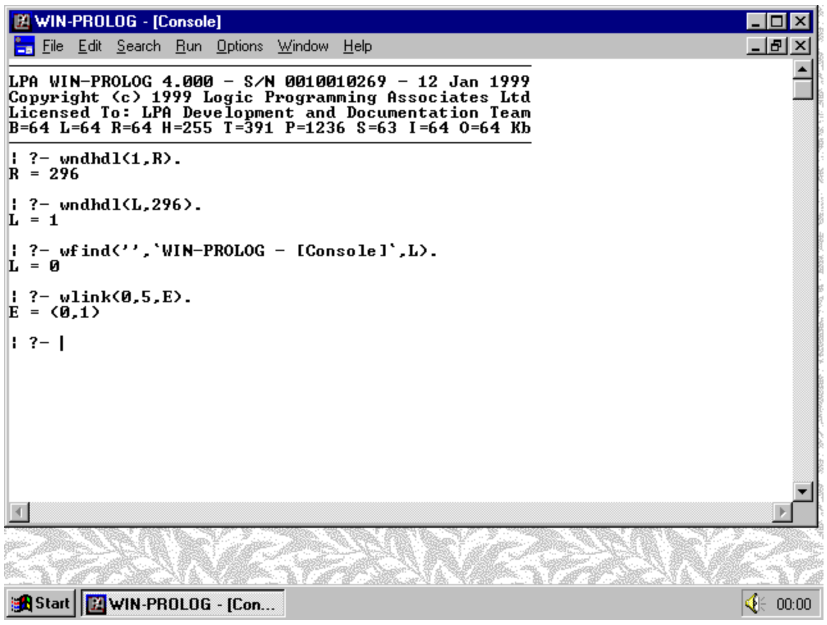
and you will get the handle "(0,1)" returned in "E", as shown in*Fig 9.3*. The value "5" in the second argument of *wlink/3* means "first child of", and what you have now got is the handle of *WIN-PROLOG*'s MDI client window. You can link to the parents, siblings, children and owners of any window, using the values listed in *Table 9.2*.

Table 9.2 - Window Linking Parameters

Integer	Meaning
-1	return parent of window
0	return first sibling of window
1	return last sibling of window
2	return next sibling of window
3	return previous sibling of window
4	return owner of window
5	return first child of window

Window Styles

When creating dialog and button windows, you will have used various 32-bit integer styles to define the exact appearance and behaviour of these windows. You can retrieve the style of an existing window, or even change it, with the



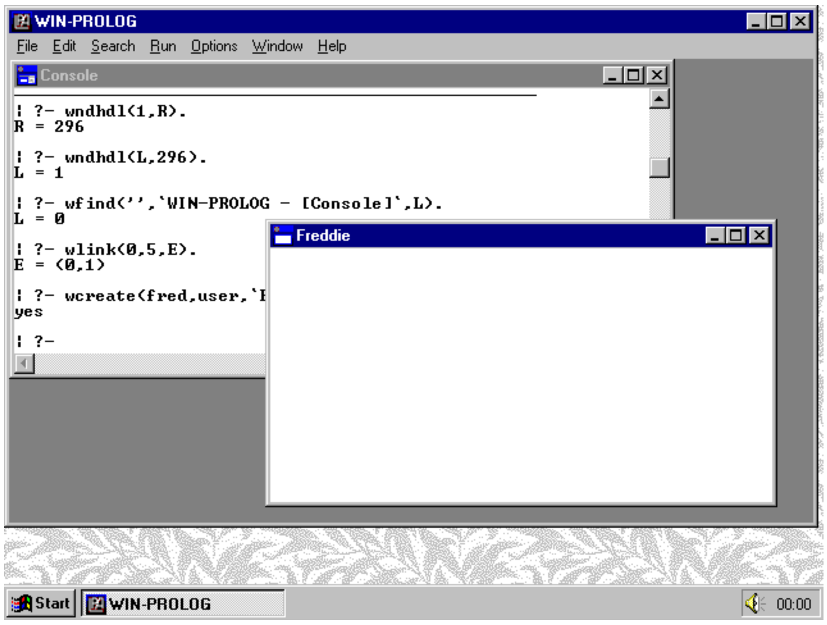
```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wndhdl(1,R).
R = 296

| ?- wndhdl(L,296).
L = 1

| ?- wfind('','WIN-PROLOG - [Console]`,L).
L = 0

| ?- wlink(0,5,E).
E = (0,1)

| ?- |
```

Fig 9.3 - Linking to a child window from a parent window

wstyle/2 predicate. Care should be taken when changing window styles, because many styles are incompatible with certain windows. The most common use of this predicate is to change the appearance of control windows, say from "pushbutton" to "defpushbutton". Click back on the console window and type the following command:

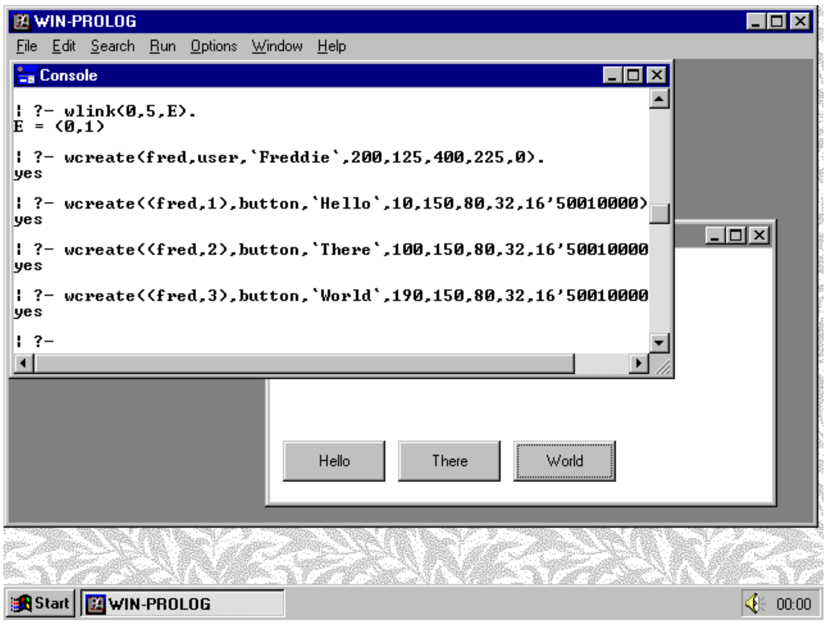
?- **wcreate(fred,user,`Freddie`,200,125,400,225,0).** *<enter>*

This creates a user window, as shown in *Fig 9.4*. Now click back on the console window and add some buttons to it with the following commands:



```
| ?- wndhdl(1,R).
R = 296

| ?- wndhdl(L,296).
L = 1

| ?- wfind('','WIN-PROLOG - [Console]`,L).
L = 0

| ?- wlink(0,5,E).
E = (0,1)

| ?- wcreate(fred,user,`F
yes

| ?-
```

Fig 9.4 - A user window read for experimentation



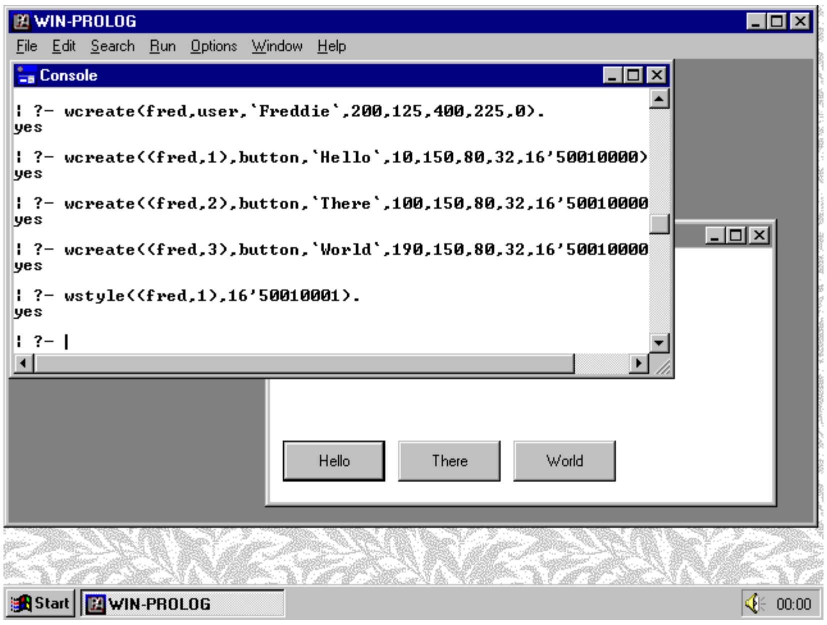
```
| ?- wlink(0,5,E).
E = (0,1)

| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010000
yes

| ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010000
yes

| ?-
```

Fig 9.5 - Three buttons added to the user window

?- **wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000).** *<enter>*

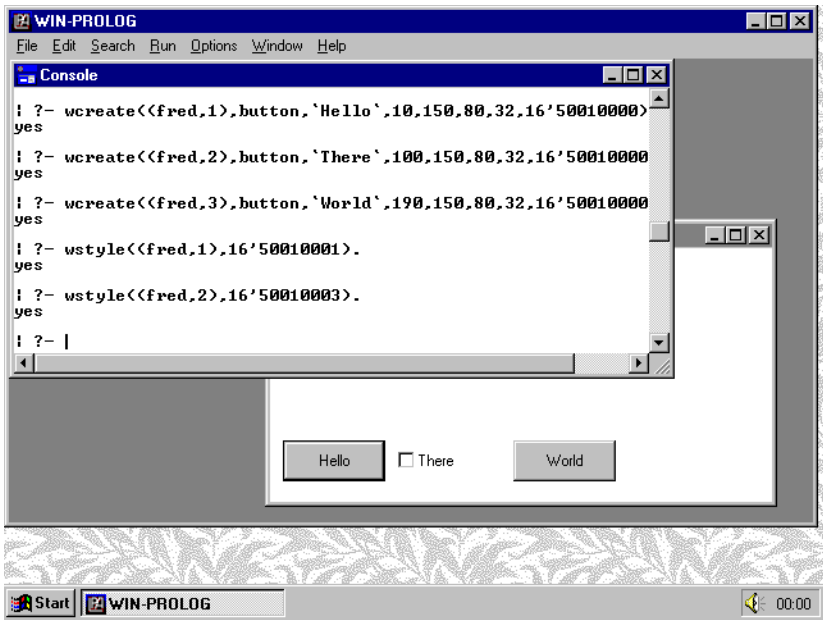
?- **wcreate((fred,2),button,`There`,100,150,80,32,16'50010000).** *<enter>*

?- **wcreate((fred,3),button,`World`,190,150,80,32,16'50010000).** *<enter>*



```
| ?- wcreate(fred,user,`Freddie`,200,125,400,225,0).
yes

| ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

| ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010000
yes

| ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010000
yes

| ?- wstyle((fred,1),16'50010001).
yes

| ?- |
```

Fig 9.6 - Changing a button style with wstyle/2



```
¦ ?- wcreate((fred,1),button,`Hello`,10,150,80,32,16'50010000)
yes

¦ ?- wcreate((fred,2),button,`There`,100,150,80,32,16'50010000
yes

¦ ?- wcreate((fred,3),button,`World`,190,150,80,32,16'50010000
yes

¦ ?- wstyle((fred,1),16'50010001).
yes

¦ ?- wstyle((fred,2),16'50010003).
yes

¦ ?- |
```

Fig 9.7 - Changing a button type with wstyle/2

These commands will have added three buttons to the user window "fred", as shown in *Fig 9.5*. Now change the appearance of the first button from that of an ordinary "pushbutton" to that of a default "defpushbutton" by typing:

```
?- wstyle((fred,1),16'50010001).                    <enter>
```

The first button, "(fred,1)", will be redrawn with a thicker, "default" border, as shown in *Fig 9.6*. Now type the command:

```
?- wstyle((fred,2),16'50010003).                    <enter>
```

This will change the appearance and behaviour of the second button, "(fred,2)", to an "autocheckbox", as shown in *Fig 9.7*. It is not recommended to perform changes of this kind, it being far safer to create a new control of the desired new type. Uses of *wstyle/2* should be limited to "borrowing" the a style of an existing window for use when creating a similar new one, or toggling buttons between the "pushbutton" and "defpushbutton" styles.

Window Size and Area

In *Chapter 5* you saw the *wsize/5* predicate, which is used to test or set the size of a window: there is a related predicate, *warea/5*, which returns the size and position of a window's active, or "client" area. The overall size of a window (as handled by *wsize/5*) includes the space taken up by its menus, scroll bars and borders. The client area is the portion of the window which is available for controls and other objects. Type the command:

```
?- warea(0,X,Y,DX,DY),
wcreate(fred,button,`BIG`,X,Y,DX,DY,16'90010000).   <enter>
```

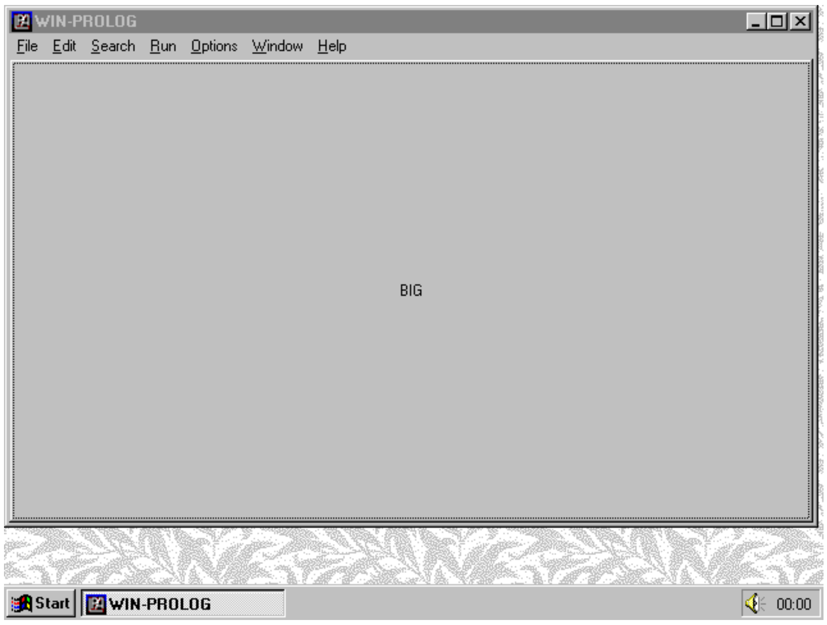


Fig 9.8 - A giant button window on the desktop

This will pick up the size and position of the main window's client area, and create a button of this exact size. Note that because the button has been given an atom ("fred") as its handle, and that the style includes the popup rather than child bit ("9" rather than "5"), it is created as a top level window in its own right, as shown in *Fig 9.8*. Although not very useful, this particular example does help impress that controls are windows just like any other. Click back on the main window and type the command:

```
?- wclose(fred).                                    <enter>
```

to close the giant desktop button.

Window Classes

Every window created in the Windows environment belongs to one or another class. When you create windows, you specify the class as the second argument to *wcreate/8*, for example:

```
?- wcreate((fred,123),edit,...
```

creates a window of the "edit" class. You can find out the class of any given window with the *wclass/2* predicate. For example, type the command:

```
?- wclass(0,C).                                     <enter>
```

This will return the result "C = 'Main'", giving the class name for the **WIN-PROLOG** main window ("0"). This is one of several classes defined internally by **WIN-PROLOG**, which also include the console window ("Cons") and of course "Grafix" controls.

Chapter 10 - Windows Messages

This chapter describes the handling of Windows messages by WIN-PROLOG, covering the programming of responses to menus, modal and modeless dialogs.

Windows and Messages

As an operating system, Windows depends on message passing to perform virtually all functions. Windows messages are simply packets of information which may be passed between processes, containing commands, parameters, or notification of actions that are about to be or have already been performed.

Most messages are of a very basic, low-level nature, and would be of little or no interest to the Prolog programmer. For example, if the mouse is moved across the screen, literally hundreds of messages are generated. Some just report each change of mouse position, others notify windows that their space has been encroached upon by the mouse, and yet others alert of changes in cursor shape, window borders, and so forth: even a simple keystroke generates around half a dozen distinct messages.

Prolog and Messages

It would not make sense to try to process all messages in WIN-PROLOG: for one point, a great many messages are simple notifications or acknowledgements needed by low level systems code, and all Prolog would want to do is ignore them and pass them on; for another, the sheer number of messages swilling about in Windows would knock the performance of WIN-PROLOG flying if every one caused a Prolog-level interrupt!

It is equally clear that some messages must be passed on to WIN-PROLOG, otherwise it would not be possible for Prolog programs to react to certain user actions. Messages of this kind work by interrupting the execution of a Prolog program, transferring control to a message handler, which is itself a Prolog program. As soon as a message occurs, further messages are disabled while the message handler performs whatever operations are necessary. When it is finished, the message handler reenables the message queue and returns control to the original program.

Messages passed to Prolog fall into several groups, including menu selections, modal and modeless dialogs, window state changes and graphics events.

Anatomy of a Message

A message is a combination of four pieces of information: the first of these is a window handle identifying the source of the message, the second is a 16-bit integer indicating the message number or type, the third is a 32-bit integer containing supplementary data, and the fourth is the Prolog goal that was interrupted by the message. You will be using all four components of the message when programming the various hooks and handlers described in later sections.

In general, the numbering and meaning of messages is entirely up to the programmer, but there are four groups of predefined message numbers, and these are outlined in the following sections.

Predefined Modeless Dialog Messages

The first group of predefined messages is generated by the two system modeless dialogs, namely the "find" and "change" boxes; these messages are numbered between 90..98, and each message refers to one of the action buttons in the dialog. Depending upon whether the dialog is in its "find first" or "find next" state (see *Chapter 4*), the "Find" button returns one of two different messages. The modeless dialog messages are listed in *Table 10.1*.

Table 10.1 - Messages Generated by Modeless Dialogs

Prolog Atom	Integer	Meaning
MSG_FBCLOSE	90	The Find Box close button
MSG_FBFIND	91	The Find Box fiNd button
MSG_FBFNDNXT	92	The Find Box fiNd next button
MSG_CBCLOSE	93	The Change Box close button
MSG_CBFIND	94	The Change Box fiNd button
MSG_CBFNDNXT	95	The Change Box fiNd next button
MSG_CBCHANGE	96	The Change Box chanGe button
MSG_CBCHGFND	97	The Change Box changE+find button
MSG_CBCHGALL	98	The Change Box change aLl button

When actions are requested in either the find or change box, the message refers to the MDI edit window currently being worked on as its source, and not the find or change box dialog.

Window Menu and State Messages

The second group of messages provides information about menu choices and changes in the state of windows. Whenever you select a menu item, switch focus between windows, or change the contents of an edit window, one of these messages is generated. Similarly, if you click a button or change the selection in a list box, a message is generated to tell WIN-PROLOG what you have done. The window state messages are listed in *Table 10.2*.

Table 10.2 - Messages Generated by Menus and Changes in Window State

Prolog Atom	Integer	Meaning
MSG_MENU	00 (M)	Menu item selected
MSG_SYSMENU	01 (M)	System menu item selected
MSG_CLOSE	02 (X)	Window has been requested to close
MSG_FOCUS	03 (F)	Window has come into focus
MSG_FUZZY	04 (F)	Window has gone out of focus
MSG_CHANGE	05 (T)	Edit control has been changed
MSG_BUTTON	06	Button has been clicked
MSG_SELECT	07	List or Combo box selection made
MSG_DOUBLE	08	List or Combo box double clicked
MSG_SIZE	09 (L)	Window has been sized
MSG_MOVE	10 (L)	Window has been moved
MSG_HORZ	11 (S)	Horizontal scroll bar been moved
MSG_VERT	12 (S)	Vertical scroll bar been moved
MSG_DROP	22 (D)	A collection of files have been dropped

When a menu is selected, or a window's state changes, the message refers directly to the window or control which generated the message. Most of these messages include additional information in the 32-bit integer: the two menu messages, marked (M), return the menu item ID of the selected entry; the close message, marked (X), returns a value if 0 if the window has been requested to close, and 1 if a request has been made to terminate the current Windows session altogether; the two focus message, marked (F), return the raw handle of the window which lost or gained focus; the change message, marked (T), returns a value of 0, 1 or 2 to indicate a selection change, actual change or failed change respectively; the two size and move messages, marked (L), return the new position or size; the two scroll messages, marked (S), return the new scroll position in the supplementary data argument, and the drop message, marked (D), returns a drop handle.

Predefined and User Menu Messages

The two menu messages, marked (M) in *Table 10.2*, are processed differently for "predefined" and "user" menu messages. The former group comprises those messages generated by the menus belonging to the menu bar in the **WIN-PROLOG** programming environment. These messages are numbered so that 100-199 constitute the "File" messages, 200-299 the "Edit" messages, and 300-399 the "Search" messages, and so on. All values less than 1000 are reserved for use by the system, and so should be considered "predefined": these messages are not sent to window handlers (see below). You should use menu ID codes of between 1000 and 63999 for user menu items, to ensure that your window handler will receive the messages.

When selections are made from the main window menu bar or system menu, the message refers to the top MDI window as its source, and not the main window itself; when, however, selections are made from a dialog's menu bar or system menu, the message refers to the dialog window directly.

Graphics Messages

The third group of messages provides information about changes in the graphics state of windows. Whenever you expose part of a "grafix" control, so that it needs to be repainted, or move the mouse across such a window, messages are generated to help graphics programming in WIN-PROLOG. The window graphics messages are listed in *Table 10.3*.

Table 10.3 - Messages Generated by Graphics Events

Prolog Atom	Integer	Meaning
MSG_PAINT	13 (B)	Window needs painting
MSG_LEFTDOWN	14 (P)	Left mouse button pressed
MSG_LEFTDOUBLE	15 (P)	Left mouse button double clicked
MSG_LEFTUP	16 (P)	Left mouse button released
MSG_RIGHTDOWN	17 (P)	Right mouse button pressed
MSG_RIGHTDOUBLE	18 (P)	Right mouse button double clicked
MSG_RIGHTUP	19 (P)	Right mouse button released
MSG_MOUSEMOVE	20 (P)	Mouse moved to new position
MSG_CHAR	21 (C)	Character returned from keyboard

Most of these messages will only occur in graphics windows, although there are two exceptions. The paint message, marked (B), returns one of three values in the supplementary integer: "0" indicates that a "grafix" window needs repainting, while "1" and "2" indicate that the messages originate from a button in its "button up" or "button down" state respectively. The seven mouse messages, marked (P), return the mouse position encoded in the 32-bit integer, and only originate from "grafix" windows. The final message, marked (C), returns the character code of any keypress made while a "grafix" window is in focus, while in "edit" windows it returns the character code of any unprocessed control character.

Message Hooks and Subclassing

When a message is received by WIN-PROLOG, it is initially analysed and sorted into one of several classes of message described above; once its class has been established, suitable operations are performed in response to the message. For example, when the "MSG_FBCLOSE" message is received, WIN-PROLOG first establishes that this is a message from the "find" box dialog, and then closes the dialog.

Messages from each source are processed by separate "hook" programs, and each of these hooks can be subclassed, or taken over, by suitably written user programs. Programs which subclass messages from predefined sources are called "user hooks", and generally have the form:

```
'?CLASS?'(Window,Message,Data,Goal) :-
        ...
```

where "CLASS" is an uppercase name identifying the message class, "Window" is a window handle identifying the message source, "Message" is the message itself, "Data" is the supplementary data item, and "Goal" is the Prolog goal whose execution was replaced by the message.

For each of the user hooks, there is a corresponding built-in hook function that performs default processing, and which you can call if you want **WIN-PROLOG** to take control back. Each built-in hook is simply a predicate of the form:

```
class_hook(Window,Message,Data,Goal) :-
```

where "class" is the lowercase version of the corresponding "CLASS" name. A user hook definition which performs no special operations at all could be written:

```
'?CLASS?'(Window,Message,Data,Goal) :-
        class_hook(Window,Message,Data,Goal).
```

This would simply pass all messages of the given type directly to the built-in hook. Such a definition is unnecessary, however, since **WIN-PROLOG** will call the default hook automatically if a given user hook is not defined.

Message Preprocessing

Most predefined messages (in fact, all those originating from sources other than the menus) are preprocessed before being handed over to user hooks. The preprocessing consists of two steps: firstly, the message number is replaced with a lowercase atom naming the message, and secondly, the supplementary data parameter is decoded where necessary to make it easier to use. Thus the message containing the elements:

```
(foo,1), 16, 8061384, bar
```

will be translated to:

```
(foo,1), msg_leftup, (456,123), bar
```

In all hook programs (except *'?MESSAGE?'/4*, see below), you should use the symbolic names of predefined messages. The data for messages marked (F) in *Tables 10.2* and *10.3* return a window handle; those marked (T) return a value of "0" if the only operation was a change of selection, "1" if an actual change has been successful, or "2" if the attempted change resulted in truncation, and the data parameter of messages marked (S) simply returns the integer scroll position, and those marked (L) return a pair of integers. These messages, as well as those which are unmarked, do not perform any translation of the data item.

