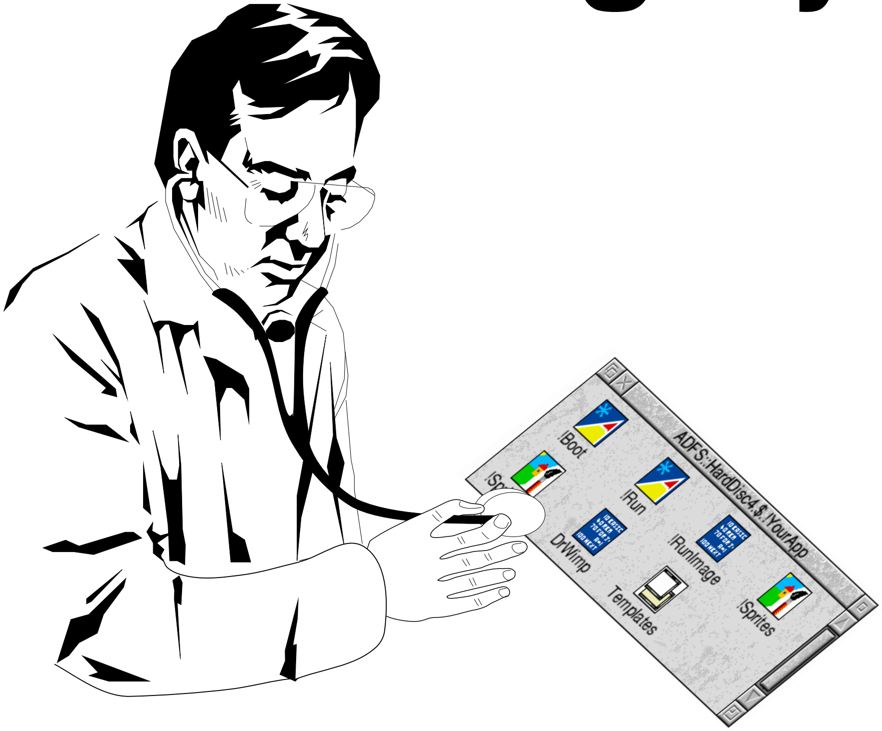
Dr Wimp's Surgery



Ray Favre

Dr Wimp's Surgery

An introduction to Wimp programming and the 'Dr Wimp' package

(for Acorn RISC OS computers)

Ray Favre

Second Edition 2003

As with “Starting Basic” (still available!) this book has been produced for charity. That is, all excess of income over costs actually paid to third parties will go to charity.

The main charity is the Norfolk and Norwich Children’s Fund, attached to the Norfolk and Norwich Hospital. I have no association with this Fund at all, but have been happy to pass on over £2000 to it so far from the sales of both books.

I gratefully acknowledge the encouragement and assistance given by Andrew Ayre, the original author of the ‘Dr Wimp’ package.

The ‘Dr Wimp’ package is Public Domain (PD) and includes several other PD items by other authors - most of whom I couldn’t contact. Because of the above-stated charity arrangements I believe (and hope!) that I am not transgressing any of their individual distribution conditions and I gratefully acknowledge their contribution to this exercise.

Thanks are also due to Eddie Lord who made helpful suggestions on layout etc.

Ray Favre

March 2003

First edition July 1998

Second edition March 2003

This book comes with floppy discs (or a CD-ROM) containing:

- all the tutorial applications developed in the book, including complete successive versions of the main tutorial application built up over Chapters 4-16.
- the complete ‘Dr Wimp’ package (latest version available)
- for reference purposes, the version of the ‘Dr Wimp’ package on which this book is based (Version 3.80)
- a ‘de-archiver’ application (!SparkPlug) to allow the above items to be ‘unpacked’.

De-archiving procedure (A hard disc is necessary.)

Load the floppy disc (or CD-ROM) provided with the book into your drive and open its window.

Double-click on !SparkPlug to load it onto the iconbar.

Open a suitable directory window on your hard disc and ‘drag copy’ the rest of the floppy disc (or CD-ROM) contents to it.

In turn, double-click on each of the copied items which are archived. For each one, a conventional-looking window will open with various items in it.

Drag any or all of these items to your hard disc directory window. This will cause de-archived copies to be saved to your hard disc. Use these hard disc versions as your working copies.

Close/dismount the floppy disc (or CD-ROM) and keep it in a safe place as backup.

With effect from 1st May 1999 the support of the Dr Wimp package was taken over by Ray Favre - the author of this book.

The latest version of the package can be downloaded from the web site at the address on the previous page.

Foreword

Nearly three years ago, I was persuaded by the late Jim Nagel to help him resurrect this book. Apparently, after Ray's death, the original files had disappeared and also the A4 printing masters. Using an OCR program I produced two versions of a sample chapter and sent them to him.

Though pleased with the samples, he died in the spring of 2021 before he could give me the go-ahead and the project fell into abeyance. A year later I began to look at what would be involved in recreating the whole book but before I finished I decided to find out whether or not I had the right to release it. Thanks to Martin Avison, I discovered that the rights to all of Ray's work had passed to the editor of *Archive* magazine. These rights were vested in Jim (which he never mentioned) and are now passed to Gavin Smith who picked up the reins as the editor of *Archive*.