The remaining marked messages do perform various transformations, as outlined below.

The data for messages marked (B) are translated from the integers "0", "1" and "2" to the atoms "grafix", "button_up" and "button_down" respectively. Similarly, the data for messages marked "C" return either the character code of a printable character, or one of the following names for assorted cursor and edit keys: "prior", "next", "end", "home", "left", "up", "right", "down", "select", "print", "execute", "snapshot", "insert" and "delete".

Window Handlers

As well as being able to define hooks for predefined messages, you can define handlers for messages generated from your own dialogs and menus. Like the predefined message hooks, there is a default handler which processes any messages that your application does not wish to handle. Window handlers have exactly the same status with respect to your dialogs and menus as the built-in hooks have for the predefined dialogs and menus.

After, or even before you create a window or a dialog, you can specify the name of the handler program for messages to that window or dialog. Each window or dialog may have its own handler, or you can share a given handler between several windows and dialog: the choice is yours. You define the relationship between a window and its handler with the *window_handler/2* predicate. For example, the call:

```
?- window_handler(foo,bar).                         <enter>
```

would register your predicate, *bar/4*, as the handler for the window or dialog called "foo". If you create a window or dialog without specifying a handler, then the built-in handler, *window_handler/4*, is used automatically.

'?FIND?'/3: The Find Box Hook

Whenever the "Find" box dialog is displayed, either as a result of selecting the "Search/Find" option from the main window menu bar, or by calling the*fndbox/2* predicate, messages are generated when the user clicks its main button or requests the box to go away. If you write a definition for*'?FIND?'/3*, you can pick up and interpret find box messages; if you want the system to process any particular message, you should return it to **WIN-PROLOG** via the *find_hook/3* predicate. For example, if you wanted to ignore "MSG_FBFNDNXT" messages, you could write the following program:

```
'?FIND?'(Window,msg_fbfndnxt,Goal) :-
     !,
     fndbox(``,1),
     flag(1),
   Goal.
```

```
'?FIND?'(Window,Message,Goal) :-
      find_hook(Window,Message,Goal).
```

The first clause matches with the "MSG_FBFNDNXT" message, and simply reenables the find box; the second clause returns other messages to WIN-PROLOG for further processing. The "Window" parameter returns the handle of the edit control that was last in focus before the find box. The find box hook does not include a data parameter, because this would always be set to zero.

Note that it is your responsibility to reenable messages at the end of a hook by calling the *flag/1* predicate, and then to call the goal that was passed in. If you fail to do the former, you will prevent further messages being processed; if you omit to call the goal, you will effectively cause it to succeed without binding any of its variables, which could cause obvious problems. Similarly, if your hook fails, it will be as if the goal failed, and whatever query was running will be forced to backtrack. Note also that because the find box is automatically disabled whenever its button is clicked, it is necessary to reenable it with a call to *fndbox/2* whenever you want to be able to process further messages.

'?CHANGE?'/3: The Change Box Hook

Whenever the "Change" box dialog is displayed, either as a result of selecting the "Search/Change" option from the main window menu bar, or by calling the *chgbox/3* predicate, messages are generated when the user clicks its main button or requests the box to go away. If you write a definition for *'?CHANGE?'/3*, you can pick up and interpret change box messages; if you want the system to process any particular message, you should return it to WIN-PROLOG via the *change_hook/3* predicate. For example, if you wanted to display a message box in response to the "MSG_CBCLOSE" message, you could write the following program:

```
'?CHANGE?'(Window,msg_cbclose,Goal) :-
      !,
      msgbox('WIN-PROLOG','Closing Change Box',0,_),
      chgbox(``,-1),
      flag(1),
      Goal.

'?CHANGE?'(Window,Message,Goal) :-
      change_hook(Window,Message,Goal).
```

The first clause matches on the "MSG_CBCLOSE" message, and displays a message box before closing the change box; the second clause simply returns other messages to WIN-PROLOG for further processing. The "Window" parameter returns the handle of the edit control that was last in focus before the change box. The change box hook does not include a data parameter, because this would always be set to zero.

Again, note that you are responsible for reenabling the message stream and calling the interrupted "Goal" if you wish Prolog to proceed unhindered. Note also that because the change box is automatically disabled whenever its button is clicked, it is necessary to reenable it with a call to*chgbox/3* whenever you want to be able to process further messages.

Modal Dialog Message Handlers

Before you call a modal dialog using the *call_dialog/2* predicate, you should specify the name of a program that will handle the messages for that dialog by calling *window_handler/2*. The name must refer to a predicate with an arity of 4, the arguments of which are the window name, message number, supplementary data item and original input variable. For example, if you have defined a dialog called "fred", you could call it modally with the command:

```
?- window_handler(fred,foo), call_dialog(fred,X).          <enter>
```

This would display the dialog "fred", and direct all messages from its controls to a program called*foo/4*, suspending all other program execution until the named handler returns a binding to the variable "X".

Suppose "fred" contained just two controls, a button of ID "1" labelled "Display", and a button of ID "2" labelled "Close": you could have written such a program as follows:

```
foo((fred,1),msg_button,_,_) :-
      !,
      write('display button clicked'),
      nl.

foo((fred,1),msg_double,_,_) :-
      !,
      write('display button double clicked'),
      nl.

foo((fred,2),msg_button,_,done).
```

The first two clauses would display suitable messages whenever the "Display" button ("(fred,1)") was clicked or double clicked, while the third would terminate the dialog when the "Close" button ("(fred,2)") was clicked. Modal dialogs are closed by binding the fourth argument to a non-variable value, such as the atom "done" in this case. This binding will be returned as the binding to "X" in the call to *call_dialog/2* above.

Note that unlike hooks, user handlers do not pass in an interrupted goal, so you do not have to make any special provisions to ensure that **WIN-PROLOG** can continue executing.

Modeless Dialog Message Handlers

As with modal dialogs (see above), before you call a modeless dialog with the *show_dialog/1* predicate, you should specify a message handler for that dialog by calling *window_handler/2*. Once again, the name must refer to a predicate with an arity of 4, the arguments of which are the window name, message number, supplementary data item, and binding. For example, if you have defined a dialog called "fred", you could call it modelessly with the command:

```
?- window_handler(fred,foo), show_dialog(fred).          <enter>
```

This would display the dialog "fred", and direct all messages from its controls to a program called *foo/4*, but would immediately return control to WIN-PROLOG.

There is no fundamental difference between modal and modeless dialog handlers, and modeless dialogs are also closed by binding the fourth argument to a non-variable value; however, only modal dialogs can return this value to the calling program. The definition for *foo/4* above will handle modeless as well as modal dialogs.

Generic Window Handlers

All top-level windows, not just modal and modeless dialogs, can be assigned a handler, and all handlers take the same form. As we saw above, there need not be any differences between modal and modeless dialog handlers, and the same is true between window handlers and dialog handlers. In fact, dialogs are simply special types of window which process certain keystrokes (like <*tab*> and <*enter*>) specially.

You can, if you want, create windows, modal and modeless dialogs without specifying a handler: there is a built-in generic window handler, called *window_handler/4* which processes important messages itself. In particular, whenever a button is clicked, it binds its fourth argument to an atom containing the normalised (lower case, with any hotkey "&" character removed) name of the button. This is sufficient to close (or rather, hide) whichever top level window or dialog contained the button. If all you want from a dialog is the name of the button pressed, with the option of subsequently reading some of the other fields, you do not need to define a handler for the dialog.

'?MESSAGE?'/4: The Message Hook

We have just looked at a number of different hooks and handlers, each of which deals with a particular class of messages. While simple to program and use, these hooks and handlers preclude certain operations: for example, you could not intercept messages from the predefined environment menus, or write a program which simply logged all messages of all types.

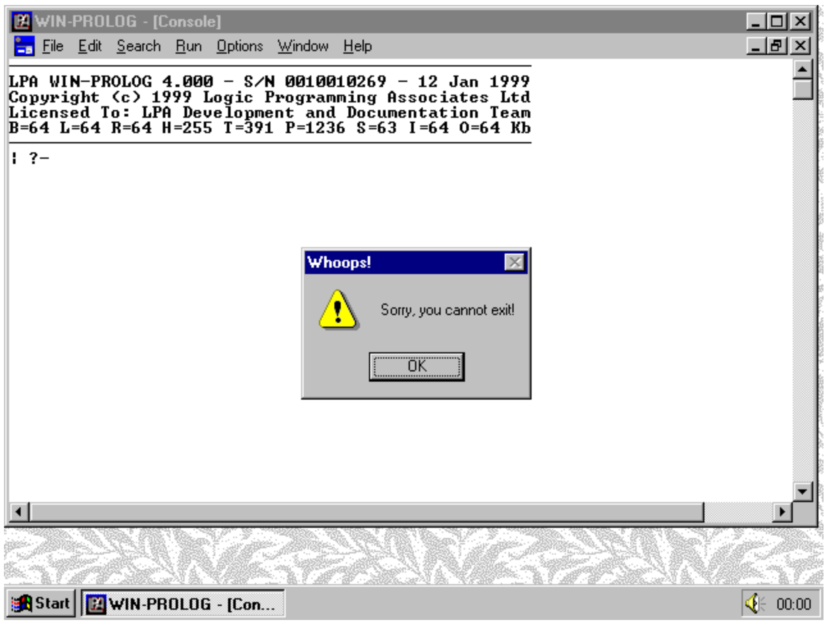

```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?-
```

Fig 10.1 - A message box displayed by the message hook

Normally, all messages are initially handled internally by ***WIN-PROLOG***, which sorts them into appropriate categories before translating the message numbers into symbolic names, preprocessing the data items, and passing the results to one of the various hooks and handlers described in the previous sections. If, however, you want to process messages yourself before ***WIN-PROLOG*** has had a chance to see them, you can do so by writing a definition of the "hook" predicate, *'?MESSAGE?'/4*. The arguments to this predicate consist of the window handle, message number, raw data item and interrupted goal respectively.

Any messages that you do not wish to handle should be passed on to the *message_hook/4* predicate for handling by ***WIN-PROLOG***; if you do decide to handle the message, it is up to you to reenable the message handler and process the goal. The following example shows a definition of the message hook which prevents the "File/eXit" mechanism menu item from terminating a ***WIN-PROLOG*** session:

```
'?MESSAGE?'(_,131,_,Goal) :-
      !,
      msgbox('Whoops!', 'Sorry, you cannot exit!', 48, _),
      flag(1),
      Goal.

'?MESSAGE?'(Window,Message,Data,Goal) :-
      message_hook(Window,Message,Data,Goal).
```

Whenever the "File/eXit" message (131) is received, a message box is displayed on the screen saying that the system does not want you to leave, as shown in *Fig 10.1*. Your only way round this is either to delete the definition of *'?MESSAGE?'/4*, or to exit via the *halt/0* or *halt/1* predicates.

Any messages can be intercepted in this way, except those used internally by the "Edit" and "Window" menus. If you do not pass the message to*message_hook/4*, then **WIN-PROLOG** will not react to the message. You can, of course, pass the message on after you have performed some preliminary operations: for example, you might want to ensure that all windows were recompiled before allowing "Run/Query" to begin a new query.

Note that all messages and supplementary data items are given as integers: the message hook is called directly a message is received, and before any processing has been done on the message. The assignment of names to messages and translation of data items for some of the other hooks and handlers is performed by *message_hook/4*.

Chapter 11 - Creating and Using Dialogs

This chapter describes the creation and use of modal and modeless dialogs by WIN-PROLOG, bringing together an assortment of topics seen in earlier chapters, and covering some new aspects of Windows programming.

Dialog and Control Windows

In *Chapter 5*, we looked at how to create "dialog" windows, in *Chapter 6* we saw how to create "control" windows, and in *Chapter 10* we examined messages, hooks and handlers. A dialog consists minimally of a dialog window containing a single control window, plus some code to handle the messages generated by that control: the three chapters just mentioned provide all the necessary building blocks; in this chapter, we will apply these concepts to produce some simple example dialogs.

Building a Dialog

For now, we will ignore the subtleties of modal and modeless dialog design, and build a generic dialog which can be used both in a modal and modeless fashion. The first step is to define the dialog window. If you are already in WIN-PROLOG, exit back to Windows by selecting the "eXit" option from the "Files" menu, or by typing the "halt." command. Next, start WIN-PROLOG up, and type the following command:

```
?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'80c80000).
                                                      <enter>
```

In this, and all other examples, only type the characters in **bold** letters, and press the named key for anything bracketed in <*italics*>. This will create an invisible dialog window (the first "8" in the style parameter does not include the visibility bit): normally, dialogs are created and their controls added while invisible, just as here. Now add some buttons to the dialog by typing (**bold** letters only):

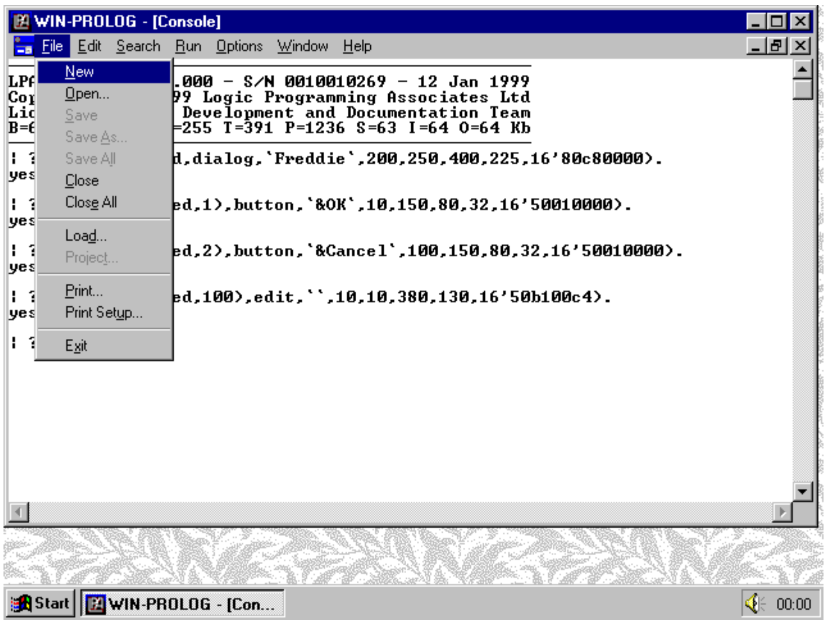
```
?- wcreate((fred,1),button,`&OK`,10,150,80,32,16'50010000).
                                                      <enter>

?- wcreate((fred,2),button,`&Cancel`,100,150,80,32,16'50010000).
                                                      <enter>

?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
                                                      <enter>
```

These commands will create a pair of buttons labelled "OK" and "Cancel", with IDs of "1" and "2" respectively, and a multi-line "edit" control with scroll bars, a border, automatic scrolling, and an ID of "100" (see *Appendix D* for a full explanation of window styles). Although the controls have been created "visible" (the "5" digit in the style), you will still not see anything, since the dialog "fred" is itself still invisible.

Writing a Modal Dialog Handler

In order to use your dialog, you must write a handler program to process its messages. Normally, a modal dialog handler simply terminates the dialog when one of its buttons is clicked, although, as with any WIN-PROLOG hook or handler, it can perform literally any operation it likes. Select "File/New" from the main menu, as shown in *Fig 11.1*, to create a new program window, and enter the following program (omitting the comments if you like):

```
% when the OK button is clicked, return the text in the edit window

fred_handler( (fred,1), msg_button, _, Text ) :-
      wtext( (fred,100), Text ).

% when the Cancel button is clicked, return the atom "cancel"

fred_handler( (fred,2), msg_button, _, cancel ).

% when an edit is made, give a short beep for each change

fred_handler( (fred,100), msg_change, _, _) :-
      beep( 440, 32 ).
```



```
LPA          .000 - S/N 0010010269 - 12 Jan 1999
Co           99 Logic Programming Associates Ltd
Lic           Development and Documentation Team
B=6          =255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?          d,dialog,`Freddie`,200,250,400,225,16'80c80000).
yes

| ?          ed,1),button,`&OK`,10,150,80,32,16'50010000).
yes

| ?          ed,2),button,`&Cancel`,100,150,80,32,16'50010000).
yes

| ?          ed,100),edit,``,10,10,380,130,16'50b100c4).
yes

| ?
```

Fig 11.1 - Creating a new program window

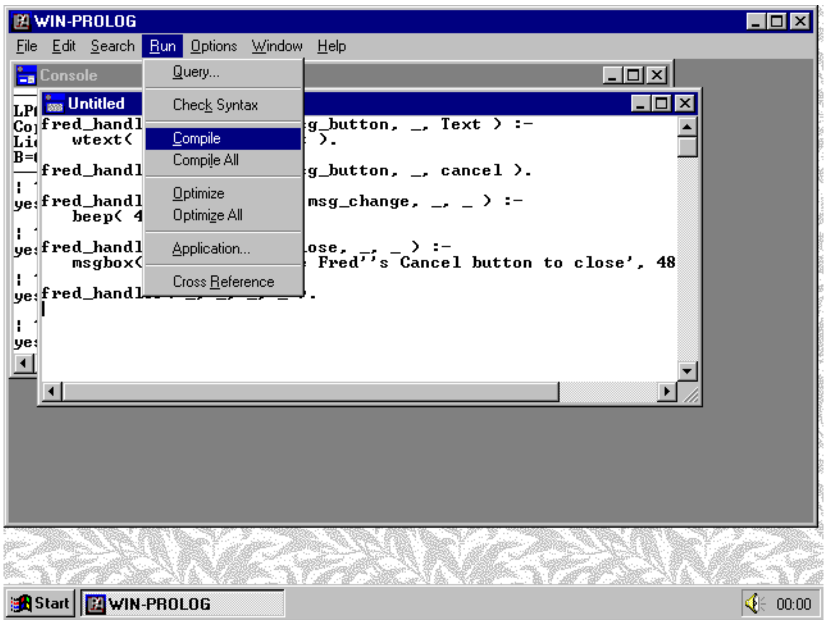
```
% when the close icon is double clicked, show a message box

fred_handler( fred, msg_close, _, _ ) :-
      msgbox( 'Whoops!', 'Use Fred''s Cancel button to close', 48, _ ).

% ignore all other messages

fred_handler( _, _, _, _ ).
```

Modal dialog handlers terminate the dialog by binding the input (fourth) argument, and in this example, both the buttons (but not the close icon) do this: the first clause matches when you click the "OK" button "(fred,1)", binding the input argument to the entire text contents of the edit control "(fred,100)", and the second matches when you click the "Cancel" button "(fred,2)", binding the input to the atom "cancel".

The remaining clauses do not bind the input argument, and so do not terminate the dialog: the third clause is just a bit of fun, and beeps briefly for every character you type into the edit control "(fred,100)", the fourth clause displays a helpful message box whenever you use the "close" icon, and the fifth simply ignores all other messages. Choose "Run/Compile" from the main menu to compile your program, as shown in *Fig 11.2*, correcting any errors and recompiling as necessary.

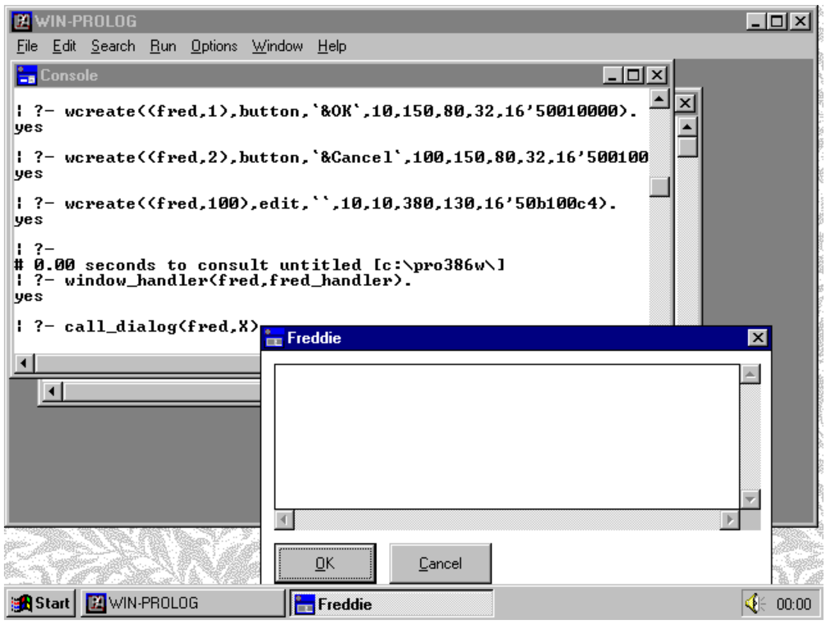
Calling the Modal Dialog

The three components, namely a dialog window, some controls, and handler code, are all now in place. Before you can use your handler with your dialog,



```
fred_handl                    g_button, _, Text ) :-
    wtext(                    ).

fred_handl                    g_button, _, cancel ).

fred_handl                    msg_change, _, _ ) :-
    beep( 4

fred_handl                    ose, _, _ ) :-
    msgbox(                   Fred''s Cancel button to close', 48

fred_handl                    .
```

Fig 11.2 - Compiling the modal dialog handler



```
¦ ?- wcreate((fred,1),button,`&OK`,10,150,80,32,16'50010000).
yes

¦ ?- wcreate((fred,2),button,`&Cancel`,100,150,80,32,16'500100
yes

¦ ?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
yes

¦ ?-
# 0.00 seconds to consult untitled [c:\pro386w\]
¦ ?- window_handler(fred,fred_handler).
yes

¦ ?- call_dialog(fred,X)
```

Fig 11.3 - Calling a modal dialog with call_dialog/2

you must register their association; click back on the console window, and type the command:

?- **window_handler(fred,fred_handler).** *<enter>*

This will store an internal record which says that the program *fred_handler/4* will handle messages for the dialog "fred". Note that the arity of your handler is not given: all window and dialog handlers have an arity of four. Now type the command:

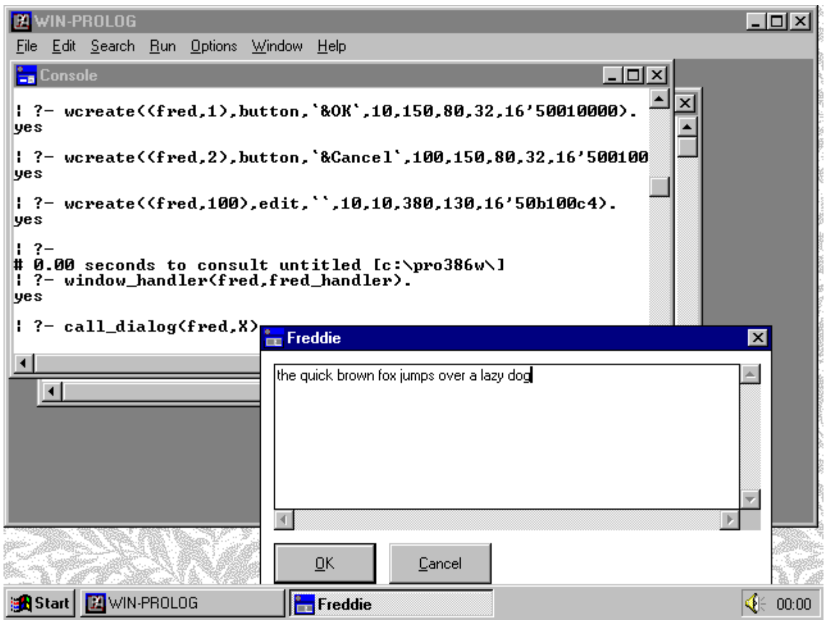
?- **call_dialog(fred,X).** *<enter>*

The two arguments include the name of the dialog to be called ("fred"), and a variable to return the result. Your dialog will be displayed, as shown in *Fig 11.3*.

Click on the edit control, and type some input, as shown in *Fig 11.4*, and notice how each character you type is accompanied by a short, medium-pitched beep: this shows you that your handler is being called for every event that takes place in the dialog.

Now try closing "fred" by double-clicking the "close" icon at the top left corner of the dialog, and see how this simply displays a message box, as shown in *Fig 11.5*. Click on "OK" to close the message box, and then on "OK" within your dialog. The dialog will now vanish, and the contents of the edit control will be returned bound to the variable "X", as shown in *Fig 11.6*.