After contacting Gavin, I carried on and finished the project with his blessing. Initially, I recreated the second edition of the book as closely as was reasonable. It is the intention of both Gavin and me that this should be an on-going project. Further editions will be made available as PDF files from the Archive web site. There will be no charge for these downloads.

Gavin has indicated to me that it might be possible to publish the book as hard-copy and to donate any profit in the same way that Ray did.

I am not a programmer and so cannot maintain the program, Dr Wimp. Chris Johnson has carried out some under-the-hood bug fixes and the software has reached version 5.02 as of 1st January 2012. At the time of writing (May 2022) the program can be downloaded from Ray's website at http://drwimp.riscos.org/!home.php.

Updating the book is a long-term project and I would welcome help in the form of suggestions for improvements, correction of errors and proof-reading.

John McCartney
July 2022

Website: http://www.archivemag.co.uk
Email: docwimp@gmail.com

How close to the original?

I have made every effort to stick close to the original look of the book. Given that I haven't been able to accurately identify the page metrics or the font used, there are bound to be differences.

Necessarily, there has been an impact on pagination and I have tried to make any adjustments be limited to within a chapter so that the next chapter starts on the same page as the original.

Ray set the entire book in Impression with the option to use smart quotes set. As a result any plain quotes in a program listing or a program fragment are incorrectly shown as “ (opening) or ” (closing) which, in Courier, appear to be the same: `“` (opening) or `”` (closing). I have chosen to leave them as such for this new version of the second edition. It will be corrected for later editions.

Ray overlooked a reference needed on page 297. He wanted to refer to an illustration in an earlier chapter and left the reference as (Page ???). I have corrected this to (Page 84).

There have been a (very) few corrections to spellings/punctuation. If any more are found, I would be obliged if they can be fed back to me so that I can incorporate corrections in later editions.

Contents

Introduction

This book is for users of RISCOS computers and aims to serve two main purposes.

Its first intention is to provide a comprehensive practical guide to using the Dr Wimp package for producing Wimp programs, for anyone reasonably familiar with BBC Basic V (or VI).

Its second intention is to give a good introduction to the general nature of Wimp programming in Basic.

These two aims are not contradictory, particularly as there are several fundamental Wimp topics which need to be learned whether or not Dr Wimp is used e.g. window template editor, button types, validation strings. Such items are covered in detail in this book and thus provide a useful guide to all wishing to write Wimp programs.

In fact, as it is hoped you will find, the more you learn about Wimp programming in general, the further you will be able to benefit from using Dr Wimp - and the more you use Dr Wimp, the more you will learn about Wimp programming in general.

The hoped-for result is that, however little or much you know about Wimp programming in general, you will want to use Dr Wimp for its sheer all-round convenience.

As already indicated, it is assumed the reader is reasonably adept in using the BBC Basic programming language. But if you are not, the author's companion book "Starting Basic" is still available. *(In many ways, this current book is the natural sequel to "Starting Basic".)*

Book sequence

The book starts with an introduction to the nature of Wimp programming in general, with particular emphasis on the Wimp Poll, Reason Codes and SYS (SWI) calls.

It then asks the question “Why use Dr Wimp?” and seeks to answer this in a very practical way by presenting an example of a simple Wimp application produced in three different ways:

a) ‘from scratch’, without using any libraries or other utilities;

b) as in a) but using window definition templates;

c) using Dr Wimp.

Then, after making sure you are familiar with what the Dr Wimp package contains and how to get the latest version, the preparations for using Dr Wimp in earnest are covered. Several subsequent chapters adopt a tutorial style to build progressively a real Wimp application using the main features of the Dr Wimp package.

Once the tutorial application is completed, later chapters ensure that a fuller range of the Dr Wimp package is adequately explored.

Appendices cover reference material mainly. One gives a detailed practical guide to using a window template editor.

If you are really anxious to get straight into Dr Wimp then you could skip directly to Chapter 3………

Program listings and discs

Throughout the book, many program extracts are listed, but it is a fact of life that full listings of most Wimp programs are simply too long for sensible printing in a book.

Thus, sufficient listings are printed to get the points across and successive full listings of the tutorial are on the supplied floppy discs (or CD-ROM). **It is strongly recommended that you make a ‘working copy’ of these listings and keep the original in a safe place as backup. This is**

particularly relevant if you intend to follow the tutorial sequence 'at the keyboard'. (Some files on the disc have been deliberately locked to reinforce this point. Take a copy and unlock the copy.)

> *Please also note that due to the length of some of the Basic statements, the listings/extracts in this book sometimes have additional line-breaks, spaces, hyphens etc. in order to enhance their readability in print. You should refer to the supplied listings on the floppy disc (or CD-ROM) for complete accuracy.*

Programming style

It needs to be stressed that the programming style and routines used in this book are not being advanced by the author as paragons of virtue, nor 'the only way it should be done'. One of the joys of BBC Basic and the RISCOS is their flexibility: there is always more than one way to achieve an end. Sometimes this book has definitely given priority to making the listings easier to follow rather than locating a snippet of code in a more logical place.