Note that your dialog is still present: *call_dialog/2* simply shows a predefined dialog when you first call it, and hides it again on completion. This means that you only ever need to define a dialog once per session, but can call it as often as you like. To see this, once again type the command:



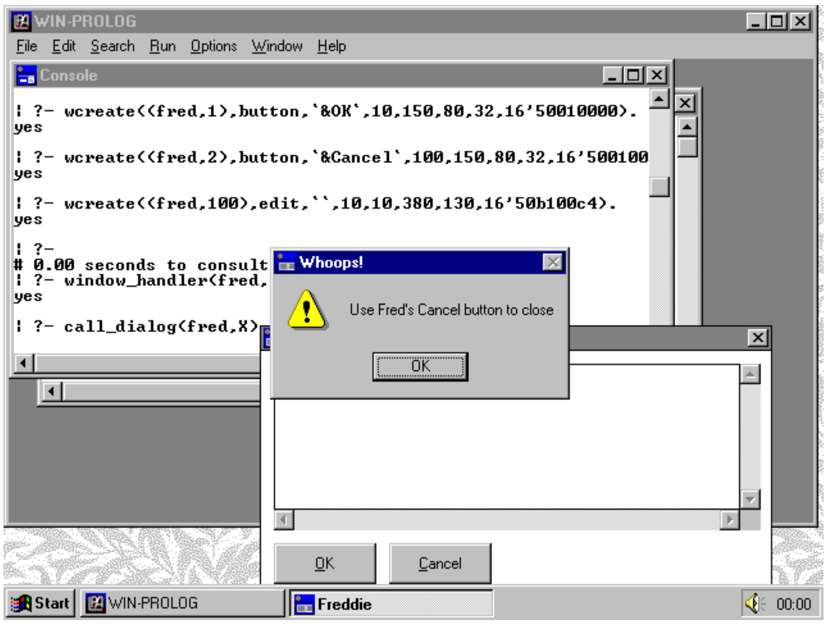
```
| ?- wcreate((fred,1),button,`&OK`,10,150,80,32,16'50010000).
yes

| ?- wcreate((fred,2),button,`&Cancel`,100,150,80,32,16'500100
yes

| ?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
yes

| ?-
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- window_handler(fred,fred_handler).
yes

| ?- call_dialog(fred,X)
```

```
the quick brown fox jumps over a lazy dog
```

Fig 11.4 - Typing text into the modal dialog edit control

?- **call_dialog(fred,X).** *<enter>*

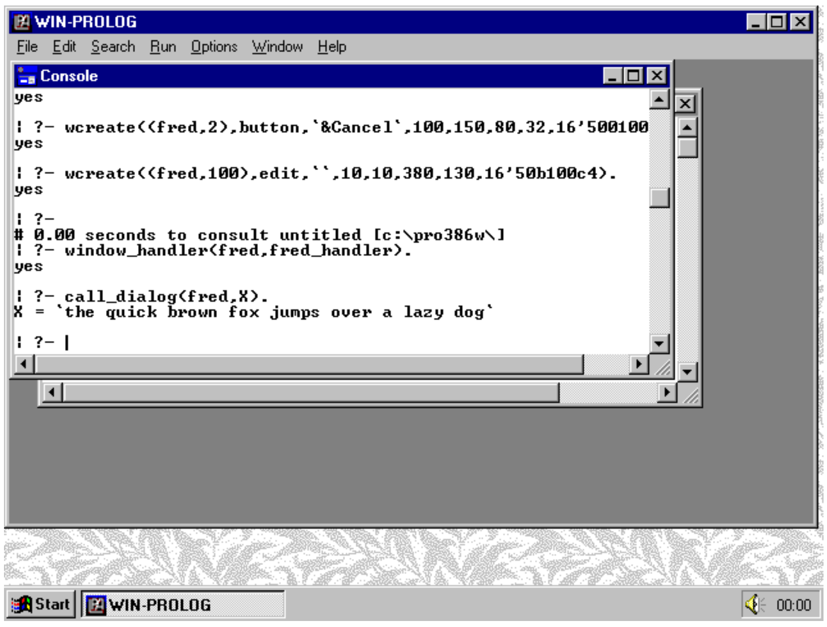
The dialog will reappear, complete with the text you typed on the previous call, as shown in *Fig 11.7*. Now click the "Cancel" button, and it will once again vanish, this time returning the variable "X" bound to the atom "cancel", as shown in *Fig 11.8*.



```
| ?- wcreate((fred,1),button,`&OK`,10,150,80,32,16'50010000).
yes

| ?- wcreate((fred,2),button,`&Cancel`,100,150,80,32,16'500100
yes

| ?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
yes

| ?-
# 0.00 seconds to consult
| ?- window_handler(fred,
yes

| ?- call_dialog(fred,X)
```

Use Fred's Cancel button to close

Fig 11.5 - The result of double-clicking the "close" icon

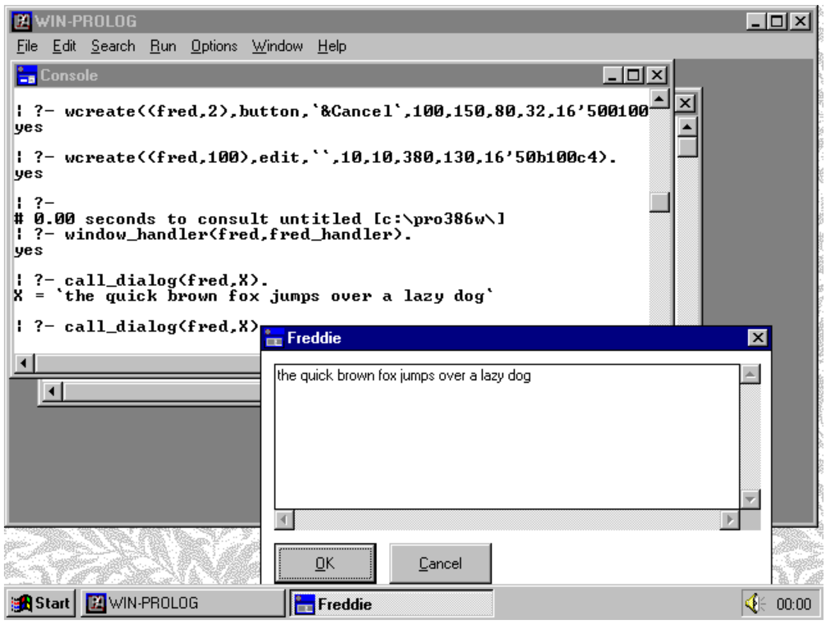


```
yes

¦ ?- wcreate((fred,2),button,`&Cancel`,100,150,80,32,16'500100
yes

¦ ?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
yes

¦ ?-
# 0.00 seconds to consult untitled [c:\pro386w\]
¦ ?- window_handler(fred,fred_handler).
yes

¦ ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

¦ ?- |
```

Fig 11.6 - Getting the edit box text by clicking "OK"

Writing a Modeless Dialog Handler

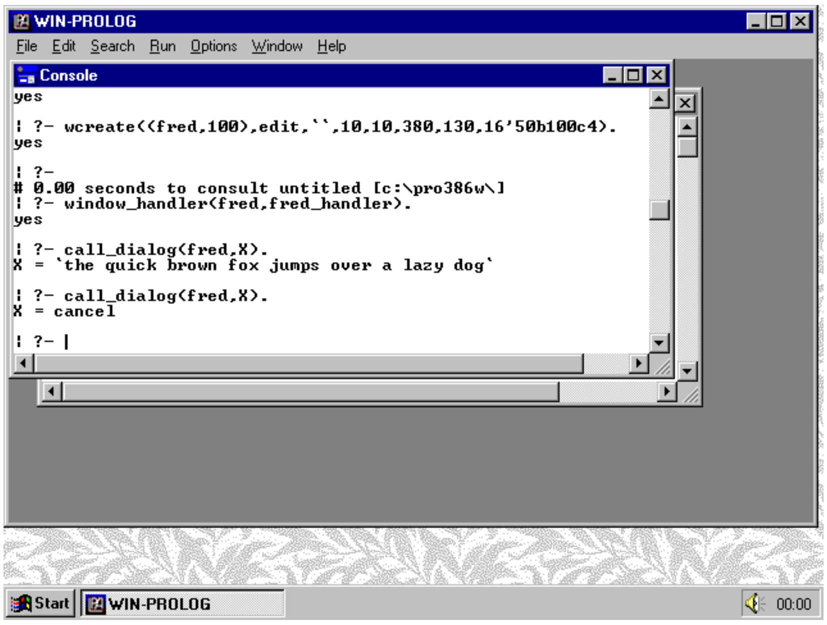
Modeless dialogs differ from modal ones only in that they do not return values: indeed, when they are called, they return control to the calling program immediately. Normally, a modeless dialog handler reacts to button clicks by performing specific actions, and as with any WIN-PROLOG hook or handler, it can perform literally any operation it likes. Click back on the "Untitled" program window, and edit the first clause of the *fred_handler/4* program (omitting the comment if you like):



```
¦ ?- wcreate((fred,2),button,`&Cancel`,100,150,80,32,16'500100
yes

¦ ?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
yes

¦ ?-
# 0.00 seconds to consult untitled [c:\pro386w\]
¦ ?- window_handler(fred,fred_handler).
yes

¦ ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

¦ ?- call_dialog(fred,X).
```

```
the quick brown fox jumps over a lazy dog
```

Fig 11.7 - Reusing the modal dialog with call_dialog/2



```
yes

¦ ?- wcreate((fred,100),edit,``,10,10,380,130,16'50b100c4).
yes

¦ ?-
# 0.00 seconds to consult untitled [c:\pro386w\]
¦ ?- window_handler(fred,fred_handler).
yes

¦ ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

¦ ?- call_dialog(fred,X).
X = cancel

¦ ?- |
```

Fig 11.8 - The result of clicking the "Cancel" button

```
% when the OK button is clicked, write out the text in the edit window

fred_handler( (fred,1), msg_button, _, _ ) :-
        wtext( (fred,100), Text ),
        writeq( Text ),
        nl,
        ttyflush.
```

Modeless dialog handlers cannot return values through their fourth argument, because the program which invoked them will already have carried on executing other code; however, binding the fourth argument still has the effect of terminating the dialog: in this example, only the "Cancel" button does this.

The new first clause matches when you click the "OK" button "(fred,1)", writing out the entire text contents of the edit control "(fred,100)", but does not terminate the dialog, the remaining clauses behave just as in the modal dialog example. Once again choose "Run/Compile" from the main menu to compile your program, as shown in *Fig 11.9*, correcting any errors and recompiling as necessary.

Calling the Modeless Dialog

We will use the existing dialog "fred", completely unchanged, to illustrate a modeless dialog: click back on the console window, and type the command:

?- **show_dialog(fred).** *<enter>*

The single argument consists of the name of the dialog to be displayed ("fred"). Note that it is not necessary to call *window_handler/2* again, since the dialog/

handler registration we performed earlier is persistent, and needs to be done only once per ***WIN-PROLOG*** session.

Your dialog will be displayed, again with the input you typed previously. Notice that this time, control returns immediately to ***WIN-PROLOG***, which displays the "?-" prompt, as shown in *Fig 11.10*.

Click on the edit control, and type some more input, as shown in *Fig 11.11*, and then click on "OK": see how the contents of the edit control are displayed on the console, as shown in *Fig 11.12*. You can type more input, and press "OK", as often as you like, because the modeless dialog is not terminated by the "OK button. To make the dialog disappear, click the "Cancel" button, and it will vanish, as shown in *Fig 11.13*. Once again, the dialog has not been destroyed, but merely hidden.

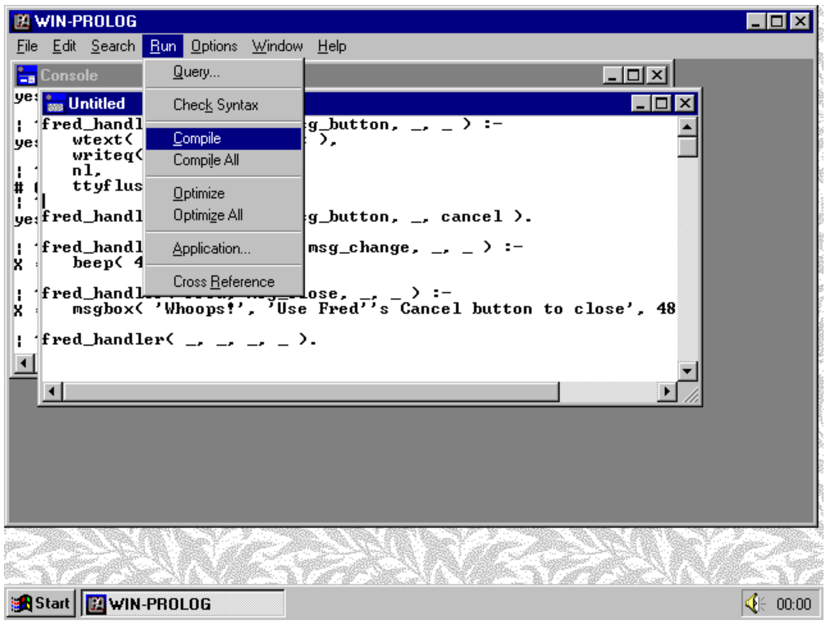
To show that the dialog is truly modeless, call it up again with the command:

```
?- show_dialog(fred).                                  <enter>
```

The dialog will be displayed, including the input you typed in the previous example. Now click back on the console window, and type the command:

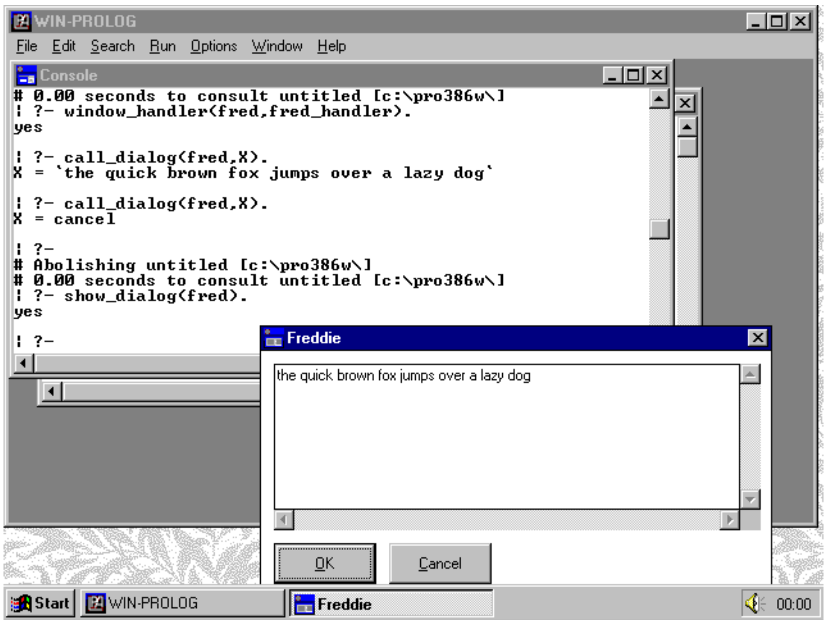
```
?- wshow(fred,0).                                      <enter>
```

This will cause the dialog to disappear, just as if you had clicked the "Cancel" button, as shown in *Fig 11.14*. Unlike modal dialogs, where you are forced to respond to them before performing any other actions, modeless ones merely coexist with the other ***WIN-PROLOG*** windows, allowing you to initiate actions (such as the copying and writing of the text of the edit control) whenever you



```
fred_handl                     g_button, _, _ ) :-
   wtext(                      ),
   writeq(
   nl,
   ttyflus

fred_handl                     g_button, _, cancel ).

fred_handl                     msg_change, _, _ ) :-
   beep( 4

fred_handl                     ose, _, _ ) :-
   msgbox( 'Whoops!', 'Use Fred''s Cancel button to close', 48

fred_handler( _, _, _, _ ).
```

Fig 11.9 - Compiling the modeless dialog handler



```
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- window_handler(fred,fred_handler).
yes

| ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

| ?- call_dialog(fred,X).
X = cancel

| ?-
# Abolishing untitled [c:\pro386w\]
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- show_dialog(fred).
yes

| ?-
```

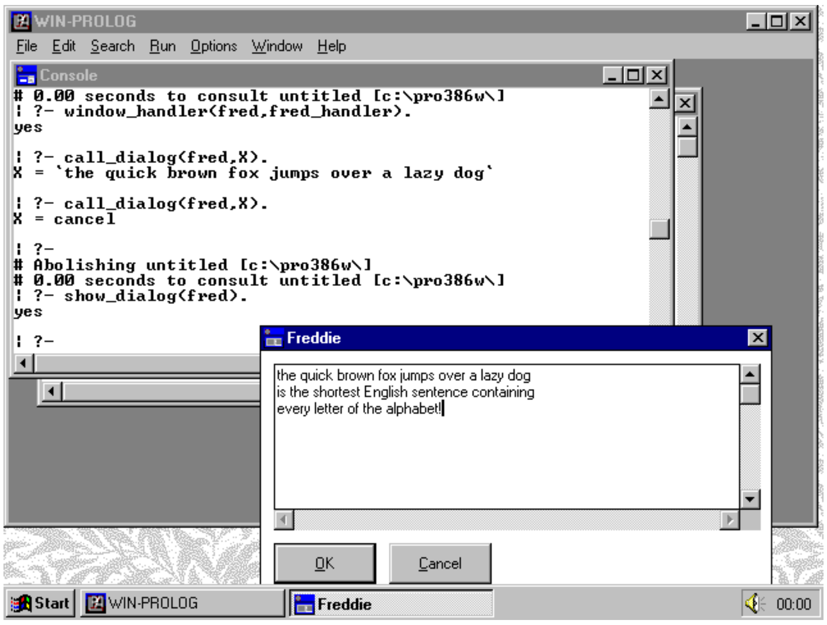
```
the quick brown fox jumps over a lazy dog
```

Fig 11.10 - Calling a modeless dialog with show_dialog/1

want to; this is examined a little further in the next section.

Responses and Actions

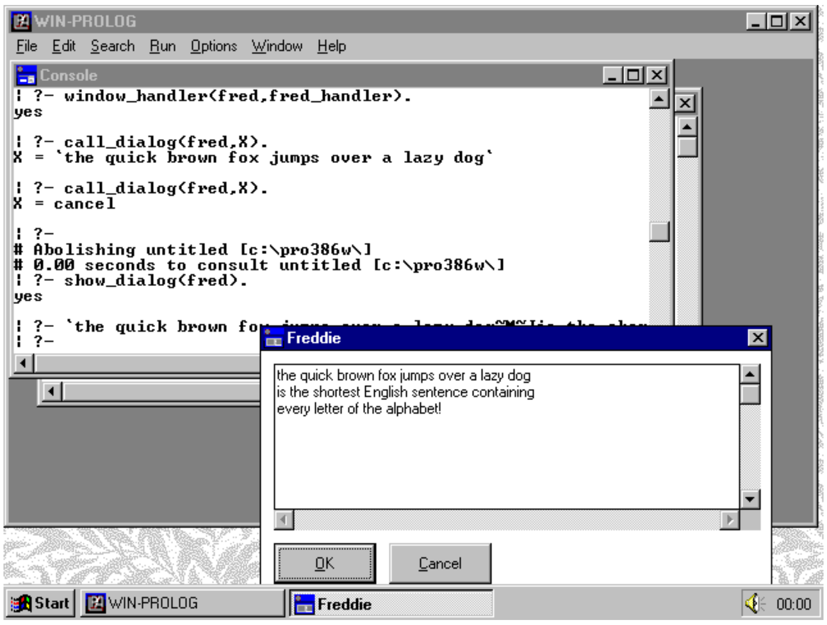
Modal dialogs can be thought of as requests for input: the system might be saying, "give me a file name", or perhaps, "decide which font you want to use"; when presented with a modal dialog, you are forced to respond, usually indicating acceptance by clicking an "OK" button, or rejection by clicking "Cancel". Modeless dialogs, on the other hand, do not request information; they are simply tools which you can select to initiate actions when you want, such as



```
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- window_handler(fred,fred_handler).
yes

| ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

| ?- call_dialog(fred,X).
X = cancel

| ?-
# Abolishing untitled [c:\pro386w\]
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- show_dialog(fred).
yes

| ?-
```

```
the quick brown fox jumps over a lazy dog
is the shortest English sentence containing
every letter of the alphabet!
```

Fig 11.11 - Adding further text to the modeless dialog



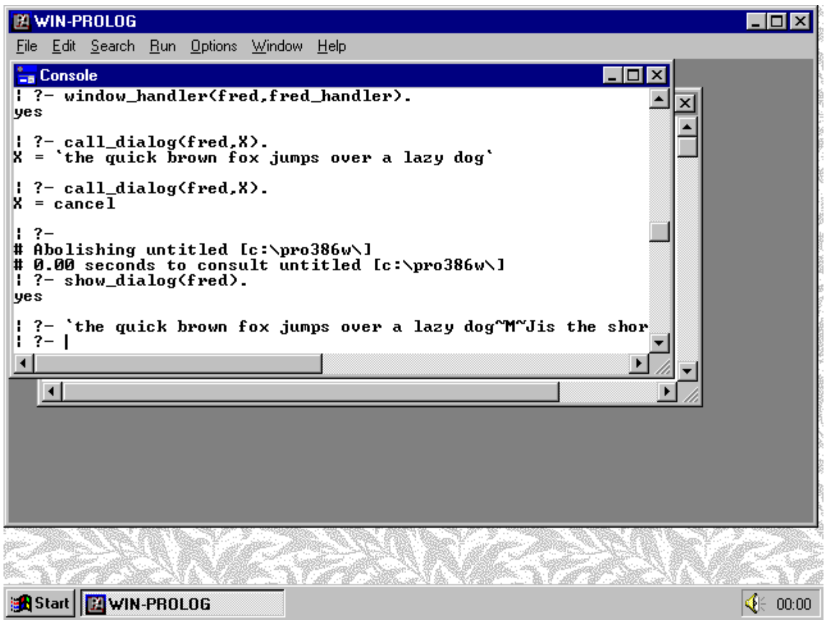
```
| ?- window_handler(fred,fred_handler).
yes

| ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

| ?- call_dialog(fred,X).
X = cancel

| ?-
# Abolishing untitled [c:\pro386w\]
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- show_dialog(fred).
yes

| ?- `the quick brown fo
| ?-
```

```
the quick brown fox jumps over a lazy dog
is the shortest English sentence containing
every letter of the alphabet!
```

Fig 11.12 - Text written to the console by clicking "OK"

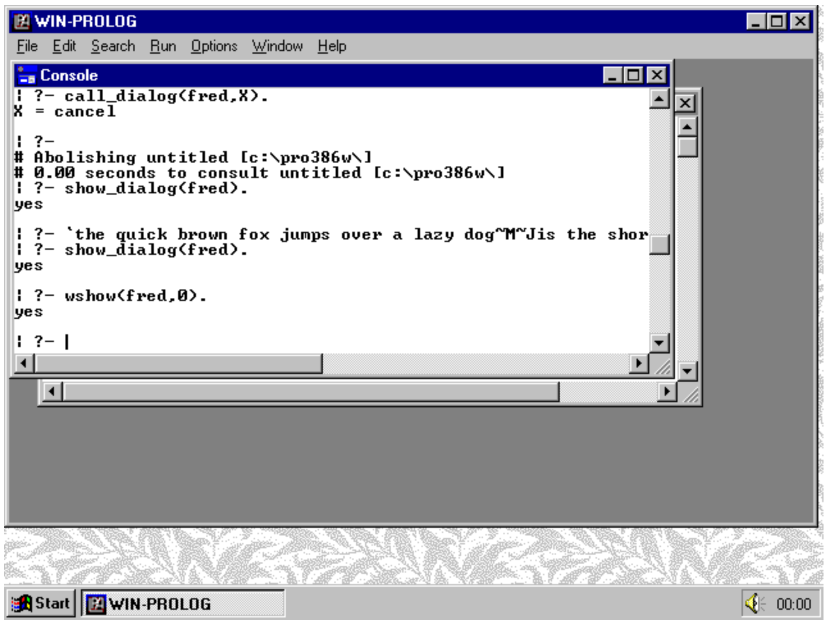
a text search or replace function during editing: in this respect, modeless dialogs behave less like modal dialogs, and more like menus.

When designing dialogs, the desired modality should be borne in mind. A modal "message" box, for example, will include buttons with labels like "OK", "Cancel", "Yes", "No", and so forth, simply allowing the user to indicate a response. A modeless "change" box for an edit will have buttons with names such as "Find", "Change", and so forth: these are the names of specific actions. To put an "OK" button into a modeless dialog (as we did in the example above) is usually a poor



```
| ?- window_handler(fred,fred_handler).
yes

| ?- call_dialog(fred,X).
X = `the quick brown fox jumps over a lazy dog`

| ?- call_dialog(fred,X).
X = cancel

| ?-
# Abolishing untitled [c:\pro386w\]
# 0.00 seconds to consult untitled [c:\pro386w\]
| ?- show_dialog(fred).
yes

| ?- `the quick brown fox jumps over a lazy dog~M~Jis the shor
| ?- |
```

Fig 11.13 - The result of clicking the "Cancel" button



```
¦ ?- call_dialog(fred,X).
X = cancel

¦ ?-
# Abolishing untitled [c:\pro386w\]
# 0.00 seconds to consult untitled [c:\pro386w\]
¦ ?- show_dialog(fred).
yes

¦ ?- `the quick brown fox jumps over a lazy dog^M^Jis the shor
¦ ?- show_dialog(fred).
yes

¦ ?- wshow(fred,0).
yes

¦ ?- |
```

Fig 11.14 - Hiding a modeless dialog with wshow/2

piece of design. The word "OK" is used to respond to a question, but modeless dialogs do not ask questions!

When used modally, our "fred" dialog is effectively saying "enter some text, and press OK when done"; the call to *call_dialog/2* actually waits until the second argument is bound to that text, or some other value. When used modelessly, "fred" is really waiting for you to initiate write commands, or to close it. The "OK" and "Cancel" buttons should really be renamed "Print" and "Close". Apart from recreating "fred" from scratch, the obvious way to rename the buttons is to use the *wtext/2* predicate. Type the following two commands:

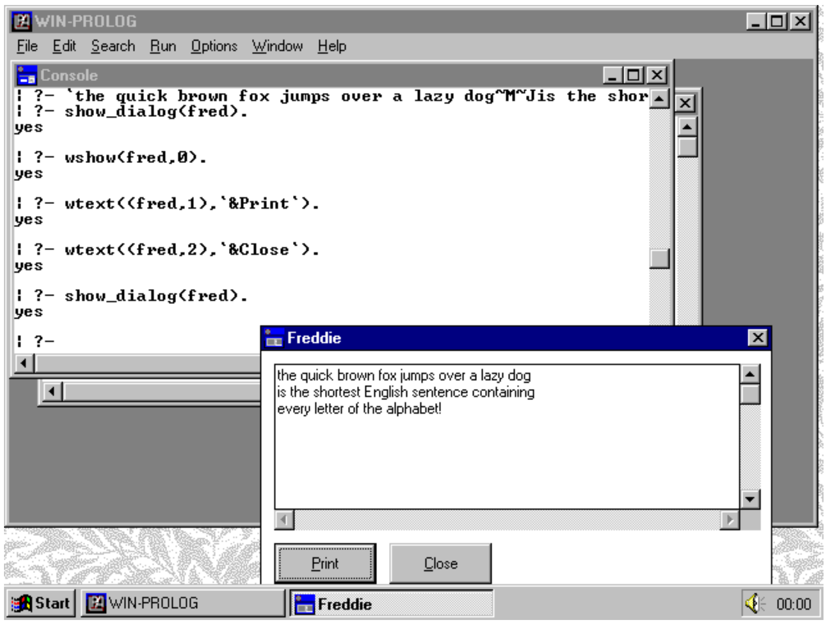
?- **wtext((fred,1),`&Print`).** *<enter>*

?- **wtext((fred,2),`&Close`).** *<enter>*

Now call the dialog modelessly for one last time, by typing the call:

?- **show_dialog(fred).** *<enter>*

The dialog "fred" will appear once again, but this time with its buttons relabelled, as shown in *Fig 11.15*. Click on the "Print" button, and the contents of the edit window will still be written to the console, as shown in *Fig 11.16*: because dialog handlers use the window handle ("(name,ID)"), and not the text labels, to identify the source of messages, your existing handler will work unchanged. Click on the "Close" button to make your dialog go away.



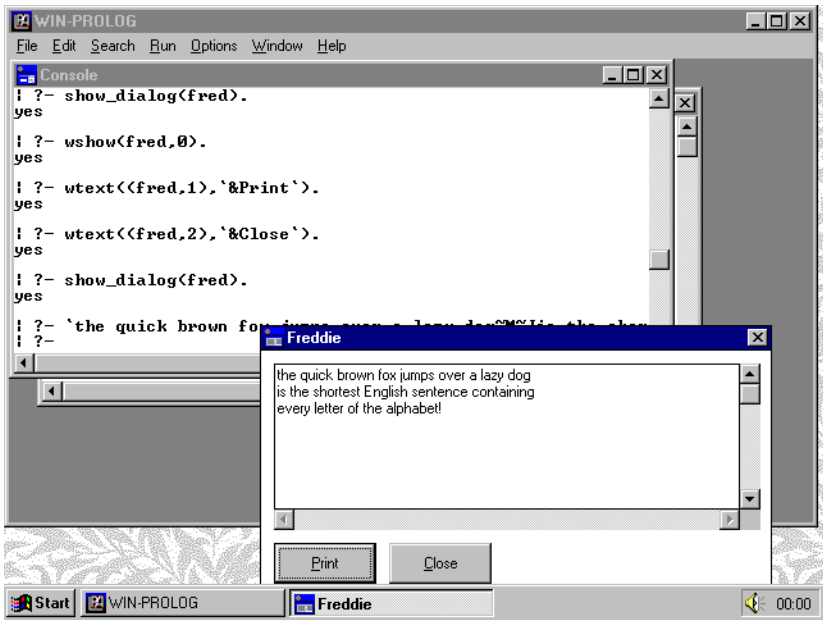
```
| ?- `the quick brown fox jumps over a lazy dog^M^Jis the shor
| ?- show_dialog(fred).
yes

| ?- wshow(fred,0).
yes

| ?- wtext((fred,1),`&Print`).
yes

| ?- wtext((fred,2),`&Close`).
yes

| ?- show_dialog(fred).
yes

| ?-
```

```
the quick brown fox jumps over a lazy dog
is the shortest English sentence containing
every letter of the alphabet!
```

Fig 11.15 - The dialog displayed with relabelled buttons

Freeing Memory Resources

As you have seen, dialogs are not destroyed after calls to *call_dialog/2* or *show_dialog/1*, but are merely hidden, so that they can be reused. In some applications, however, keeping copies of all dialogs between calls would be wasteful of memory, and so it would be desirable to free their resources. To do this, all you need to do is close the dialog window with the *wclose/1* predicate. Type the command:



```
| ?- show_dialog(fred).
yes

| ?- wshow(fred,0).
yes

| ?- wtext((fred,1),`&Print`).
yes

| ?- wtext((fred,2),`&Close`).
yes

| ?- show_dialog(fred).
yes

| ?- `the quick brown fo
| ?-
```

```
the quick brown fox jumps over a lazy dog
is the shortest English sentence containing
every letter of the alphabet!
```

Fig 11.16 - The result of clicking the relabelled "Print"

```
?- wclose(fred).                                        <enter>
```

This will close the dialog "fred", and its three controls, returning their resources to Windows. Note that you should not close a dialog which is currently being displayed by *call_dialog/2*, but should terminate the dialog through the handler first. This is because your program is waiting for a response from the dialog, and if you close the dialog without giving that response, your program will be waiting forever! Note also that you do not need to close controls individually: because they are children of the dialog window, they will be destroyed automatically by Windows when the dialog itself is closed.

Chapter 12 - Graphics

This chapter describes the Graphics subsystem of ***WIN*-PROLOG**, covering various aspects of their creation, manipulation and low level programming. The subject of graphics is large and potentially complex, and touches on many of the topics discussed in earlier chapters.

Fundamental Considerations

Under DOS, graphics programming is very simple. Programs can perform various types of output (lines, polygons, etc), and can wait for mouse or keyboard input. Applications tend to be system-driven: for example, a program to allow the user to define a polygon might contain logic along the following lines:

```
get_polygon( Poly ) :-
      get_mouse( X, Y, Button ),
      process_polygon( Button, X, Y, Poly ).

process_polygon( right, _, _, [] ).

process_polygon( left, X, Y, [X,Y|Poly] ) :-
      line_to( X, Y ),
      get_mouse( X1, Y1, Button ),
      process_polygon( Button, X1, Y1, Poly ).
```

Any input and output in such a program is basic and immediate: the mouse clicks are "read", and lines are "written". Graphics I/O is fundamentally identical to text I/O, using a simple extension of the glass teletype concept.

Under Windows, matters are not so cut and dry. Input is not performed by writing programs which wait for a mouse click: rather, mouse events, which include movement, clicking, releasing, double-clicking, etc., are reported to programs via "events" which interrupt program execution. Output is more complex, since as well as drawing shapes, programs must be able to redraw them at Windows' request: for example, if another window is brought on top of one containing graphics, when the top window is moved or closed, the newly exposed window will need to refresh its graphics. Under Windows, the structure of a polygon input program would be something like the following:

```
get_polygon( Poly ) :-
      abolish( poly/1 ),
      abolish( points/1 ),
      assert( points([]) ),
      repeat,
         wait( 0 ),
         def( poly, 1, _ ),
         poly( Poly ).

    poly_handler( right, _, _ ) :-
         points( P ),
         reverse( P, R ),
         assert( poly(R) ).

    poly_handler( left, X, Y ) :-
         retract( points(P) ),
         assert( points( [Y,X|P] ),
         lineto( X, Y ).

    poly_handler( paint, _, _ ) :-
         points( P ),
         polyline( P ).
```

The essential difference between the two versions of *get_polygon/1* is that the DOS version is a conventional procedural program which reads input, performs output, and recursively builds a data structure containing the required information, while the Windows version is a collection of handlers for individual events, and which uses global data structures to maintain information between these events.

Please note that the two "definitions" just presented consist of pseudocode, and use imaginary predicates, button names and messages: they are for illustrative purposes only. The remainder of this chapter uses real examples.

Windows and Device Contexts

One major concept needs to be discussed briefly before continuing with the details of graphics programming. In Windows, graphical operations are not performed directly to windows, but rather to a logical copy of the window, called the "device context". There are several types of device context, including those for windows, printers, bitmaps, metafiles and others. Once a device context has been obtained, it can be used together with generic, device-independent graphics code, effectively hiding the main differences between different types of physical device.

In **WIN-PROLOG**, device contexts are managed by three predicates, *gfx_begin/1*, *gfx_paint/1* and *gfx_end/1*. The differences between the first two of these will be explained below, but it is important to note that device contexts are potentially scarce resources, and are "borrowed" from the operating system, and so should always be "returned" upon completion of a graphics sequence. In practice, this simply means ensuring that each call to *gfx_begin/1* or *gfx_paint/1* is matched by an equivalent call to *gfx_end/1*.

Repainting Graphics

Probably the most onerous task of any Windows graphics application is the need to maintain a logical representation of the image in any window, so that it can be repainted when necessary. Simple operations such as the moving, scrolling and resizing of windows can cause portions of graphics regions to be uncovered. Windows does not maintain the contents of its windows, and so it is up to the application to redraw the newly exposed areas. In practice, this requires that the application store a copy of every "on-the-fly" graphics operation so that it can be repeated on demand. A simple Prolog program to draw a line in a window, and refresh it when needed, could be written as follows:

```
line( Window, X0, Y0, X1, Y1 ) :-
      gfx_begin( Window ),
      gfx( polyline(X0,Y0,X1,Y1) ),
      gfx_end( Window ),
      assert( grafix(polyline(X0,Y0,X1,Y1)) ).

grafix_handler( Window, msg_paint, _, _ ) :-
      gfx_begin( Window ),
      forall( grafix( Grafix ),
              gfx( Grafix )
            ),
      gfx_end( Window ).
```

The output program, *line/5*, obtains a device context for the window, performs the graphics, then returns the device context before storing a copy of its operation in the dynamic relation*grafix/1*. A window handler, *grafix_handler/ 4*, reacts to "MSG_PAINT" messages by repeating all such stored commands.

The only problem with this program is that each time any one part of the "Grafix" window needs to be refreshed, the entire window is redrawn. ***WIN*-PROLOG** maintains information about which part of a "Grafix" window needs to be redrawn, and this is represented by a "region" which is comprised of the accumulated "dirty" areas of a window. A refinement of the dialog handler just shown would be as follows:

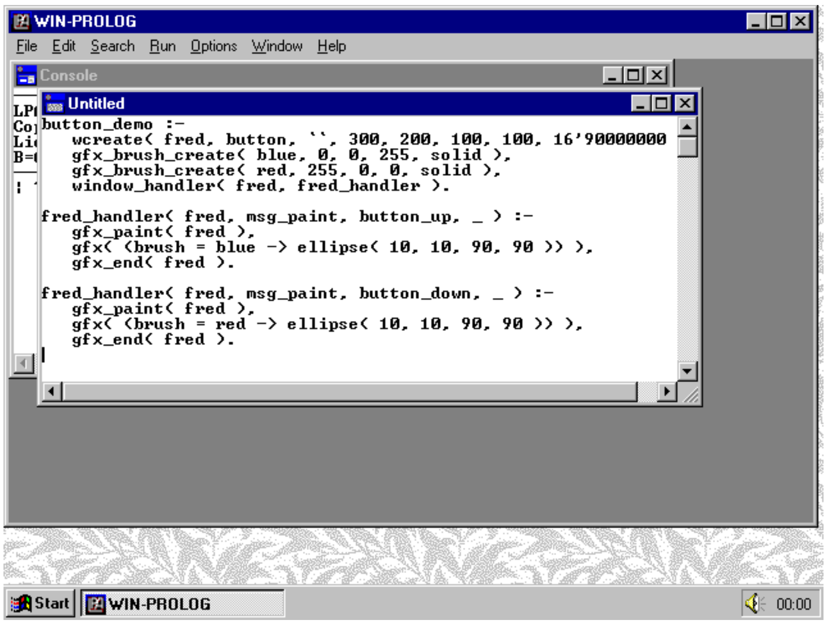
```
grafix_handler( Window, msg_paint, _, _ ) :-
      gfx_paint( Window ),
      forall( grafix( Grafix ),
              gfx( Grafix )
            ),
      gfx_end( Window ).
```

The predicate*gfx_paint/1* obtains a device context clipped to include only those portions of the window which have been invalidated, so that only those parts of a graphics window which need refreshing are redrawn.

Graphics, "Button" and "Grafix" Windows

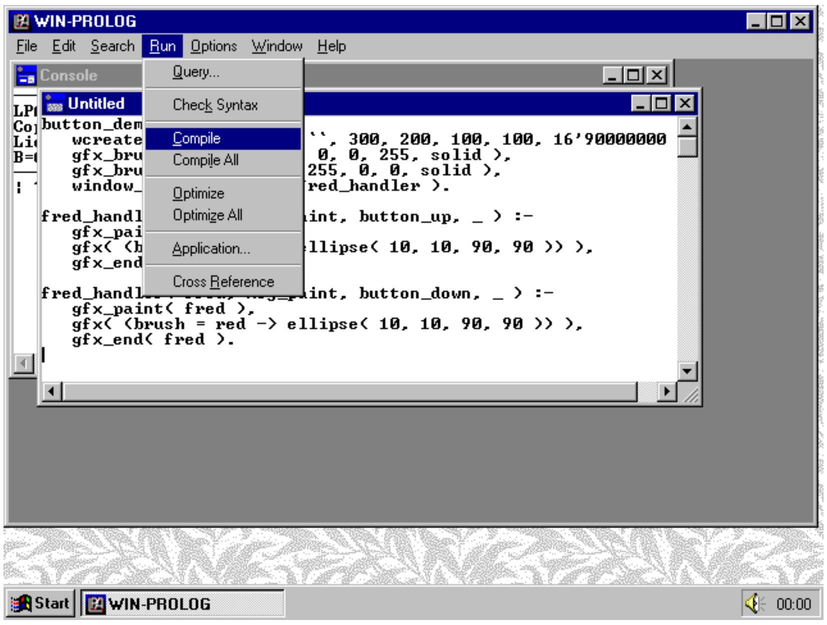
The *gfx/1* predicates can perform graphics output to any type of device context, but only two classes of window support for the "MSG_PAINT" messages which alert Prolog programs when a window needs repainting. One of these classes, the "Button", sends paint messages when it is clicked on or released, as well as when it is uncovered by another window, allowing graphics buttons to be created. Create a new untitled file by choosing the "File/New" menu option, and type in the following program as shown in *Figure 12.1*, and then choose the "Run/Compile" menu option to compile it, as shown in *Figure 12.2*.

```
button_demo :-
        wcreate( fred, button, ``, 300, 200, 100, 100, 16'90000000 ),
        gfx_brush_create( blue, 0, 0, 255, solid ),
        gfx_brush_create( red, 255, 0, 0, solid ),
        window_handler( fred, fred_handler ).

fred_handler( fred, msg_paint, button_up, _ ) :-
        gfx_paint( fred ),
        gfx( (brush = blue -> ellipse( 10, 10, 90, 90 )) ),
        gfx_end( fred ).

fred_handler( fred, msg_paint, button_down, _ ) :-
        gfx_paint( fred ),
        gfx( (brush = red -> ellipse( 10, 10, 90, 90 )) ),
        gfx_end( fred ).
```



```
button_demo :-
    wcreate( fred, button, ``, 300, 200, 100, 100, 16'90000000
    gfx_brush_create( blue, 0, 0, 255, solid ),
    gfx_brush_create( red, 255, 0, 0, solid ),
    window_handler( fred, fred_handler ).

fred_handler( fred, msg_paint, button_up, _ ) :-
    gfx_paint( fred ),
    gfx( (brush = blue -> ellipse( 10, 10, 90, 90 )) ),
    gfx_end( fred ).

fred_handler( fred, msg_paint, button_down, _ ) :-
    gfx_paint( fred ),
    gfx( (brush = red -> ellipse( 10, 10, 90, 90 )) ),
    gfx_end( fred ).
```

Fig 12.1 - A program to display a graphic button

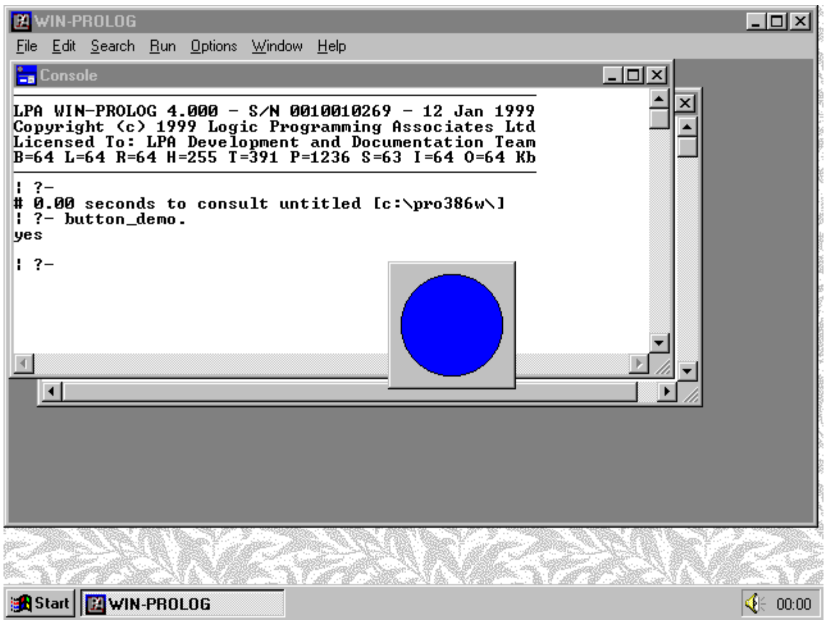


```
button_dem
   wcreate                 ``, 300, 200, 100, 100, 16'90000000
   gfx_bru                  0, 0, 255, solid ),
   gfx_bru                 255, 0, 0, solid ),
   window_                 red_handler ).

fred_handl                 int, button_up, _ ) :-
   gfx_pai
   gfx( (b                 llipse( 10, 10, 90, 90 )) ),
   gfx_end

fred_handl                 int, button_down, _ ) :-
   gfx_paint( fred ),
   gfx( (brush = red -> ellipse( 10, 10, 90, 90 )) ),
   gfx_end( fred ).
```

Fig 12.2 - Compiling the graphic button program

This program creates a "desktop" button in the middle of the screen, as shown in *Fig 12.3*, which contains a blue circle whenever it is unpressed and a red one whenever it is pressed. Note that "Button" windows do not maintain a "dirty" region, and so are always repainted in their entirety. Because they are small, and will typically contain simple graphics, this is not generally a problem.

The other class of window which sends "MSG_PAINT" messages, the "Grafix" window, also provides keyboard, mouse movement and mouse button messages, to allow virtually any type of input to be processed.



```
LPA WIN-PROLOG 4.000 - S/N 00010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

¦ ?-
# 0.00 seconds to consult untitled [c:\pro386w\]
¦ ?- button_demo.
yes

¦ ?-
```

Fig 12.3 - A desktop button containing graphics

Messages and Graphics

A number of messages in WIN-PROLOG relate specifically to graphics operations (see *Chapter 10* for more information about messages). The following sections describe the graphics messages in greater detail.

MSG_PAINT - Window needs painting

This message is shared by "Button" and "Grafix" windows, and is generated whenever their contents need repainting. The data parameter contains one of three values listed in *Table 12.1*:

Table 12.1 - MSG_PAINT Data Values

Prolog Atom	Meaning
grafix	Window is a "Grafix" window
button_up	Window is an un-pressed "Button" window
button_down	Window is a pressed "Button" window

When reacting to "Grafix" windows, it is important to obtain a clipped device context using *gfx_paint/1*, and to redraw any graphics before returning the device context with *gfx_end/1*; for "Button" windows, these steps are optional, and are only needed if it is desired to add some graphics to the button.

MSG_LEFTDOWN - Left mouse button pressed

This message is generated by "Grafix" windows only, and indicates that the left mouse button has been pressed. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y). For any message of this kind, there will always be a corresponding "MSG_LEFTUP" message.

MSG_LEFTDOUBLE - Left mouse button double clicked

This message is generated by "Grafix" windows only, and indicates that the left mouse button has been double-clicked. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y). For any message of this kind, there will always be a preceding "MSG_LEFTDOWN", and corresponding "MSG_LEFTUP" message.

MSG_LEFTUP - Left mouse button released

This message is generated by "Grafix" windows only, and indicates that the left mouse button has been released. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y). This message will only be generated if there has been a corresponding "MSG_LEFTDOWN" or "MSG_LEFTDOUBLE" message.

MSG_RIGHTDOWN - Right mouse button pressed

This message is generated by "Grafix" windows only, and indicates that the right mouse button has been pressed. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y). For any message of this kind, there will always be a corresponding "MSG_RIGHTUP" message.

MSG_RIGHTDOUBLE - Right mouse button double clicked

This message is generated by "Grafix" windows only, and indicates that the right mouse button has been double-clicked. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y). For any message of this kind, there will always be a preceding "MSG_RIGHTDOWN", and a corresponding "MSG_RIGHTUP" message.

MSG_RIGHTUP - Right mouse button released

This message is generated by "Grafix" windows only, and indicates that the right mouse button has released. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y). This message will only be generated if there has been a corresponding "MSG_RIGHTDOWN" or "MSG_RIGHTDOUBLE" message.

MSG_MOUSEMOVE - Mouse moved to new position

This message is generated by "Grafix" windows only, and indicates that the cursor has moved to a new position over the window. It is also generated for any cursor movements after "MSG_LEFTDOWN", "MSG_LEFTDOUBLE", "MSG_RIGHTDOWN" and "MSG_RIGHTDOUBLE", until their corresponding "MSG_LEFTUP" or "MSG_RIGHTUP" messages, even when the cursor is moved outside the area of the window. The data parameter contains the X and Y coordinates of the cursor position, relative to the window origin, in the form of a conjunction (X,Y).

MSG_CHAR - Character returned from keyboard

This message is generated by "Grafix" windows only, and indicates that a keystroke has been made while the window was in focus. The data parameter contains the character code of the key that was typed, or one of the special key codes listed in *Table 12.2*:

Table 12.2 - MSG_CHAR Special Key Codes

Prolog Atom	Meaning
prior	The "PRIOR" key
next	The "NEXT" key
end	The "END" key
home	The "HOME" key
left	The "LEFT" key
up	The "UP" key
right	The "RIGHT" key
down	The "DOWN" key
select	The "SELECT" key
print	The "PRINT" key
execute	The "EXECUTE" key
snapshot	The "SNAPSHOT" key
insert	The "INSERT" key
delete	The "DELETE" key

Graphics Components: Objects and Functions

There are two main components to the graphics subsystem: objects and functions. The former family consists of traditional Windows GDI objects such as brushes, pens, fonts and so forth, as well as some *WIN-PROLOG*-defined ones such as foregrounds and backgrounds. Graphics functions are simply the primitives which actually perform drawing operations.

Graphics objects are applied to graphics functions in order to govern their appearance, using a syntax rather like the Prolog "if-then" construct. We have already seen an example of this with the *button_demo/0* program above, and here is another example of a graphics call:

```
gfx(  (       pen = red,
              brush = blue
          ->  rectangle( 100, 100, 200, 200 ),
              (       brush = green
              ->      ellipse( 200, 200, 300, 300 )
              )
      )  ).
```

This call selects two objects, a red pen and a blue brush, draws a rectangle with these attributes, before selecting a green brush (keeping the existing red pen), and drawing an ellipse. The pen and brushes are assumed to have been defined previously with calls to *gfx_pen_create/5* and *gfx_brush_create/5* respectively, and a device context is assumed to have been obtained by a call to either *gfx_begin/1* or *gfx_paint/1*.

The nested structure passed into *gfx/1* is known as a "GraFiX procedure", and can be arbitrarily complex. Objects selected in outer nested levels in GraFiX procedures are inherited by inner levels, but not vice versa; nesting can be arbitrarily deep, and is limited only by internal Windows stack structures. This chapter describes GraFiX in general terms: please see the *Technical Reference* for specific details about GraFiX procedures, primitives and predicates.

GraFiX Objects: Backgrounds

Backgrounds are ***WIN*-PROLOG** objects which consist of a colour, specified as an "RGB" triplet (three 8-bit integers, one each for red, green and blue) which, when selected into a device context, define the background colour for text and the "gaps" in hatched brushes and dotted lines. Several stock backgrounds are defined, as listed in *Table 12.3*:

Table 12.3 - Stock Backgrounds

Object Name	Description
white_back	white (0% black) background
ltgray_back	light grey (25% black) background
gray_back	mid grey (50% black) background
dkgray_back	dark grey (75% black) background
black_back	black (100% black) background
null_back	null (transparent) background

Additional backgrounds are created with the *gfx_back_create/4* predicate, which specifies a name and an RGB value for the background. For example, the call:

```
?- gfx_back_create( yellow, 255, 255, 0 ).
```

creates a background called "yellow", assigning it maximum values for red and green, and minimum for blue. This background could then be selected in a *gfx/1* call showing, for example, some text as follows:

```
?- gfx( (back = yellow -> text( 100, 100, `Hello World`)) ).
```

Backgrounds are closed with the *gfx_back_close/1* predicate, and a list of existing backgrounds (excluding the stock backgrounds) can be found with the *gfx_back_dict/1* predicate. A final predicate, *gfx_back_handle/2*, is used to convert between a background object and its handle.

GraFiX Objects: Bitmaps

Bitmaps are Windows objects which are used to store raster images. These images are created by other applications, such as Windows Paint(Brush), Adobe

PhotoShop, or by the device drivers for hardware such as scanners and fax modems. They do not affect the drawing of other objects, but are displayed directly in calls to *gfx/1*. There are no stock bitmaps.

Bitmaps are loaded from ".BMP" files using the *gfx_bitmap_load/2* predicate, and are closed with*gfx_bitmap_close/1*; they can be listed by*gfx_bitmap_dict/1* and their object descriptors and handles converted between one another with *gfx_bitmap_handle/2*.

GraFiX Objects: Brushes

Brushes are Windows objects which combine a colour, specified as an "RGB" triplet (three 8-bit integers, one each for red, green and blue) with a style, which may be solid or one of several hatched patterns. When selected into a device context, the brush defines the fill colour and pattern for solid objects apart from text. Several stock brushes are defined, as listed in *Table 12.4*:

Table 12.4 - Stock Brushes

Object Name	Description
white_brush	white (0% black) brush
ltgray_brush	light grey (25% black) brush
gray_brush	mid grey (50% black) brush
dkgray_brush	dark grey (75% black) brush
black_brush	black (100% black) brush
null_brush	null (transparent) brush

Additional brushes are created with the *gfx_brush_create/5* predicate, which specifies a name and an RGB value for the brush, and one of the styles listed in *Table 12.5*:

Table 12.5 - Brush Styles

Style Name	Description
solid	solid brush ( )
horizontal	horizontal hatch (-----)
vertical	vertical hatch (|||||)
fdiagonal	forwards diagonal hatch (\\\\\)
bdiagonal	backwards diagonal hatch (/////)
cross	cross hatch (+++++)
diagcross	diagonal cross hatch (XXXXX)

For example, the call:

```
?- gfx_brush_create( blue_stripes, 0, 0, 255, vertical ).
```

creates a brush called "blue_stripes", with its colour set to maximum intensity blue, and the vertical stripe style. This brush could then be selected in a *gfx/1* call showing, for example, a filled circle as follows:

```
?- gfx( (brush = blue_stripes -> ellipse( 100, 100, 100, 100 )) ).
```

Brushes are closed with the*gfx_brush_close/1* predicate, and can be listed with *gfx_brush_dict/1*; finally, *gfx_brush_handle/2* is used to convert between a brush object and its handle.

GraFiX Objects: Cursors

Cursors are Windows objects which are used to track the cursor position. No support is provided in **WIN-PROLOG** for creating, loading or closing cursors, but a number of stock cursors are supplied, as listed in *Table 12.6*:

Table 12.6 - Stock Cursors

Object Name	Description
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
sizens_cursor	n/s two-headed arrow cursor
sizeall_cursor	win32 size all cursor
no_cursor	win32 no cursor
appstarting_cursor	win32 application stating cursor

Cursors are not strictly graphics objects, but their handling is related to that of the other objects. The only cursor object predicate is *gfx_cursor_handle/2*, which is used to convert between a cursor object and its handle. Each "Grafix" window can be assigned its own cursor, using the *gfx_window_cursor/2* predicate; for example:

```
?- gfx_window_cursor( (fred,1), stock(wait_cursor) ).
```

causes the "wait" cursor to be displayed automatically whenever the cursor is directly over the "Grafix" window "(fred,1)".

GraFiX Objects: Fonts

Fonts are Windows objects which, when selected into a device context, define the outline shapes used to display text Several stock fonts are defined, as listed in *Table 12.7*:

Table 12.7 - Stock Fonts

Object Name	Description
oem_fixed_font	OEM fixed font (IBM PC char set)
ansi_fixed_font	ANSI fixed font (Windows char set)
ansi_var_font	ANSI var font (Windows char set)
system_font	system var font (Windows char set)
device_default_font	default fixed font (Windows char set)
system_fixed_font	system fixed font (Windows char set)

Additional fonts are created with the *gfx_font_create/4* predicate, which specifies a name, a typeface, a point size and one of the styles listed in *Table 12.8*:

Table 12.8 - Font Styles

Style Name	Description
normal	normal roman font
italic	normal italic font
bold	bold roman font
bolditalic	bold italic font

For example, the call:

```
?- gfx_font_create( big, 'courier new', 48, bold ).
```

creates a font called "big" from the "Courier New" typeface, in 48 point bold. This font could then be selected in a *gfx/1* call showing, for example, some text as follows:

```
?- gfx( (font = big -> text( 100, 100, `BOO!` )) ).
```

Fonts are closed with the *gfx_font_close/1* predicate, listed with *gfx_font_dict/1* and their object descriptions and handles are converted by *gfx_font_handle/2*. Note that the gfx_font*/n predicates overlap in functionality with the wf*/n family (*wfcreate/4*, *wfclose/1*, etc), and that fonts created with either set of predicates can be used by the other. In due course, the older wf*/n family will probably be dropped from the system, and the gfx_font*/n family extended to encompass additional functionality.

GraFiX Objects: Foregrounds

Foregrounds are **WIN-PROLOG** objects which consist of a colour, specified as an "RGB" triplet (three 8-bit integers, one each for red, green and blue) which, when selected into a device context, define the foreground colour for text. Several stock foregrounds are defined, as listed in *Table 12.9*:

Table 12.9 - Stock Foregrounds

Object Name	Description
white_fore	white (0% black) foreground
ltgray_fore	light grey (25% black) foreground
gray_fore	mid grey (50% black) foreground
dkgray_fore	dark grey (75% black) foreground
black_fore	black (100% black) foreground

Additional foregrounds are created with the*gfx_fore_create/4* predicate, which specifies a name and an RGB value for the foreground. For example, the call:

```
?- gfx_fore_create( green, 0, 255, 0 ).
```

creates a foreground called "green", assigning it maximum intensity for green, and minimum for red and blue. This foreground could then be selected in a *gfx/1* call showing, for example, some text as follows:

```
?- gfx( (fore = green -> text( 100, 100, `Ecology`)) ).
```

Foregrounds are closed with the *gfx_fore_close/1* predicate, and a list of existing foregrounds (excluding the stock foregrounds) can be found with the *gfx_fore_dict/1* predicate. A final predicate, *gfx_fore_handle/2*, is used to convert between a foreground object and its handle.

GraFiX Objects: Icons

Icons are Windows objects which are used to store small raster images to identify files, programs and windows. These images are created by other applications, such as ImagEdit. They do not affect the drawing of other objects, but are displayed directly in calls to*gfx/1*. There are a number of stock icons, which are listed in *Table 12.10*:

Table 12.10 - Stock Icons

Object Name	Description
hand_icon	stop sign icon
question_icon	question mark icon

exclamation_icon	exclamation mark icon
asterisk_icon	information icon

Icons are loaded from ".ICO", ".EXE", ".DLL" and other files using the *gfx_icon_load/3* predicate, and are closed with *gfx_icon_close/1*; they can be listed by *gfx_icon_dict/1* and their object descriptors and handles converted between one another with *gfx_icon_handle/2*.

GraFiX Objects: Metafiles

Metafiles are Windows objects which are used to store vector images. These images are created by other applications, such as CorelDRAW!, or even by**WIN-PROLOG** programs. They do not affect the drawing of other objects, but are displayed directly in calls to *gfx/1*. There are no stock metafiles.

Metafiles are loaded from ".WMF" files using the *gfx_metafile_load/2* predicate, and are closed with *gfx_metafile_close/1*; they can be listed by *gfx_metafile_dict/1* and their object descriptors and handles converted between one another with *gfx_metafile_handle/2*.

GraFiX Objects: Pens

Pens are Windows objects which combine a colour, specified as an "RGB" triplet (three 8-bit integers, one each for red, green and blue) with a style, which may be a solid thickness or one of several dotted or dashed patterns. When selected into a device context, the pen defines the outline colour and pattern for solid objects apart from text. Several stock pens are defined, as listed in*Table 12.11*:

Table 12.11 - Stock Pens

Object Name	Description
white_pen	white (0% black) pen
black_pen	black (100% black) pen
null_pen	null (transparent) pen

Additional pens are created with the *gfx_pen_create/5* predicate, which specifies a name and an RGB value for the pen, and either an integer thickness or one of the styles listed in *Table 12.12*:

Table 12.12 - Pen Styles

Style Name	Description
solid	solid pen ( )
dash	dashed pen (-----)
dot	dotted pen (.....)

dashdot	dashed/single dotted pen (_._._)
dashdotdot	dashed/double dotted pen (_.._.)

For example, the call:

```
?- gfx_pen_create( black_thick, 0, 0, 0, 5 ).
```

creates a pen called "black_thick", with its colour set to black, and a solid thickness of 5 pixels. This pen could then be selected in a *gfx/1* call showing, for example, a thick-framed square as follows:

```
?- gfx( (pen = back_thick -> rectangle( 100, 100, 100, 100 )) ).
```

Pens are closed with the *gfx_pen_close/1* predicate, and can be listed with *gfx_pen_dict/1*; finally, *gfx_pen_handle/2* is used convert between a pen object and its handle.

GraFiX Objects: Raster Operations

Raster operations, or "rops" are **WIN-PROLOG** objects which, when selected into a device context, defines the way in which a graphics primitive combines its output with data already drawn on the device context. No support is provided in **WIN-PROLOG** for creating, loading or closing rops, because all possible cases are supplied as stock rops, as listed in *Table 12.13*:

Table 12.13 - Stock Rops

Object Name	Description
black_rop	black raster operation mode
notmergepen_rop	not merge pen raster operation mode
masknotpen_rop	mask not pen raster operation mode
notcopypen_rop	not copy pen raster operation mode
maskpennot_rop	mask pen not raster operation mode
not_rop	not raster operation mode
xorpen_rop	xor pen raster operation mode
notmaskpen_rop	not mask pen raster operation mode
maskpen_rop	mask pen raster operation mode
notxorpen_rop	not xor pen raster operation mode
nop_rop	nop raster operation mode
mergenotpen_rop	merge not pen raster operation mode
copypen_rop	copy pen raster operation mode
mergepennot_rop	merge pen not raster operation mode
mergepen_rop	merge pen raster operation mode
white_rop	white raster operation mode

For example, the call:

```
?- gfx( (rop = stock(xorpen_rop) -> polyline( 10, 10, 100, 100 )) ).
```

draws a diagonal polyline segment in "xor" mode. The only rop object predicate is *gfx_rop_handle/2*, which is used to convert between a rop object and its handle.

GraFiX: Setting the Device Context

All GraFiX operations are performed within a logical "device context", which allows the same piece of GraFiX code to be used in a window on the screen, on the printer, for hit testing, or other features not yet implemented (such as bitmap and metafile creation). Before any calls can be made to *gfx/1*, a device context must be set up, usually by a call to*gfx_begin/1*. This takes the name of a window, and stores its device context on an internal stack. One or more calls is then made to *gfx/1* to perform graphics operations, at the end of which the device context should be unstacked and returned with a call to *gfx_end/1*. For example, a simple program to draw a circle in a "Grafix" window called "(fred,1)" might be written as follows:

```
fred :-
      gfx_begin( (fred,1) ),
      gfx( ellipse( 100, 100, 200, 200 ) ),
      gfx_end( (fred,1) ).
```

Normally, the setting up and handling of the device context should be kept separate from the code which actually performs output, to facilitate the porting of code between devices. The above program could be rewritten:

```
fred :-
      gfx_begin( (fred,1 ),
      circle,
      gfx_end( (fred,1 ).

circle :-
      gfx( ellipse( 100, 100, 200, 200 ) ).
```

The benefit of this approach will become apparent below, but is mainly due to the device-independent nature of GraFiX calls, which would enable "circle/0" to be called for painting, printing, and hit testing as well as in the initial drawing of a graphic.

When a "Grafix" or "button" window needs repainting, a "MSG_PAINT" message is sent by **WIN-PROLOG** to its window handler. The correct response to this is to call *gfx_paint/1* to obtain a clipped device context, and then to redraw any graphics. For example, to refresh the circle above, the window handler for dialog "fred" should include the clause:

```
fred_handler( (fred,1), msg_paint, _, _ ) :-
      gfx_paint( (fred,1) ),
      circle,
      gfx_end( (fred,1) ).
```

Because we separated explicit device context setup routines (level 2) from actual GraFiX calls (level 3), we could use the "circle/0" program to perform actual graphics output in both the "fred/0" and "fred_handler/4" programs.

GraFiX: Changing Device Context Object Selections

Each time *gfx_begin/1* or *gfx_paint/1* is called to obtain a device context, any existing device context for the window concerned is saved on an internal stack, and all settings such as brush and pen selections, mapping modes, and so forth, are reset to their initial defaults (see above). As well as choosing selections for objects and modes within calls to *gfx/1*, several predicates allow the defaults to be changed. These changes last until the call to *gfx_end/1* which terminates the graphics sequence, at which points the previously existing device context is unstacked and all its settings restored.

Objects can be selected into a device context either during a call to *gfx/1*, using the "implication" (->) structure, or prior to a such a call by using the *gfx_select/1* predicate. This predicate allows settings to be selected over multiple GraFiX calls to improve efficiency. Consider a program which makes two calls to *gfx/1* to draw two shapes, both using a brush called "red" and a pen called "blue". These selections could be in each of the calls:

```
shapes1 :-
      gfx(  (     brush = red,
                  pen = blue
            ->    ellipse( 100, 100, 200, 200 )
         )  ),
      gfx(  (     brush = red,
                  pen = blue
            ->    rectangle( 200, 200, 300, 300 )
         )  ).
```

The selections of brush and pen could be made once only by rewriting this program as follows:

```
shapes2 :-
      gfx_select( (     brush = red,
                        pen = blue
                  ) ),
      gfx( ellipse( 100, 100, 200, 200 ) ),
          gfx( rectangle( 200, 200, 300, 300 ) ).
```

Of course, with this simple example, an even shorter version of the program could be written:

```
     shapes3 :-
          gfx(  (      brush = red,
                       pen = blue
                ->     ellipse( 100, 100, 200, 200 ),
                       rectangle( 200, 200, 300, 300 )
             )  ).
```

but the purpose of "shapes2/0" is to demonstrate that the default selections of objects can be overwritten by using settings for a series of graphics calls.

Any of the "selectable objects" or "selectable transformations" that can be selected with a call to *gfx/1* can also be selected by *gfx_select/1*.

GraFiX: Changing Device Context Mapping and Origin

As well as object selections, device contexts have both a "mapping" and an "origin" which between them define the mathematical relationship between logical (device) coordinates and physical (viewport) coordinates. The mapping is set up by the *gfx_mapping/4* predicate, which controls magnification and orientation, while the origin is set by *gfx_origin/2*. The default mapping is (0,0,0,0), which effectively means 1:1 magnification, while the default origin is (0,0), which means top left. The following program changes the mapping to give a 2.5:1 horizontal magnification, a 3:1 vertical magnification, with an origin at (100,200), before drawing a (distorted) circle using our previous "circle/0" example:

```
     distort :-
          gfx_begin( (fred,1) ),
          gfx_mapping( 2, 1, 5, 3 ),
          gfx_origin( 100, 200 ),
          circle,
          gfx_end( (fred,1  ).
```

Because the *gfx*/n* predicates only work with integer values, the desired horizontal scaling of "2.5:1" is set as "5:2".

A third predicate,*gfx_resolution/4*, is used to obtain the pixel-per-inch and total pixel resolution of the current device. Used in conjunction with *gfx_mapping/4*, it facilitates the mapping of logical coordinates to physical measurements. The following program sets up a mapping of 100 logical units per physical inch in an existing device context:

```
set_to_100dpi :-
        gfx_resolution( Horz, Vert, _, _ ),
        gfx_mapping( 100, 100, Horz, Vert ).
```

Similarly, the following sets up a device context so that its total area is divided into 1024 logical units in both dimensions:

```
set_1024_across_and_down :-
        gfx_resolution( _, _, Width, Depth ),
        gfx_mapping( 1024, 1024, Width, Depth ).
```

A final predicate, which fits into the present category, is *gfx_clipping/4*. This is used to define a clipping rectangle within the current device context, and is useful for trimming graphics within a bounding box. When the existing device context is already clipped (for example, when it has been obtained by a call to *gfx_paint/1* or where *gfx_clipping/4* has already been called), this predicate causes clipping to the union (overlap) of the existing clipping region and the newly specified clipping rectangle.

GraFiX: Mouse Interaction

A number of mouse messages are sent to "Grafix" windows, and in certain types of program, it is desirable to interact between graphics shapes and the cursor. For example, a program that allows objects to be moved about with the mouse needs to be able to tell when the cursor is over a particular object. The process of detecting which part(s) of a graphic coincide with the cursor is known as "hit testing", and it is supported with the *gfx_begin/3* and *gfx_end/3* predicates, together with *gfx_test/1*. The *gfx_begin/3* predicate sets up a special device context, which causes *gfx/1* calls to perform hit tests rather than actual output; the *gfx_end/3* predicate is used to terminate the test sequence, while *gfx_test/1* returns the number of "hits" so far at any point during the test. A "hit" is scored when the cursor coordinates specified in the call to *gfx_begin/3* impinge on an object defined in a call to *gfx/1*; for example, the program:

```
fred_test( X, Y, Count ) :-
        gfx_begin( (fred,1), X, Y ),
        circle,
        gfx_test( Count ),
        gfx_end( (fred,1), X, Y ).
```

will test whether or not a given pair of cursor coordinates (X,Y) impinge upon the circle drawn by "circle/0" in the "(fred,1)" window, returning a count of "1" or "0" respectively.

GraFiX: Printer Control

Much has been made about the device independence of the GraFiX system, but the physical differences between output devices, for example between windows and the printer, require some additional support. The predicates required to set up a user or dialog window, and to plant one or more "Grafix" controls in it, are discussed in earlier chapters. For the printer, four simple predicates provide equivalent functions, as described here. The printer is initialised with a call to *prnbox/4*, specifying the name of the document, or to *prnini/4*, additionally specifying the printer, its device driver and output port. Each new page is requested by a call to *prnpag/1*, which returns the current page number after ejecting any existing page. The status of a print job can be checked with *prnstt/1*, which returns one of four values, as listed in *Table 12.23*:

Table 12.23 - Printer Status Values

Status	Meaning
0	printer not initialised
1	printer ready with no current page
2	printer idle on current page
3	printer active on current page

Apart from setting up and handling page control with the printer, a printing program is identical to a window-based GraFiX program. For example, to print the "circle" example, a program could be written such as:

```
print_circle :-
      prnini( 'Circle', 'HP Deskjet Plus', 'hpdskjet', 'lpt1' ),
      prnpag( _ ),
      gfx_begin( [] ),
      circle,
      gfx_end( [] ),
      prnend( 0 ).
```

The call to *prnini/4* sets up a document called "Circle" on a printer called "HP Deskjet Plus", with printer driver "hpdskjet.drv" on the standard printer port "lpt1". Different versions of Windows need different bits of this information: for example, WinNT and Win95 ignore the driver argument ("hpdskjet"), while Win3.1 with Win32s ignores the printer name ("HP Deskjet Plus"). Once initialised, an initial page must be obtained with a call to*prnpag/1*; next, a printer device context is obtained with a call to*gfx_begin/1*, this time passing an empty list ("[]") as the argument to specify the printer, rather than a window. After performing the graphics, the printer device context is returned with a call to *gfx_end/1*, and the print job is completed with a call to *prnend/1*. The latter predicate takes a single integer argument which specifies how to terminate the print job, as shown in *Table 12.15*:

Table 12.15 - Printer Termination Values

Command	Meaning
0	terminate job normally, ejecting final page
1	terminate job abruptly, aborting pending pages

GraFiX: Window Control

Three predicates provide special support for "Grafix" windows, but are not directly part of the GraFiX system as such. The first of these, *gfx_window_cursor/2*, determines which cursor is displayed when the cursor is positioned directly over the named "Grafix" window. For example, the call:

```
?- gfx_window_cursor( (fred,1), stock(cross_cursor) ).
```

will cause the "cross" stock cursor to be displayed automatically whenever the cursor is over the "Grafix" window "(fred,1)". A second predicate, *gfx_window_redraw/5*, explicitly invalidates a specified portion of a "Grafix" window's client area before sending it a "MSG_PAINT" message. Provided that an appropriate window handler is attached to the dialog or user window containing the "Grafix" window, this message causes the latter to be redrawn. The third predicate, *gfx_window_scroll/3*, scrolls the client area of a "Grafix" window before sending it a "MSG_PAINT" message to request it to redraw the newly exposed portions; once again, the latter function requires a suitable window handler to be attached.

GraFiX: Device Contexts and Error Handling

As mentioned earlier, device contexts should be handled as scarce resources. Most versions of Windows only support five such device contexts, which are shared globally amongst all processes. For good behaviour, it is imperative that an application not only keep its device contexts for as short a time as possible, but also that it guarantees to restore them during error recovery. A well behaved **WIN-PROLOG** program should call *gfx_end/1* once for each corresponding call to *gfx_begin/1* and *gfx_paint/1*, but in the event of an error, this may not always be possible; the *gfx_cleanup/0* predicate allows programs restore all device contexts in such cases, as shown in the following simple error handler:

```
'?ERROR?'( Error, Goal ) :-
      writeq( Error - Goal ),
      nl,
      gfx_cleanup,
      abort.
```

Omitting the call to *gfx_cleanup/0* in a GraFiX program might result in device contexts not being returned to Windows, possibly causing subsequent display problems not just in **WIN-PROLOG**, but also in other concurrent processes.

Appendix A - Character Sets and Fonts

This appendix discusses the different character sets used by DOS and Windows, and shows how these impinge both on the user and on WIN-PROLOG and its programs.

ASCII, ANSI and the IBM PC Legacy

Versions of LPA-PROLOG up to 4.040 utilised a 256-character set based on that in the IBM PC ROM BIOS. More correctly known as "*Codepage 437*", it comprised of two halves: the lower 128 characters (characters 00h..7fh) of this set conform to the 7-bit ASCII standard, but the upper half of the table, (characters 80h..ffh), contain a mixture of accented letters, as well as the majority of currency symbols and graphics characters which do not map onto any accepted standard. Most importantly, the top half of Codepage 437 does not map well either to the Windows "*ANSI*" character set, or to *Unicode*: for example, the UK "Pound" sign ("£") is IBM PC code 9ch (156 decimal), while in Windows and in Unicode it is ach (163 decimal).

The 32-bit Character Set, Unicode and ISO/IEC 8859-1

Starting in version 4.100, LPA-PROLOG has replaced the old 256-character set with one containing over 4,000,000,000 characters: specifically, its internal data structures now handle characters of up to 32-bit width, as opposed to the old 8-bit characters. Full details of how this is done, without seriously impinging on program size or memory usuage, can be found by reading *Appendix L* in the main *Technical Reference*.

The first 128 characters are still mapped onto ASCII, a universally accepted 7-bit character standard, but more importantly, it is within this set that reside all characters with special meanings to LPA-PROLOG. Punctuation, graphic, digit, bracket, quote and other special characters are all ASCII characters: in earlier versions of the system, similar attributes were also assigned to characters in the range 80h..0ffh, according to the character's meaning in Codepage 437. This is no longer the case: these characters are now assumed to map to their Unicode equivalents, which leads to the next standard...

The first 256 characters in LPA-PROLOG map directly to a standard called "*ISO/IEC 8859-1*", which we will normally simply call "*ISO*" for the sake of brevity. ISO is very similar to Windows "*ANSI*", with only a few character differing; ISO is preferable to ANSI because it maps to the first 256 characters of Unicode. The main difference between ISO and ANSI is that the former (like Unicode) reserves character codes 80h..9fh for control purposes, while the latter assigns certain characters, such as the Euro currency symbol, "€"and the Trademark Sign, "™", to within this range.

With a few special exceptions, the first 1,000,000 or so characters in **LPA-PROLOG** map directly to the "*Unicode*" standard. Originally conceived of as a simple 16-bit character encoding, Unicode version 3.0 incorporates so-called "*surrogate pairs*", comprising codes in the ranges 0d800h..0dfff, some bits of which are extracted to make up 20-bit characters. In Unicode, these codes cannot be used as characters in their own right, but **LPA-PROLOG** has no such restrictions for its internal 32-bit character set.

The remaining 4,000,000,000 or characters characters in **LPA-PROLOG** have no current mapping, but will be assumed to map to future extensions to Unicode; the full 32-bit character set is known in this documentation simply as "*RAW*". The relationship between the various parts of the **LPA-PROLOG** character set are described in *Table A.1*:

Table A.1 - Character Sets

Name	Description
ASCII	the first 128 characters, which are also part of...
ISO/IEC 8859-1	the first 256 characters, which are also part of...
Unicode	the fist 1,000,000 or so characters, included in...
RAW	all 2^{32} characters supported by LPA-PROLOG

The Lexical Table

It has already been mentioned that versions of **LPA-PROLOG** up to 4.040 assigned special meanings to each of the first 256 characters, using their Codepage 437 definitions to determine whether characters in the range 80h..ffh were letters, graphics or something else. The types of the characters were stored internally in a 256-byte array, called the "*lexical table*". With the advent of version 4.100, **LPA-PROLOG**'s character set is extended not just to the 1,000,000+ characters in Unicode, but to the 4,294,967,296 characters supported by the new 32-bit character type.

Patently there is no way in which a lexical table could be used to assign special types to each of more than four billion characters, so from version 4.100 onwards, the **LPA-PROLOG** lexical table has been cut to just 128 entries, covering the 7-bit ASCII character set. All other characters, in the range 80h..FFFFFFFFh, are now assumed to be lowercase letters. This has a few, mostly beneficial implications. Firstly, atoms containing arbitrary mixtures of non-ASCII characters no longer need to be quoted, so programs can be written more easily than before in non-English languages. Secondly, it removes the old need to convert "ANSI" text into "OEM" (Codepage 437) text before compiling or otherwise processing in **LPA-PROLOG**. The only consideration that needs attention is where characters previously treated as graphics, such as the UK Pound sign (£), are present in a previously unquoted atom. For example, the following call will now generate a syntax error:

```
?- write( $£$ ).                                        <enter>
```

Previously, bouth the dollar ($) and pound (£) characters were treated as graphic symbols. The former is an ASCII character, and retains its assignment, but the latter is ISO code a3h, which (like all other non-ASCII characters) is now treated as a lowercase letter. So the above call must be written with quotes:

```
?- write( '$£$' ).                                      <enter>
```

Conversely, a program that defined "£" as a prefix operator, so that the term "£(123)" could be written without the brackets, such as in the call:

```
?- Price = £123.                                        <enter>
```

now requires that at least one space is placed after the pound sign, because by itself, "£123" is simply a single atom, just as is, say, "a123".

Old Source Files and the Codepage 437 Character Set

Files containing source code for LPA-PROLOG versions up to 4.040 might contain characters in the range 80h..ffh, such as the UK Pound sign, but these will be assigned to their Codepage 437 values, such as 9ch in this case. Such files will display these characters incorrectly when loaded into WIN-PROLOG 4.100. An archaic predicte, *ansoem/2*, persists as a method of converting between Codepage 437 (formerly called the "OEM" character set) and ANSI (which, apart from a few characters, is the same as ISO/IEC 8859-1). This predicate can be used to convert any non-ISO text into its correct new form.

New Source Files, ISO/IEC 8859-1 and Unicode

Source files under LPA-PROLOG version 4.100 and later are assumed to contain characters which conform to their Unicode meanings. This is not to say that all files must now contain Unicode, with its space implications (16-bit characters, for example): it simply means that a file containing 8-bit characters is assumed to map to the first 256 characters of Unicode, which in turn means that this is an ISO/IEC 8859-1 ("ISO") file. Any such file containing nothing but 7-bit characters is also an ASCII file. The WIN-PROLOG 4.100 environment recognises a range of Unicode file formats automatically, and loads these accordingly. When saving files, the environment writes them as 8-bit ISO files if possible, resorting to Unicode only when necessary.

Sorting and Term Comparison

Unicode specifies very complex rules regarding sorting and comparison in text, but these are not applied within LPA-PROLOG itself. Instead, as it always has done, this system sorts atoms and strings according to the character codes that comprise their text. All that has changed is that these codes used to be unsigned 8-bit integers, and are now unsigned 32-bit integers.

Appendix B - Text Data Types

This appendix discusses the string in relation to the other text data types in LPA-PROLOG, namely atoms and char lists, because of the former's considerable importance in Windows programming.

The Atom

Atoms, which are written using the single quote ('), or without quotation, are the fundamental text objects in Prolog, and are used to name predicates, files, windows, and so forth. In LPA-PROLOG, each atom occurs only once in memory, stored in a special dictionary. References to this atom in code and data structures are in the form of a simple 32-bit pointer. Each time an atom is encountered on input, it must be searched for in the dictionary. If it is not found, then a new atom is created, and its address stored in the input term. If it is found, the address of the existing atom is stored.

Atoms are compact, but the need for lookup and dictionary maintenance takes time whenever they are created, and forces an upper limit on their length, namely 1024 bytes in LPA-PROLOG. On the plus side, because each atom exists only once in the whole workspace, comparing two existing atoms for identity is simple a matter of one 32-bit address comparison. Note that, following the introduction of Unicode support and a 32-bit character set in LPA-PROLOG 4.100, the 1024-byte limitation means an upper limit of between 204 and 1024 characters, depending on their size.

Atoms are used in Windows programming to name user windows and dynamic link libraries, although in many cases they can also be used in place of strings for other text parameters.

The Char list

Char lists, which are written using the double quote ("), are simply Prolog lists in which each element contains a single 32-bit integer value corresponding to the code for a character. Because each character occupies a list element, together with overheads for type tags and pointers to link each list element with the next, char lists require ten bytes for each character stored! One 64kb file would need well over half a megabyte of heap space to represent this way!

Char lists have great flexibility, because standard Prolog list processing algorithms can be used to manipulate their contents, and are limited in length only by the size of the available heap, theoretically up to 4Gb in LPA-PROLOG, but their considerable memory overhead makes them clumsy, and they are not used for Windows programming.

The String

Strings, which are written using the backward quote (`), provide a useful solution to the dilemma of how best to handle text in LPA-PROLOG. They are more flexible than atoms, being able to reach any length right up to the limits of available space in the text heap, and more compact than char lists, requiring on average around 1.3 bytes storage per character. Each time a string is encountered on input, it is created from scratch: thus two identical strings on an input term will result in two copies of the text being stored. Comparing strings is very quick when they are different: their lengths and initial few characters usually indicate the mismatch immediately; when two identical strings are compared, however, a complete character-for-character matching must be carried out.

Because no attempt is made to look strings up in a dictionary, they can be created more quickly than atoms. By sharing the atom's text storage mechanism, they do not use up heap space, and their compactness is a great advantage over char lists. For these reasons, many of the low level input/output features of LPA-PROLOG, including text operations in conjunction with windows, use strings as the data transfer medium.

Uses of Strings and Atoms in Windows

Atoms, as has been mentioned above, are used to name objects in Prolog. For example, files, predicates, windows and dynamic link libraries are named by atoms. Because of their limited length (1024 bytes, 204-1024 characters) in LPA-PROLOG), atoms are not ideal for representing larger amounts of data. Strings, on the other hand, can contain an indefinite number of characters in LPA-PROLOG. Both atoms and strings can be made up of bytes of any value.

Many Windows functions work together with strings at the C level. In C, strings are packed arrays of characters terminated with a null (zero) character. In interfacing between WIN-PROLOG and Windows, it is often necessary to pass text to and from the operating system. While atoms could have been used to transfer such data, their limited maximum length would involve breaking up text transfers into numerous smaller calls. LPA-PROLOG's strings provide a natural solution, since their capabilities actually exceed those of C's.

When a text parameter is passed from WIN-PROLOG to the underlying Windows system, the Prolog string is copied into a buffer in C string format, replacing any embedded null characters (which would appear to C as the end of the string) with spaces. Once Windows has performed its functions any resulting output strings are copied back to Prolog. Buffers, and their associated predicates, are described in *Appendix G*.

Character Encoding in Strings and Atoms

As has been mentioned in various places in this manual, LPA-PROLOG has introduced 32-bit characters to all its text types, principally in the full support of Unicode. The concerned reader might wonder how this affects text storage requirements within LPA-PROLOG itself: fortunately, for existing applications, the hanswer is that it hardly makes any difference, as we will explain here.

Characters in the range 00h..FDh, which includes all but two of the 8-bit ISO/IEC 8859-1 ("ISO") character set (and therefore the whole of ASCII) are stored, as before, one byte at a time in strings and atoms. Characters in the range FEh..FFFFh are stored using feh as a tag byte, with the actual code occupying the next two bytes of memory. The remaining characters, in the range 10000h..FFFFFFFFh, are stored using ffh as a tag byte, with the actual code occupying the next four bytes.

In short, this means that an atom containing all 256 8-bit ISO characters requires 254*1+2*3, or 260 bytes of text storage, compared with 256 bytes in older, pre-4.100 versions of LPA-PROLOG: this is an overhead of about 1.56% for random 8-bit binary data! Furthermore, any 8-bit text which avoids using the two characters, "thorn" and "y-umlaut" ("ÿ" and "þ" respectively), which includes most English and European text, requires exactly the same amount of storage in the new 32-bit character set model as in the previous 8-bit model: truly a case of something for (next to) nothing!

Optional and Compulsory Atoms and Strings

Some window handling predicates give you the option of using either strings or atoms, especially where the length of text item being processed is likely to remain short, for example with the text parameters of *msgbox/4* and *sttbox/2*, but in most cases, you are forced to use strings or atoms for specific arguments.

In general, atoms are used for text items which are names, such as typefaces, window classes and filenames, and strings are used for general text parameters, such as the contents of an edit control or displayed title of a window. Predicates such as *wcreate/8* clearly illustrate this point; the call:

```
?- wcreate(fred,dialog,`Freddie`,...).
```

gives "fred" (an atom) as the Prolog name for a window, "dialog" (another atom) as the class of window, and "`Freddie`" (a string) as its title. When you get or set a dialog title with *wtext/2*, the same types are used; the call:

```
?- wtext(fred,`Bloggs`).
```

changes the title of a window called "fred" (an atom) to the new value "`Bloggs`" (a string).

Appendix C - Programming Considerations

This appendix discusses some of the special considerations that should be borne in mind when writing for the Windows environment, especially with regards to the comparative simplicity of DOS.

Multitasking and Good Behaviour

The single biggest difference between Windows and DOS is that the former is a multitasking operating system. Within the constraints of memory, disk space and processor performance, large numbers of applications may simultaneously be running on a single processor, sharing hardware resources such as screen, disk and printer devices, as well as memory. It is important that applications are "well behaved", and are not selfish in their use of such resources, since one such program could bring the rest of Windows down onto its knees.

While **WIN-PROLOG** is a well behaved application, there is nothing to stop you, the Prolog programmer, writing code which would seriously affect the performance of other applications: under certain circumstances, you could even cause earlier (Win16) versions of Windows to hang indefinitely. While an exhaustive list of "do"s and "don't"s is beyond the scope of this manual (and there are plenty of good books available about how to write Windows applications), some of the more basic points deserve a little discussion. The remaining sections of this appendix outline just some such points.

Relinquishing Control

Although later (Win32) versions of Windows, such as WinNT and Win95/98, can perform preemptive multitasking, earlier (Win16) versions, such as Win3.1n, cannot: in order for multiple applications to run with Win16, each application has to yield control to Windows at regular intervals. The innermost control sequence in **WIN-PROLOG** yields to Windows once every 256 predicate calls (**DOS-PROLOG** checks DOS keyboard status for detection of <*ctrl-break*> with the same frequency); in addition, such relinquishement is performed by certain I/O operations, such as the flushing of the console window output buffer or the reading or writing of blocks of data from or to disk files.

Once optimised, many programs perform predicate calls considerably less often than when they are incrementally compiled. For example, the sequence:

```
        repeat, fail.
```

in incremental code is compiled as: "call *repeat/0*, call *fail/0*". The repeat predicate itself is defined as:

```
repeat.
repeat :- repeat.
```

which, were it in incremental mode, will have been compiled as: "either succeed, or call *repeat/0*". In a fully incrementally compiled system, the "repeat,fail" sequence would perform two predicate calls per backtracking iteration, and *repeat/0* would in turn call itself once, so submitting to Windows approximately every 85th (256/3) cycle.

When optimised, *repeat/0* is compiled using a fast, direct "goto" instruction in place of its second clause, in effect, "either succeed or start over", and in the sequence "repeat,fail", the call to *fail/0* is replaced by an explicit Prolog machine "fail" instruction. The result is that, when fully optimised, "repeat,fail" never yields control to Windows, because no predicate calls are being made.

Running an optimised program which contains "repeat,fail" will cause **LPA-PROLOG** to hang, whether under Windows or DOS. Fortunately, with Win32 and DOS, it is just your Prolog program that hangs, but in the case of Win16, because control is never yielded and Win16 cannot perform preemptive multitasking, the entire Windows session will hang. In the latter case, your only option will be to reboot the computer using <*ctrl-alt-del*>.

The "repeat/fail" operation is obviously a degenerate case, since such a program serves no useful purpose: the more likely "repeat,<do something>,fail" sequence is perfectly alright, since the "<do something>" component will involve at least one predicate call per cycle.

Optimised Tail Recursion

Another type of optimised program can cause problems when optimised, namely a program with a clause of the general form:

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

Again, when incrementally compiled, the code includes a "call foo/n" instruction, able to yield control once every 256 iterations; when optimised, however, a direct "goto" instruction is used in place of the recursive call, in effect, "to do foo, start over", and again **WIN-PROLOG** will not yield control. Provided that there is at least one other predicate call in the program, ie: "foo(...) :- <do something>, foo(...).", then there will be no problem.

Typical cases where problems could arise include standard predicates such as *append/3*, *member/2* and other list processing programs. Once optimised, these programs will not relinquish control during their execution. Programs like *reverse/2* (if implemented as naive reverse) do not pose problems, because they call another predicate (*append/3* in this case) once per iteration.

It is unlikely that anything other than the degenerate program "foo :- foo." will actually cause **WIN-PROLOG** to hang, because unless programs like *append/3* and *member/2* are given extremely large lists, they will complete their processing in a very short time. The worst that normally occurs in programs using such optimised predicates excessively is that Windows feels a little sluggish.

Explicit Yielding of Control

Because **WIN-PROLOG** automatically yields control to Windows on a regular basis, there is normally no need to call an explicit "yield" predicate: there are times, however, when you might want to surrender control more often than the standard once per 256 predicate calls; this section describes how to do this.

Under Windows, the keyboard (console input) device is driven by messages, so **WIN-PROLOG** has to yield control to Windows whenever keyboard input is attempted. This leads to the first method of yielding control: any calls you make to *read/1*, *get/1* and so forth from the *user* device will allow Windows to process other tasks until you confirm your input by pressing <*enter*>.

As well as the *user* device, which is buffered (it waits for a line of input to be typed and then confirmed with <*enter*>), you can also yield control by using the direct keyboard input predicate *getb/1*. This does not wait for <*enter*> to be hit, but still yields control to Windows until you press a key or mouse button.

The third method of yielding control is the most useful in a running application, and involves the *wait/1* predicate. This predicate is designed to yield to windows at least once, and optionally to yield repeatedly until at least one message has been received by **WIN-PROLOG**. Usually it is called with zero (0) as its argument. In "repeat,fail", or other computationally intensive loops, you might consider inserting a call to *wait/1* as follows:

```
foo :-
        repeat,
        wait( 0 ),
        fail.
```

or:

```
foo :-
        wait( 0 ),
        foo.
```

These examples are guaranteed to yield to Windows at least once per iteration, rather than once every 256 predicate calls or perhaps even never (see above). Even in Win32, calling *wait/1* in a tight loop is necessary to reduce processor overhead, and to free the processor for other applications.

File Management

There is no direct support for shared file access in LPA-PROLOG: files are opened in a "deny access" mode, so while a Prolog program is processing any given file, no other applications can use it. Under DOS, this is only ever a consideration when writing network-aware programs, but under Windows, all programs should be written with this in mind.

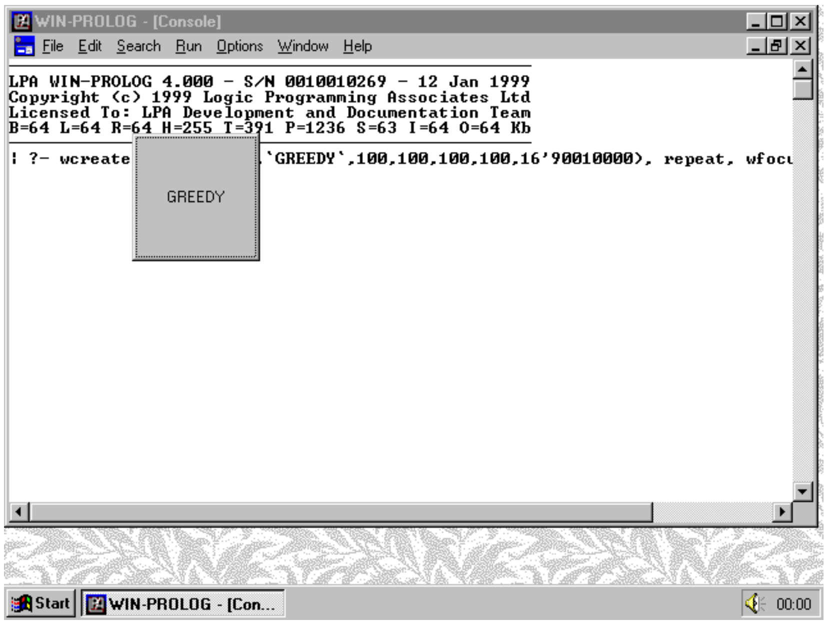
It is recommended that files are opened for as short a time as possible, performing the I/O immediately after opening the file, and then closing the file until next needed. Since LPA-PROLOG has powerful random access pointer predicates, and its string data type (see *Appendix B*) provides the ability quickly to read or write large blocks of data from or to disk files, in most cases there is no need to keep files open for more than a few seconds at a time.

The Keyboard and Focus

Windows only permits keyboard input to one window at a time, and it should be left to the user to decide which window receives attention: focus should not be changed at random by the application. Predicates such as *wfocus/1* and *wshow/2* should therefore be used with discretion, since they switch focus between windows, even if the user is in another application at the time. For example, type the command (as always, type the **bold** characters only):

```
?- wcreate(fred,button,`GREEDY`,100,100,100,100,16'90010000),
repeat, wfocus(fred), fail.                              <enter>
```

The result is a "desktop" button appearing on the screen, as shown in *Fig C.1*.



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

| ?- wcreate                  ,`GREEDY`,100,100,100,100,16'90010000), repeat, wfocu
```

Fig C.1 - Greedily grabbing focus with a desktop button

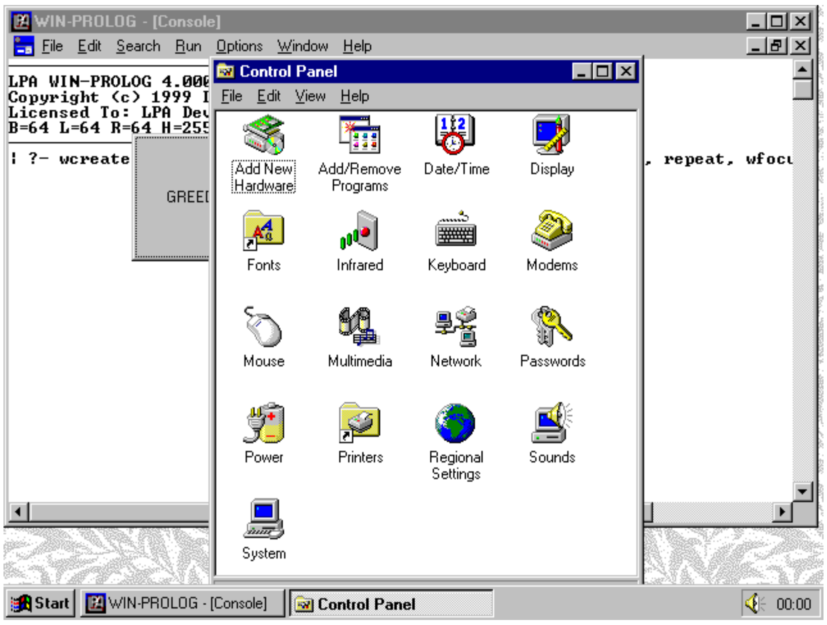
Because each time *wfocus/1* is called within the repeat/fail loop, focus is forced back to the button, it is now virtually impossible to get control back to the ***WIN-PROLOG*** console. Fortunately, in WinNT and Win95/98, it is at least possible to run other applications while this loop is running; for example, try invoking the Windows control panel, as shown in *Fig C.2*.

In Windows 3.1, even if you click on a Program Manager menu item, you will lose control before being able to select an option unless you are very quick. Luckily, ***WIN-PROLOG*** can be interrupted when any of its windows or controls is in focus: press *<ctrl-break>* to stop this program, as shown in *Fig C.3*.

The Mouse and the Cursor

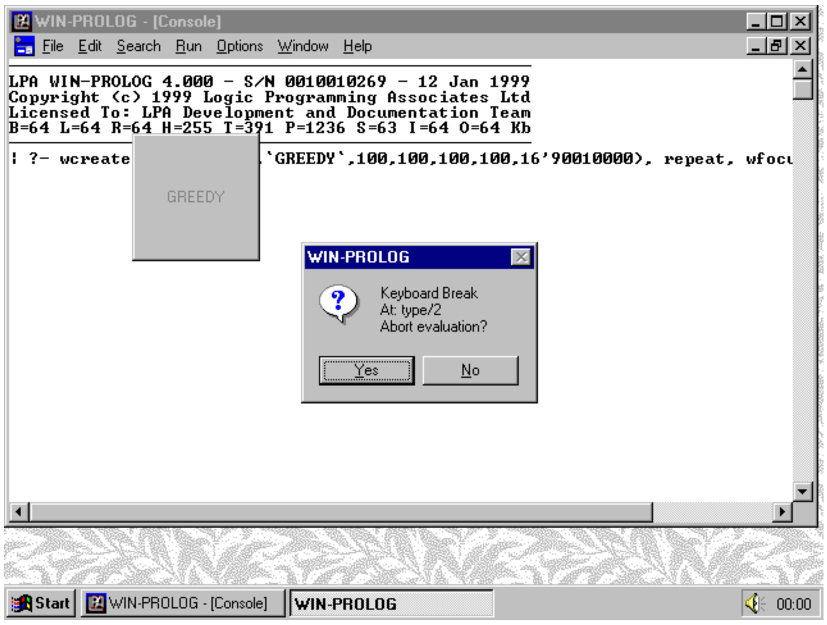
As with focus, Windows only supports one cursor. Normally this is controlled by the mouse, although depending upon the application it might also be controlled by keyboard input. the cursor is normally sensitive to the context in which it is being displayed, but when a Windows application is busy, this behaviour can be overridden, and the cursor forced into a fixed, and inactive hourglass shape. The *busy/1* predicate can be used to create an hourglass (busy) or normal (idle) cursor from within Prolog programs, and you should be aware when the hourglass cursor is enabled, the user cannot switch applications, operate menus, or perform keyboard input.

Like the *wfocus/1* predicate and *???box/n* dialogs, *busy/1* allows you to hijack the cursor when another application has focus. You could, if you wanted to, simply set the hourglass willy-nilly, but a more constructive approach is to wait until you know that ***WIN-PROLOG*** is in control, and then set it. A simple test is to use the fact that windows created by ***WIN-PROLOG*** have handles which are



```
LPA WIN-PROLOG 4.000
Copyright (c) 1999 I
Licensed To: LPA Dev
B=64 L=64 R=64 H=255

| ?- wcreate                                    , repeat, wfocu
```

Fig C.2 - Result of running the Windows control panel



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

¦ ?- wcreate ,`GREEDY`,100,100,100,100,16'90010000), repeat, wfocu
```

Keyboard Break
At: type/2
Abort evaluation?

Fig C.3 - Interrupting the greedy loop with <ctrl-break>

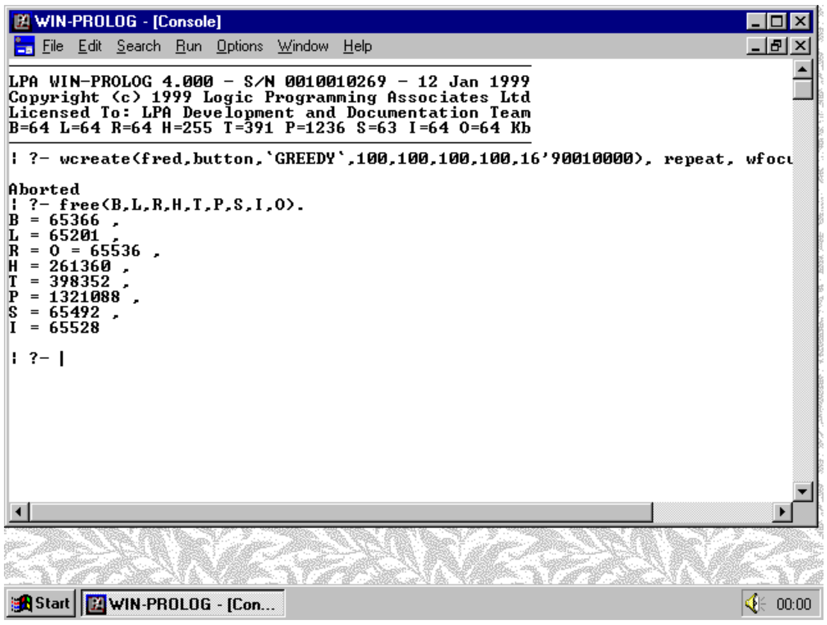
either atoms, or conjunctions of atoms and integer ID codes. Windows created elsewhere than by *WIN-PROLOG* have simple integer handles, and these handles are always greater than the low integers (0 and 1) used by the built-in windows. By retrieving the current focus, and waiting for the returned handle to be either a non-integer, or one of the special values (0 and 1), you could write a program such as the following:

```
wait_for_focus :-
        repeat,
        wait( 0 ),                  % yield every repeat/fail
        wfocus( F ),                % get focus
        (       integer( F )        % if it is an integer
        ->      member( F, [0,1] )  % then is it one of ours?
        ;       true)               % else it is ours anyway
        ),
        !.
```

This would wait for the user to switch focus to one of *WIN-PROLOG*'s windows, and would then return immediately. Calling this program immediately before calls to *busy/1*, *wfocus/1* or *wshow/2* in their "set" modes can help ensure user-friendly behaviour which does not interrupt other applications.

Buffered Console Output

As a console, Windows is extremely powerful, with its rich GUI and many modeless activities, but there is a penalty to be paid: text I/O is slow, in fact many hundreds of times slower than under DOS. The major overhead is on a per-operation basis, rather than on the size or complexity of that operation. Under



```
LPA WIN-PROLOG 4.000 - S/N 0010010269 - 12 Jan 1999
Copyright (c) 1999 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=255 T=391 P=1236 S=63 I=64 O=64 Kb

¦ ?- wcreate(fred,button,`GREEDY`,100,100,100,100,16'90010000), repeat, wfocu

Aborted
¦ ?- free(B,L,R,H,T,P,S,I,O).
B = 65366 ,
L = 65201 ,
R = O = 65536 ,
H = 261360 ,
T = 398352 ,
P = 1321088 ,
S = 65492 ,
I = 65528

¦ ?- |
```

Fig C.4 - Instant display of several lines of output

DOS, each byte that your Prolog program attempts to display on the screen is immediately shown, yet ***DOS-PROLOG*** can achieve output rates approaching 17500 characters of output per second on a 25MHz 386. Under Windows, trying to display each character in turn would limit output to around 25 characters per second on the same machine!

To make the rate of output to the console window acceptable, output is stored in a buffer, which is flushed only occasionally. Even flushing once per line is too slow for comfort, so the buffer is flushed only when full, when a read operation is initiated from the keyboard, or when a program forces a flush. The size of the console output buffer is 1kb (1024 bytes). If you type in a command such as:

```
?- free(B,L,R,H,T,P,S,I,O).                              <enter>
```

you will notice that the output appears all at once, together with the next prompt, as shown in *Fig C.4*. Although fewer than 1024 characters have been output, the call by ***WIN-PROLOG***'s supervisor loop to read your next command forced the output buffer to be flushed.

The only time that you need to be aware of flushing is during console window output which is not going to be followed by a keyboard input operation, such as diagnostic messages, or status reports during a computation. All you need to do to flush the output buffer is to call the *ttyflush/0* predicate, or alternatively to write the end of file character, *<ctrl-Z>* or character code 1ah, to the console window. Create a new program window using the "File/New" menu option, and enter the following code:

```
fast( 0 ).
fast( Number ) :-
        write( Number ),
        nl,
        Less is Number - 1,
        fast( Less ).

slow( 0 ).
slow( Number ) :-
        write( Number ),
        nl,
        ttyflush,
        Less is Number - 1,
        slow( Less ).
```

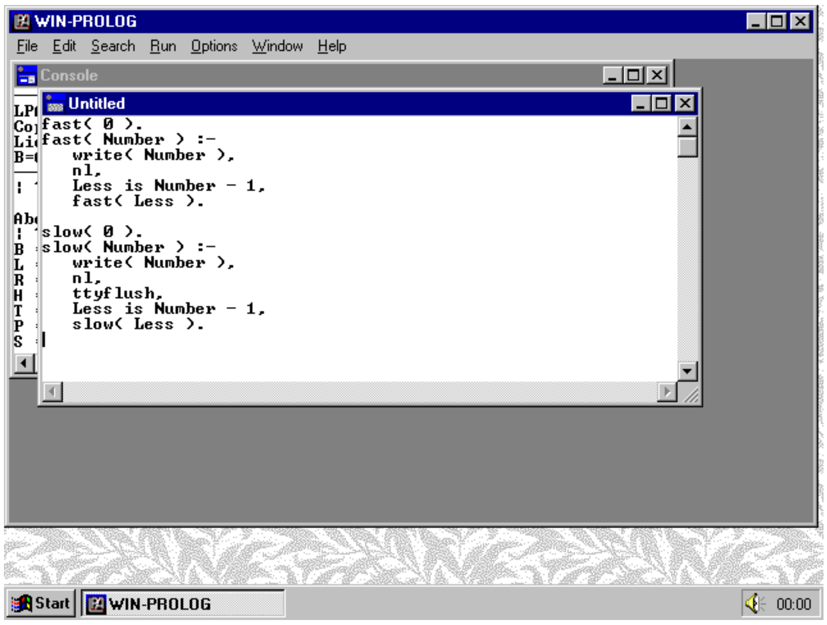
Once you have typed it in, as shown in *Fig C.5*, compile it using the "Run/Compile" menu option, and then click back on the console window. Type in the command:

```
?- fast(10).                                        <enter>
```

You will see 10 numbers appear virtually simultaneously on the screen. Now type the command:

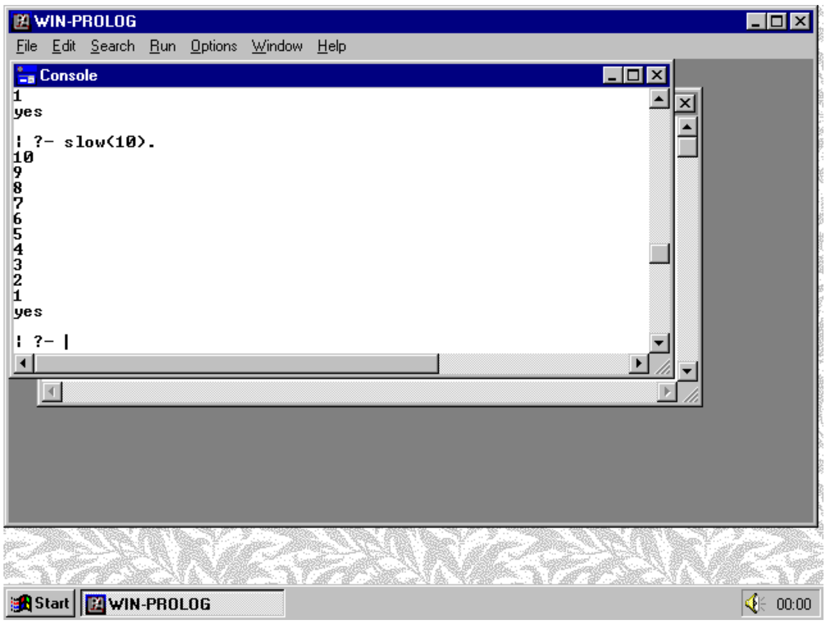
```
?- slow(10).                                        <enter>
```

The same 10 numbers will appear, but much more slowly, because each line of output is being flushed to the console window in turn, as shown in *Fig C.6*.



```
fast( 0 ).
fast( Number ) :-
    write( Number ),
    nl,
    Less is Number - 1,
    fast( Less ).

slow( 0 ).
slow( Number ) :-
    write( Number ),
    nl,
    ttyflush,
    Less is Number - 1,
    slow( Less ).
```

Fig C.5 - Entering the fast/1 and slow/1 programs



```
1
yes

| ?- slow(10).
10
9
8
7
6
5
4
3
2
1
yes

| ?- |
```

Fig C.6 - Output from the fast/1 and slow/1 programs

Executing External Applications

One of the features of WIN-PROLOG is its ability to execute other applications using its *exec/3* predicate, or *dos/n* predicates. Under DOS, the *exec/3* predicate waits for the external program to complete its operations, before returning its numerical exit code. Under Windows, which is a multi-tasking environment, *exec/3* returns control immediately, with an exit code of 0, and the external application continues to run in parallel with WIN-PROLOG.

If the external application is expected to perform some file output, and these files are to be used by a Prolog program, then with WIN-PROLOG, care must be taken to wait for the external program to complete. One way in which to arrange this would be to have the application create a temporary file, with a name such as "ALLDONE.$$$", just before it terminated. You could then use some Prolog code along the following lines to process the application:

```
run_app :-
        exec( 'c:\bin\myapp.exe', '', _ ),
        repeat,
        wait( 0 ),
        dir( 'c:\bin\alldone.$$$', -32, Dir ),
        Dir \= [],
        del( 'c:\bin\alldone.$$$' ),
        !.
```

Of course, there would be nothing to stop your Prolog program from doing some useful work while waiting for the external application to complete.

Appendix D - Window Styles

This appendix discusses the styles of windows, and describes some special purpose predicates which enable you to use symbolic names, rather than 32-bit integers, as the styles of windows you create.

Window Styles: 32-bit Integers

Every window in the Windows environment has an associated "style" field, which consists of 32 bits or flags governing the window's appearance and behaviour. For example, dialog windows can be created with a variety of borders, with or without system menus, maximise/restore and minimise buttons; edit controls can be created single- or multi-line, with or without horizontal and/or vertical scroll bars. Throughout the rest of this manual, various examples have made references to specific style combinations; the full range of window styles supported in Windows is outlined later in this appendix.

Hexadecimal Notation

Because window styles can be combined using the bitwise "or" operation to produce compound styles, it is preferable to describe styles using a non-decimal notation. The one used throughout the rest of this manual is "hexadecimal" (base 16), where the numerals 0..9 and letters A..F refer to the hexadecimal digits 0..15 respectively. Just as the decimal number:

```
    128
```

means "1*(10^2) + 2*(10^1) + 8*(10^0)", so the hexadecimal number:

```
    12A
```

means "1*(16^2) + 2*(16^1) + A*(16^0)", where "A" means "10". In ***LPA-PROLOG***, you can write any integer in any base between 2 (binary) and 36, by using a special notation in which the desired base is written in decimal, followed by an apostrophe, and then the number itself. For example, the hexadecimal number "3B2" would be written:

```
    16'3B2
```

When ***LPA-PROLOG*** encounters "16'" on input, it prepares to read a hexadecimal number, in this case "3B2". Note that the case of letter digits is not significant.

In this example, the number is calculated as "2 + 16 * (B + 16 * (3))", where "B" means "11", giving the integer "946". In fact, were you to type the query:

```
?- X = 16'3B2.                                  <enter>
```

you would get the result "X = 946". Hexadecimal (and other base) notation is simply a convenient way of entering standard integers. The same number could have been entered in "octal" (base 8):

```
?- X = 8'1662.                                  <enter>
```

or even in binary (base 2):

```
?- X = 2'1110110010.                            <enter>
```

It does not matter which base you use for defining styles, because as was noted above, all such bases map directly down onto ordinary Prolog integers; however, the bases 16, 8 and 2 are more convenient than 10 (decimal) when calculating the combination of style values. For example, to combine the styles "WS_CHILD", "WS_VISIBLE" and "WS_THICKFRAME" in decimal would require you to "or" the values:

```
1073741824          (WS_CHILD)
 268435456          (WS_VISIBLE)
    262144          (WS_THICKFRAME
----------
1342439424          (combined style)
```

Using hexadecimal numbers, the combined style could be computed far more easily:

```
16'40000000         (WS_CHILD)
16'10000000         (WS_VISIBLE)
16'00040000         (WS_THICKFRAME)
-----------
16'50040000         (combined style)
```

As you can see, the arithmetic is far simpler in hexadecimal notation, and it is easy to see which styles have been combined in the result. For these reasons, all style values in the following sections are given in hexadecimal notation.

Generic Window Styles

A number of styles can be applied to windows of more than one type, and in some cases, to windows of all types. The names for these "generic" window styles are identified with the prefix "WS_", and are listed in *Table D.1*.

Table D.1 - Generic Window Styles

Symbolic Name	Hexadecimal Value
WS_OVERLAPPED	16'00000000
WS_POPUP	16'80000000
WS_CHILD	16'40000000
WS_CLIPSIBLINGS	16'04000000
WS_CLIPCHILDREN	16'02000000
WS_VISIBLE	16'10000000
WS_DISABLED	16'08000000
WS_MINIMIZE	16'20000000
WS_MAXIMIZE	16'01000000
WS_CAPTION	16'00C00000
WS_BORDER	16'00800000
WS_DLGFRAME	16'00400000
WS_VSCROLL	16'00200000
WS_HSCROLL	16'00100000
WS_SYSMENU	16'00080000
WS_THICKFRAME	16'00040000
WS_MINIMIZEBOX	16'00020000
WS_MAXIMIZEBOX	16'00010000
WS_GROUP	16'00020000
WS_TABSTOP	16'00010000

Button Control Styles

A number of styles apply specifically to "Button" controls. In general, these styles cannot be combined, but rather they enumerate which of a number of specific types of button is created. The names for button styles are identified with the prefix "BS_", and are listed in *Table D.2*.

Table D.2 - Button Control Styles

Symbolic Name	Hexadecimal Value
BS_PUSHBUTTON	16'00000000
BS_DEFPUSHBUTTON	16'00000001
BS_CHECKBOX	16'00000002
BS_AUTOCHECKBOX	16'00000003
BS_RADIOBUTTON	16'00000004
BS_3STATE	16'00000005
BS_AUTO3STATE	16'00000006
BS_GROUPBOX	16'00000007
BS_USERBUTTON	16'00000008
BS_AUTORADIOBUTTON	16'00000009
BS_OWNERDRAW	16'0000000B
BS_LEFTTEXT	16'00000020

Edit Control Styles

A number of styles apply specifically to "Edit" controls. Like the generic styles, these can be combined to produce compound styles. The names for edit styles are identified with the prefix "ES_", and are listed in *Table D.3*.

Table D.3 - Edit Control Styles

Symbolic Name	Hexadecimal Value
ES_LEFT	16'00000000
ES_CENTER	16'00000001
ES_RIGHT	16'00000002
ES_MULTILINE	16'00000004
ES_UPPERCASE	16'00000008
ES_LOWERCASE	16'00000010
ES_PASSWORD	16'00000020
ES_AUTOVSCROLL	16'00000040
ES_AUTOHSCROLL	16'00000080
ES_NOHIDESEL	16'00000100
ES_OEMCONVERT	16'00000400
ES_READONLY	16'00000800
ES_WANTRETURN	16'00001000

Listbox Control Styles

A number of styles apply specifically to "Listbox" controls. Like the generic styles, these can be combined to produce compound styles. The names for listbox styles are identified with the prefix "LBS_", and are listed in *Table D.4*.

Table D.4 - Listbox Control Styles

Symbolic Name	Hexadecimal Value
LBS_NOTIFY	16'00000001
LBS_SORT	16'00000002
LBS_NOREDRAW	16'00000004
LBS_MULTIPLESEL	16'00000008
LBS_OWNERDRAWFIXED	16'00000010
LBS_OWNERDRAWVARIABLE	16'00000020
LBS_HASSTRINGS	16'00000040
LBS_USETABSTOPS	16'00000080
LBS_NOINTEGRALHEIGHT	16'00000100
LBS_MULTICOLUMN	16'00000200
LBS_WANTKEYBOARDINPUT	16'00000400
LBS_EXTENDEDSEL	16'00000800
LBS_DISABLENOSCROLL	16'00001000

Combobox Control Styles

A number of styles apply specifically to "Combobox" controls. Like the generic styles, most of these can be combined to produce compound styles; some, however, are used to enumerate which of several types of combobox is created, and these styles cannot be combined with each other. The names for combobox styles are identified with the prefix "CBS_", and are listed in *Table D.5*.

Table D.5 - Combobox Control Styles

Symbolic Name	Hexadecimal Value
CBS_SIMPLE	16'00000001
CBS_DROPDOWN	16'00000002
CBS_DROPDOWNLIST	16'00000003
CBS_OWNERDRAWFIXED	16'00000010
CBS_OWNERDRAWVARIABLE	16'00000020
CBS_AUTOHSCROLL	16'00000040
CBS_OEMCONVERT	16'00000080
CBS_SORT	16'00000100
CBS_HASSTRINGS	16'00000200
CBS_NOINTEGRALHEIGHT	16'00000400
CBS_DISABLENOSCROLL	16'00000800

Scrollbar Control Styles

A number of styles apply specifically to "Scrollbar" controls. Like the generic styles, these can be combined to produce compound styles. Note that there is duplication in the styles: horizontal and vertical scrollbars use "left/right" and "top/bottom" names respectively, but these map onto the same style values. The names for scrollbar styles are identified with the prefix "SBS_", and are listed in *Table D.6*.

Table D.6 - Scrollbar Control Styles

Symbolic Name	Hexadecimal Value
SBS_HORZ	16'00000000
SBS_VERT	16'00000001
SBS_TOPALIGN	16'00000002
SBS_LEFTALIGN	16'00000002
SBS_BOTTOMALIGN	16'00000004
SBS_RIGHTALIGN	16'00000004
SBS_SIZEBOXTOPLEFTALIGN	16'00000002
SBS_SIZEBOXBOTTOMRIGHTALIGN	16'00000004
SBS_SIZEBOX	16'00000008

Static Control Styles

A number of styles apply specifically to "Static" controls. In general, these styles cannot be combined, but rather they enumerate which of a number of specific types of static control is created. The names for static control styles are identified with the prefix "SS_", and are listed in *Table D.7*.

Table D.7 - Static Control Styles

Symbolic Name	Hexadecimal Value
SS_LEFT	16'00000000
SS_CENTER	16'00000001
SS_RIGHT	16'00000002
SS_ICON	16'00000003
SS_BLACKRECT	16'00000004
SS_GRAYRECT	16'00000005
SS_WHITERECT	16'00000006
SS_BLACKFRAME	16'00000007
SS_GRAYFRAME	16'00000008
SS_WHITEFRAME	16'00000009
SS_SIMPLE	16'0000000B
SS_LEFTNOWORDWRAP	16'0000000C
SS_NOPREFIX	16'00000080

Grafix Control Styles

Unlike the other control types, there are no special window styles which apply specifically to "Grafix" controls. In general, these windows can use any of the generic window styles.

Dialog Pseudostyles

In addition to the generic window styles, three special "pseudostyles" are provided by **WIN-PROLOG** for use when creating dialogs. These are not recognised by Windows itself, but are used to define the ownership of dialogs, and which of two border types to use. The symbolic names for the dialog pseudostyles are identified with the prefix "DLG_", and are listed in *Table D.8*.

Table D.8 - Dialog Pseudostyles

Symbolic Name	Hexadecimal Value
DLG_OWNEDBYDESKTOP	16'00000000
DLG_OWNEDBYPROLOG	16'00000001
DLG_MODALFRAME	16'00000002

Symbolic to Integer Conversion

Even using hexadecimal notation, the computation of compound window styles is a little tricky, and can lead to programming errors. To make life easier, there are two built-in predicates which perform this step for you. The first, *wdcreate/7*, is used to create dialogs. There is no class argument (this would always have had the value "dialog"), and the style can be given in symbolic form. The call:

```
?- wdcreate(fred,`Freddie`,200,250,400,225,[ws_popup,ws_caption]).
                                                          <enter>
```

creates a "dialog" window called "fred", with the given name and size, using the styles "WS_POPUP" and "WS_CAPTION", which compute to produce the hexdecimal style 16'80C00000; this call is directly equivalent to:

```
?- wcreate(fred,dialog,`Freddie`,200,250,400,225,16'80c00000).
                                                          <enter>
```

A similar predicate, *wccreate/8*, is used to create controls. This time, the class name must be included, since there are several different control classes. The call:

```
?- wccreate((fred,1),button,`OK`,10,150,80,32,[ws_child,ws_visible]).
                                                          <enter>
```

creates a button with combined styles "WS_CHILD" and "WS_VISIBLE", which compute to the hexadecimal style, 16'50000000, and is equivalent to:

```
?- wcreate((fred,1),button,`OK`,10,150,80,32,16'50000000).
                                                          <enter>
```

For completeness, and convenience, two more built-in predicates, *wtcreate/6* and *wucreate/6* are included to create "text" and "user" windows respectively. These predicates omit both the class name and the style, the latter not being significant in MDI windows.

Appendix E - Window Classes

This appendix discusses the classes of windows, and describes those which can be created by *WIN-PROLOG*, as well as some of the predicates used to retrieve and manipulate classes.

Window Classes and Pseudoclasses

Every window in the Windows environment belongs to a "class", which describes which basic type of window it is. Classes such as "Button", "Edit" and "Listbox" are predefined in Windows, and provide the framework for the controls of these same types; other classes are defined by applications themselves, and include support for the "text" and "user" windows in *WIN-PROLOG*. When you create a window, you specify the class name as the second argument to *wcreate/8*:

```
?- wcreate((fred,1),button,...).
```

creates a control of the "Button" class within the window "fred". There is, however, one type of window which you create using a "pseudoclass" rather than a class as the second argument. A pseudoclass is simply a descriptive word which is interpreted directly by *WIN-PROLOG*, and which may result in a window being created with a class named completely differently. The only example of a pseudoclass in the current version of *WIN-PROLOG* is the "dialog".

You can retrieve the true class of any window with the *wclass/2* predicate. For example, the call:

```
?- wclass((1,1),X).                                    <enter>
```

will return the value "X = 'Edit'", which is the class of the console window edit control, "(1,1)".

When searching for top-level windows using the *wfind/3* predicate, you can specify the window class as the first argument. If you wish to omit the class name as a search parameter, simply specify an empty atom. Thus the call:

```
?- wfind(foo,`bar`,X).                                 <enter>
```

will search the desktop for a window of class "foo", and name "`bar`", while:

```
?- wfind('',`bar`,X).                                  <enter>
```

will simply search for a window named "`bar`", without checking for any particular class.

User Window Classes and Pseudoclasses

There are nine true window classes and one pseudoclass which can be used when creating windows in **WIN-PROLOG**, with each one corresponding to one of the two types of MDI window, the dialog window class, or the seven types of control window, as listed in *Table E.1*.

Table E.1 - The Window Classes and Pseudoclasses

Class	Description
text	mdi child window with built-in edit control
user	mdi child window with no controls
grafix	WIN-PROLOG-defined graphics window
dialog	pseudoclass describing dialog windows
button	push buttons, radio buttons, checkboxes
combobox	edit control with single choice list box
edit	all types of edit control
listbox	single or multiple choice list box
scrollbar	scroll bar
static	static text or icon window

Predefined Window and Pseudoclass Classes

Each of the predefined windows in **WIN-PROLOG** has an associated class name, as does each pseudoclass. These class names should not be used when creating windows, since the results will be unpredictable, but are listed in *Tables E.3* and *E.4* for the sake of completeness.

Table E.3 - Classes of Predefined Windows

Window	Class	Comment
0	Main	WIN-PROLOG main window
(0,1)	MDIClient	WIN-PROLOG MDI client window
1	Cons	WIN-PROLOG console window
(1,1)	Edit	WIN-PROLOG console edit control

Table E.4 - Classes of Pseudoclasses

Pseudoclass	Class	Comment
dialog	#32770	Windows class name for "dialog" window

Appendix F - Initialisation Files

This appendix discusses the use of initialisation files, and describes a special predicate, *profile/4*, which enables you to read and write entries in such files.

Persistent Data: the .INI File

Most Windows applications maintain certain types of data between sessions, such as the size, position and state of each window, or the user's preferred font, language and backup mode. Such information is stored in "initialisation" files, which usually have the extension ".INI". The best known .INI file of them all is the "WIN.INI" file which Windows maintains, and which contains dozens of entries covering everything from mouse sensitivity to currency settings, sound event assignments to desktop wallpaper, and so forth.

While many applications add their own entries to WIN.INI, it is increasingly the norm for programs to maintain their own, private initialisation files. If you look in the typical Windows directory, you might see, alongside WIN.INI, files like CLOCK.INI, CONTROL.INI, MOUSE.INI and so forth. Each contains initialisation data for the application file with the corresponding ".EXE" name.

Files, Sections and Entries

Any given piece of initialisation data can uniquely be identified by a combination of three items of information: its corresponding file, section and entry names, as outlined in the next three paragraphs.

Typically, the file is either WIN.INI, or is named after the application: for example, in an application called "FOO", where the main executable file was "FOO.EXE", the .INI file would reside in the same directory as FOO.EXE, and would be called "FOO.INI".

The section is a portion of the file labelled by a word in square brackets ([...]), and is used to group pieces of information about a particular subject, for example "printing", "colours", and so forth. The section titles are entirely at the discretion of the application.

The entry is a particular item of information, and is of the form: "name=value". As with section titles, the names and values, are entirely at the discretion of the application. Below is a sample portion from a .INI file:

```
[printer]
current=hp550c
pagesize=a4
font=courier,12,0
```

```
[screen]
current=vga
font=oem,12,0
```

There are two sections, labelled "printer" and "screen". Both have "current" and "font" entries, and the printer section also has a "pagesize" entry. The meaning of this information is entirely up to the application which created and maintains the .INI file: there are no reserved words or special system commands here.

Reading and Writing Data in .INI files

You can read data from, and write data to, .INI files, using the *profile/4* predicate. This predicate completely bypasses **WIN-PROLOG**'s normal file handling, and hands your I/O requests directly over to special Windows functions. The first three arguments of *profile/4* are the file, section and entry values described above, and may be specified as atoms or strings. The fourth argument may be a variable, to return the existing value, or an atom or string to set a new one. Suppose, for example, that the sequence above was stored in a file called "FOO.INI". You could pick up the current screen type by making the call:

```
?- profile('foo.ini',screen,current,X).                <enter>
```

This would return the value "X = vga". Similarly, you could set a new page size for the printer with the call:

```
?- profile('foo.ini',printer,pagesize,letter).         <enter>
```

This would replace the "pagesize=a4" entry with a new one stating "pagesize=letter". Again, it should be stressed that these entries are meaningless until interpreted by an application. So long as you do not alter the .INI files of other applications, you can experiment freely with the *profile/4* predicate.

Note that the file, section and entry names are not case sensitive: you can use any mix of upper and lower case letters when specifying them. If a given file, section or entry name does not exist, then an attempt to read it will return the empty atom or string: such a result should be treated as failure.

Creating .INI Files

It is not necessary to create .INI files explicitly, since Windows will automatically create one the first time you want to write to one. Thus the call:

```
?- profile(temp,foo,bar,sux).                          <enter>
```

will create a new file called "temp" (if one is not already there) in the Windows home directory, with the following contents:

```
[foo]
bar=sux
```

Note that the ".INI" extension is not automatically put on the file name, and also that the default location for a new initialisation file is in the Windows home directory. In general, you should maintain .INI files in your application's home directory, and the following **WIN-PROLOG** program can be used to generate the appropriate file name:

```
ini_file_name( Ini ) :-
      fname( [], Path, Name, _ ),
      cat( [Path,Name,'.INI'], Ini, _ ).
```

The call to *fname/4* with an empty list as its first argument returns the path, name and extension components of the full name of the .EXE file currently running. Depending upon your choice of **WIN-PROLOG** home directory, this call will return a result such as "'C:\PRO386W\'", "'PRO386W'" and "'.EXE'". The second line calls *cat/3* to join the path and name to the new ".INI" extension, which would return a value such as "'C:\PRO386W\PRO386W.INI'".

The important thing about the above program is that if you create a stand-alone application in a home directory called "D:\FOODIR", and rename the **WIN-PROLOG** executable to "FOO.EXE", then this program will return the correct .INI file name, namely "D:\FOODIR\FOO.INI".

Creating and Deleting Sections and Entries

Just as a .INI file is created automatically the first time you try to write to it, so new sections and entries are added the first time you write them: in fact, you saw this with the "profile(temp,foo,bar,sux)" example above. When you want to change an entry, simply specify the section and entry name, together with its new value. The following call:

```
?- profile(temp,foo,bar,you).                              <enter>
```

would replace the existing "bar" entry from the "foo" section in "temp", leaving the file as follows:

```
[foo]
bar=you
```

You can remove an entry simply by giving an empty atom or string as the new value. Thus the call:

```
?- profile(temp,foo,bar,'').                               <enter>
```

would remove the "bar" entry from the "foo" section, leaving just the following:

```
[foo]
```

Similarly, you can remove a section simply by giving empty atoms or strings for both the entry and value fields:

```
?- profile(temp,foo,'','').                                  <enter>
```

would remove the "foo" section, and all entries it contained, from the file "temp".

Should you wish to delete the .INI file itself, simply use the *del/1* predicate, but remember that you will need to specify the full pathname of the file. Depending upon the name and whereabouts of Windows' home directory, a call such as the following will remove the file:

```
?- del('c:\windows\temp').                                   <enter>
```

Shared .INI File Etiquette

As discussed above, it is generally recommended that applications store their persistent data in their own, private .INI files; however, under some circumstances, it may be desirable to place such information centrally. Normally, applications are free to add to the WIN.INI file, but they must be careful not to overwrite or otherwise modify sections belonging to other applications or to Windows itself. The best solution, apart, that is, from using private .INI files, is to reserve a section in WIN.INI by using the application name. For example, an application called "FooDraw 2.0" might keep all of its central persistent data in a section called, simply, "[FooDraw 2.0]". To add a new item to this section (and create the section itself if not already there), you might use a call such as:

```
?- profile('win.ini','FooDraw 2.0',linewidth,'16').          <enter>
```

This would add the following section and heading to WIN.INI:

```
[FooDraw 2.0]
linewidth=16
```

If you look at your system's WIN.INI, you will see several such entries belonging to programs such as Netscape, Adobe PageMaker and CorelDRAW!, depending upon which applications you use.

Note that Windows places an upper limit of around 64 kilobytes on the size of .INI files, and this is another reason to use private, rather than shared, .INI files in your applications whenever possible.

Appendix G - The Win32 API Predicates

This appendix discusses the use of a very powerful pair of predicates, including *winapi/4* and *wintxt/4*, which between them enable virtually any function in the Win32 API function or in any 32-bit DLL to be called directly from WIN-PROLOG.

The Win32 Application Programming Interface (API)

Version 4.1 of WIN-PROLOG introduced full support for *Unicode*, and one result of this is that the old *winapi/3* and *wintxt/3* predicates were replaced by *winapi/4* and *wintxt/4* respectively. The new predicates differ from their forbears principally in the presence of a new third argument, which is an integer flag defining which of several character encodings to use when passing literal text parameters. From this version onwards, all strings and atoms are comprised of 32-bit characters, rather than of 8-bit bytes as previously; characters which fall within the Unicode range are assumed to map to their Unicode meanings, so that (for example) character code 2122h (8482 decimal) represents the "Trade Mark" symbol (™).

In order to pass the "™" character to a Windows *ANSI* function (generally one with an "A" suffix - see below), this character must be converted to its 8-bit ANSI equivalent, 99h (153 decimal): setting the third argument of either*winapi/4* and *wintxt/4* to one ("1") will perform this conversion automatically as needed. In order to pass "™" to a Windows *WIDE* function (generally one with a "W" suffix - see below), this character must be not be converted, but rather passed as a 16-bit Unicode value: setting the third argument of either *winapi/4* and *wintxt/4* to two ("2") will do this. See *Appendix L* in the *Technical Reference* for a more detailed discussion of Character Encodings.

Specifying the Module and Function

The module and function are combined into a single comma pair, and the second output value (the old fifth argument) is lost. The following call creates the same message box as the above example:

```
?-winapi( (user32,'MessageBoxA'), [0,`world`,`hello`,0)], 1, R ).
                                                          <enter>
```

Note that function names (but not module names) are case sensitive, and that in this particular example, the function name terminates in "A": this suffix exists for all ANSI text functions. Windows NT additionally supports WIDE text functions, in which case the final "A" is replaced by a "W":

```
?-winapi( (user32,'MessageBoxW'), [0,`world`,`hello`,0)], 2, R ).
                                                          <enter>
```

Optimising the Function Lookup

Each time *winapi/4* is called, both the module name and the function within it have to be searched for by the Win32 API. You can bypass this lookup by supplying the desired procedure address already dereferenced. The call:

```
?-winapi( (kernel32,'GetModuleHandleA'), [`user32`], 1, M ),
      winapi( (kernel32,'GetProcAddress'), [M,`MessageBoxA`], 1, A),
      winapi( A, [0,`world`,`hello`,0)], 1, R ).          <enter>
```

will display the same message box as the previous example. By storing the address ("A"), and using this next time a message box is required, the lookup time for "(user32,'MessageBoxA')" can be removed.

Data Buffers

Version 4.0 of WIN-PROLOG introduced a new type of I/O object, the memory file (previous Win32 releases had simple buffers). These can be created, deleted, indexed, and generally manipulated exactly like disk files, windows and other similar objects. From version 4.1 of WIN-PROLOG onwards, these memory can be created in any of eleven different character encodings. The call:

```
?- fcreate( fred, [], -2, 0, 1000000 ).          <enter>
```

uses the *fcreate/5* predicate to create a zero-initialised memory file, known as "fred", of at least 1,000,000 bytes size, in "*ISO*" encoding (the 8-bit *ISO/IEC 8859-1* encoding, which maps onto the first 256 characters of Unicode). A memory file has been created, rather than a disk file, because the file name argument has been set to an empty list ([]), rather than an atom specifying a valid file name. You can find out about any memory file with the *fdata/5* predicate, for example in the call:

```
?- fdata( fred, Address, Mode, Encoding, Size ).          <enter>
```

which will return:

```
Address = (123,456)
Mode = 2
Encoding = 0
Size = 1000000
```

The address comprises two integers in a comma pair: the second can generally be ignored (it contains the data segment selector); the first, which is the offset address of the file's buffer, might be of interest to some functions. More to the point, you can use this argument simply to confirm that the given file ("fred" in this case) is a memory file, rather than a disk file: with the latter, this predicate returns the file name, rather than an (offset,segment) address.

The mode argument simply specifies the read/write mode of the memory file, when used with normal file I/O operations. It has no bearing on its use with the *wintxt/4* predicate (see below). Finally, the size returned will always be at least as big as the amount you specified in the call to *fcreate/5*, though Windows may add extra padding for its own purposes. As well as using a memory file with all the normal input/output predicates, you can store text in it directly with a call to the *wintxt/4* predicate such as:

```
?- wintxt( fred, 0, 1, `The quick brown fox` ).                <enter>
```

The value "0" indicates that the string is to be treated as "*null-terminated*" text, and the "1" specifies that any Unicode characters in the string should first be converted to their local ANSI equivalents: this is the default data type used by Windows "A" (ANSI) text functions. Note that the earlier *winapi/3* and *wintxt/3* predicate replaced any NULL (zero) characters in the given string with spaces: neither of the new predicates,*winapi/4* and*wintxt/4*, do this: instead, strings are passed literally. The call:

```
?- wintxt( fred, 0, 1, Text ).                                  <enter>
```

will return:

```
Text = `The quick brown fox`
```

You can also specify a character count, in which case the string will be treated as binary data, with no substitution of null bytes with spaces. You could create a structure containing a Win32 "RECT" (rectangle) object with a call such as:

```
?- read( Rect ),
      Rect = [Left,Top,Right,Bottom],
      (      putx( 4, Left ),
             putx( 4, Top ),
             putx( 4, Right ),
             putx( 4, Bottom )
      )  ~> String,
      wintxt( fred, 16, 0, String ).                            <enter>
```

Here, because an explict count is given (16), the data is passed as binary. If no count is specified, any null bytes in the data are replaced with spaces, invalidating the information. Binary data can be read back just as easily:

```
?- wintxt( fred, 16, 0, String ),
      (      getx( 4, Left ),
             getx( 4, Top ),
             getx( 4, Right ),
             getx( 4, Bottom )
      ) <~ String,
        Rect = [Left,Top,Right,Bottom],
        write( Rect ).                                    <enter>
```

When storing in a buffer, if the given count is less than the length of the specified string, only <count> characters are copied. If the count is greater than the length of the string, the extra bytes are initialised to NULL (zero) values. The call:

```
    ?- wintxt( fred, 1024, 0, `` ).                       <enter>
```

is a quick way of setting the first 1024 bytes of buffer "fred" to zero, which can be handy when initialising data structures. A list of currently open files can be picked up with the *fdict/1* predicate:

```
    ?- fdict( Files ).                                    <enter>
```

which would return a list, such as:

```
    Files = [fred]
```

You can check whether any given file is a real disk file, or a memory file, by checking whether the "filename" returned by *fdata/5* is an atom or "(offset/ segment)" comma pair (disk file name or memory file address respectively). When finished with, memory files can be deleted with the *fclose/1* predicate:

```
    ?- fclose( fred ).                                    <enter>
```

Using Memory Files with Win32 API

Memory files are all very well in their own right, but are far more useful in conjunction with the *winapi/4* predicate. The "MessageBox" example above shows two data types being passed to Windows (integers and strings). The third type which can be passed is an atom which is the name of a memory file:

```
    ?- fcreate( message, [], -2, 0, 1024 ),
        wintxt( message, 0, 1, `world` ),
        winapi( (user32,'MessageBoxA'), [0,message,`hello`,0)], 0, R ),
        fclose( message ).                                <enter>
```

creates a 1024-byte memory file called "message", stores text in it using Unicode to ANSI text conversion, and then passes the buffer name to *winapi/4* before closing the buffer to restore the memory. The*winapi/4* predicate simply pushes the buffer address onto the stack for use by the Win32 API.

Although some Win32 functions return integers, and others return text pointers, there is no type cast on the function name in *winapi/4*. Instead, all calls return a 32-bit integer to Prolog. If your application knows that this integer is in fact a text pointer, it is simply a matter of calling *wintxt/4* to retrieve that text:

```
?-winapi( (kernel32,lstrcat), [`abc`,`def`], 1, R ), wintxt( R, 0, 1, T ).
                                                                <enter>
```

would return:

```
R = 2468091
T = `abcdef`
```

The value of "R" is not important, but is the address of the string resulting from the call to "lstrcat": the *wintxt/4* predicate "trusts" you to give it a valid address as its first argument, which may be an atom (a memory file name) or an integer (a memory address).

Named Memory Addresses

If a *winapi/4* function call returns an address, as above, and you want to keep a reference to this address, you can call *fcreate/5* to define it as a memory file. Simply pass the known address in as the "file name", rather than an empty list. For example, this call obtains a pointer to the Windows "environment variables", and then creates a memory file to address these:

```
?-winapi( (kernel32,'GetEnvironmentStringsA'), [], R ),
      fcreate( env, [], 0, 0, 1024 ).                    <enter>
```

Here, the (slightly dangerous) assumption is made that the returned address points to a memory area with a length of 1024 characters. A mode of "0", rather than "-2", has been used to declare this memory "read only" for file input/output, and to prevent this memory from being initialised to zero. A call:

```
?- wintxt( env, 0, 0, First ).                           <enter>
```

will return "First" bound to the first environment variable, for example:

```
First = `COMPSPEC=C:\COMMAND.COM`
```

Because "env" has been declared as a read-only memory file, it can be used with *input/1* and other file input predicates: although not particularly useful in this case, there are many applications for this powerful feature.

Indexed Memory File Addressing

As well as accepting memory file names (atoms) and addresses (integers), both *winapi/4* and *wintxt/4* accept a comma pair of a memory file name or address and an offset into the buffer. The call:

```
?-fcreate( fred, [], -2, 0, 1024 ),
      wintxt( fred, 0, 1, `The quick brown fox` ),
          wintxt( (fred,10), 0, 1, Text ).                 <enter>
```

will return:

```
     Text = `brown fox`
```

Similarly, following this with the call:

```
     ?- wintxt( (fred,10), 0, 1, `and the dead` ),
          wintxt( fred, 0, 1, Text ).                      <enter>
```

will return:

```
     Text = `The quick and the dead`
```

With named memory files, the size of the buffer is known, and **WIN-PROLOG** automatically checks to make sure that you do not write beyond its end. With integer addresses, however, the size is unknown, and any attempt to write to such a buffer should be handled with extreme care to avoid corrupting memory or generating general protection faults (GPFs).

The Internal Buffer

One special, pre-defined buffer exists, and is used by many of the built-in predicates. Its name is "[]" (the empty list), so it cannot be mistaken for a memory file. Its size is fixed at 64kb, and it can be used in calls to *winapi/4* and *wintxt/4*, but unlike the memory files described elsewhere in this appendix, it cannot be used in conjunction with file input/output. It has one additional property which is very important: it maintains an end-of-buffer pointer, which is updated each time the buffer is written to. A special call:

```
     ?- wintxt( [], -1, -1, Text ).                        <enter>
```

returns the contents of the buffer up to the end marker, in binary mode. This allows the buffer to be saved and restored around its re-use during a predicate:

```
     foo :-
          wintxt( [], -1, -1, Text ),    % save current value of buffer
          wintxt( [], 0, `World` ), % store new data in buffer
          ...                            % use the data and buffer as required
          wintxt( [], -1, -1, Text ).    % restore original value of buffer
```

This model ensures that if the predicate which calls "foo/0" itself uses the internal buffer, the contents will not be corrupted during the call.

The *winapi/4* predicate also uses the internal buffer for its on-the-fly allocation of text space, but always begins using the buffer from the first "free" byte after the end marker. If the internal buffer is full, for example after the call:

```
?-wintxt( [], 65535, 0, `` ).                                   <enter>
```

then the *winapi/4* predicate will be unable to handle in-line text items (strings). It is generally a good idea to set the internal buffer to empty before calling*winapi/4* in such a way:

```
bar :-
      wintxt( [], -1, -1, Text ),   % save current value of buffer
      wintxt( [], -1, -1, `` ),     % empty the buffer
      winapi( (user32,'MessageBoxA'), [0,`world`,`hello`,0)], Button ),
      wintxt( [], -1, -1, Text ).   % restore value of buffer
```

Note that the above precaution is necessary only if an application is likely to be using most of the internal buffer at the point it calls your routine. Currently, none of the Prolog-implemented Windows predicates directly use more than about half of the internal buffer, so there is always a reasonable amount free for*winapi/4* text parameters. If greater amounts of text are anticipated, it would be a good idea to place the text in named buffers declared specially for the purpose.

The special end-of-buffer pointer is maintainted only for the internal buffer ("[]"), and not for named buffers. With the latter, the same care should be taken with updating their contents as would be with any global assignable data object (such as a file or dynamic predicate).

Examples

This section presents some examples to illustrate various uses of*winapi/4*. The first of these uses the "GetWindowText" function to retrieve the title of the main window using the internal buffer "[]" as storage:

```
?-wndhdl( 0, Handle ),
      winapi( (user32,'GetWindowTextA'), [Handle,[],256], 1, Count ),
      wintxt( [], 0, 1, Text ).                                 <enter>
```

With this call, the following data are returned:

```
Handle = 5828
Count = 22
Text = `WIN-PROLOG - [Console]`
```

Note that this example does not preserve the contents of the internal buffer (see above). The next example calls the "GetWindowRect" function to return the size of a window, again using the internal buffer ("[]") for storage:

```
?- wndhdl( 0, Handle ),
          winapi( (user32,'GetWindowRect'),[Handle,[]], 0, _),
          wintxt( [], 16, 0, Data ),
          (       getx( 4, Left ),
                  getx( 4, Top ),
                  getx( 4, Right ),
                  getx( 4, Bottom )
          )  <~ Data.
```

This call would return:

```
     Handle = 5828
     Data = `~@~@~@~@~@~@~@~@|~B~@~@˜~A~@~@`
     Left = 0
     Top = 0
     Right = 636
     Bottom = 408
```

Once again, note that this example does not preserve the contents of the internal buffer.

The winapi/4 and wintxt/4 Predicates

In this final section, we look in slightly more detail at the Win32 API and text predicates. Both make use of "buffer address" in their arguments, and for the sake of brevity, this will be known as the "buf" data type. Possible instances of this are listed in *Table G.1*:

Table G.1 - Buffer Address Data Types

Name/Type	Description
atom	named buffer
[]	internal buffer
integer	address buffer
(atom,integer)	indexed entry in named buffer
([],integer)	indexed entry in internal buffer
(integer,integer)	indexed entry in address buffer

The *winapi/4* predicate takes four arguments, respectively these comprise the module/function name, a list of parameters, the character encoding flag and the return value. The function may be specified by a conjunction of two atoms, the first being the module name (not case sensitive) and the second the function name (case sensitive and character set dependent), or it may be the address of a function that has been computed elsewhere, as shown previously. The list of parameters can consist of any mix of the data items show in *Table G.2*:

Table G.2 - Win32 API Argument Types

Type	Description
integer	32-bit integer parameter
string	32-bit pointer to null-terminated string
"buf"	32-bit pointer to given buffer
atom	32-bit pointer to named buffer

The character encoding flag specifies in which of four formats to encode the 32-bit characters of **WIN-PROLOG** strings in memory, as shown in *Table G.3*:

Table G.3 - Win32 Character Encodings

Encoding	Description
0	8-bit text passed directly in ISO/IEC 8859-1 format
1	16-bit text converted to/from ANSI in memory
2	16-bit text passed directly in Windows WIDE format
3	32-bit text passed directly to custom DLLs

The return code is always a 32-bit integer, and *wintxt/4* can be used to convert returned text pointers to **LPA-PROLOG** strings or to retrieve data from returned structures or pointer to structures.

The predicate which passes data to and from memory files is *wintxt/4*, whose arguments include a "buf" item, the desired size of transfer, and a variable or string for reading or writing of the buffer respectively. Where the size is specified as "0" (zero), it is assumed that the string is to be treated as "null-terminated", meaning that it will be read up to (but not including) the first null (zero) byte, or written with embedded null bytes replaced by spaces and appended with a null byte. This is the format of string used by C/C++ in general and the Win32 API in particular. For binary data, it is essential to specify the size. With the internal buffer, a special size of "-1" is used to read or write the exact number of bytes in the buffer.

Index