Dr Wimp version

The book comes with a copy of the latest version of the complete Dr Wimp package - **which may be a later version than the one used to prepare the book (which was Version 3.80)**.

To help avoid any confusion whilst you are following the detail of the book, a complete copy of Version 3.80 of the Dr Wimp package is also included for reference. **Don't mix up the two versions!**

(Updated versions with added facilities are likely to appear regularly. Contact addresses for Dr Wimp updates etc. are given at the front of this book.)

The PRM

The book makes frequent references to the PRM - the RISCOS "Programmers Reference Manual". It is not needed for the purposes of this book but, nonetheless, the PRM is the only authoritative source of the details of RISCOS, particularly the details of SYS calls.

Conventions used in this book.

`Program listings/extracts are in this typeface`. The same typeface is used within the normal text when reference is made to program items e.g. " ..the function `FNwimp_quit` is used ... "

<angle brackets> are used (often also in the above typeface) in two ways: firstly to indicate in example listings/ extracts that the programmer needs to substitute something appropriate of his/her own choice at that place - and, secondly, to represent in the text a specific keyboard key press or mouse button action e.g. `<select>` to mean a press of the mouse 'Select' button.

The successive versions of the tutorial application - developed in Chapters 4-16 - are referenced to the chapter in which they first appear, using letters to distinguish between more than one in the same chapter e.g. `!Fuel16a` and `!Fuel16b` are, respectively, the first and second versions completed in Chapter 16. *The disc (or CD-ROM) associated with this book contains complete applications - with commented program listings - for each successive version of the tutorial application.*

Bold text is used **for emphasis**.

Italic typeface is used mainly for comments outside the main flow of the text.

Finally, it is necessary to distinguish between references to the Dr Wimp package as a whole and the specific `DrWimp` library within it. This is done exactly as in the previous sentence.

1. The nature of Wimp programs

This chapter gives a brief overview of how the Wimp operates and the consequential program structure. Many of the topics are dealt with in greater detail later.

Wimp and non-Wimp applications

A non-Wimp program - however small - monopolises the computer whilst it is running. Even when we do not specify a screen mode change and the program appears to run within a Task Window on the desktop screen, we cannot ‘get out of that window until the program has ended and we are invited to `"Press SPACE or click mouse to continue"`.

In contrast, when a Wimp program is run it usually adds something to what is already on the desktop screen. It doesn’t prevent user interaction with other applications already present and - memory space permitting - it doesn’t stop even more applications from being added. This is the obvious first difference between non-Wimp and Wimp programs.

The second main difference is the way the user makes an input. A Wimp program typically puts pictorial symbols (icons) on the desktop screen and most input is carried out by placing the screen pointer on one of the icons and clicking a mouse button. In non-Wimp programs, the keyboard is used much more.

Further, when a non-Wimp program needs user input, the computer sits there idly waiting until the input is made - and nothing else can happen until then. In contrast, Wimp programs can allow other tasks to proceed whilst waiting for user input.

A third difference is more pragmatic. Wimp programs tend to need a lot more program lines to be written before a program can meaningfully be run and cause something to appear on the screen. In fact, this initial hurdle has often been sufficient to put off newcomers from trying further. *(This is an area where Dr Wimp helps tremendously.)*

The Window Manager ("the Wimp")

Because a non-Wimp program monopolises the computer, the program proceeds line by line in exactly the way the programmer has structured it. There may be loops, branches and conditional statements etc. but they were all put there by the programmer alone and you can usually follow the flow from start to finish in the listing without undue difficulty. Essentially, the non-Wimp programmer never 'releases the reins'.

The fundamental change when we come to Wimp programs is that - in order to permit more than one program to be active on the desktop screen simultaneously - the programmer has to release the reins for some of the time or, more accurately, release some of the reins all the time.

And finally, the Window Manager automatically carries out a very large number of tasks which the programmer would have to do for himself/herself in a non-Wimp program.

Asking 'the question'

Roughly what happens for Wimp programs is that a manager program (called the Window Manager, or just "the Wimp") within the RISC operating system of the computer, is used to handle all the input and output to/from Wimp application programs. This does not mean just the input/output to devices such as a printer, but literally **all** the input/output to/from the application and the user e.g. mouse clicks etc.

Any Wimp application program therefore has to be structured to permit this - and this is the entry fee demanded by the Window Manager.

In colloquial terms, the application program needs to be structured to ask the Window Manager, **repeatedly**, “Anything happened lately, concerning my program?”

For much of the time the answer will be “No”. But if, for instance, the user makes a mouse click over one of the program’s icons on the screen, the Window Manager records that click and next time the question comes the Window Manager says “Yes, there was a mouse-click over an icon.” - and gives the details of which mouse button and which icon etc.

The application program then takes over control temporarily in order to carry out whatever consequential action it has been programmed to do i.e. appropriate to that particular mouse click. It then hands back control to the Wimp and reverts to repeating its earlier question, until some further positive response occurs.

What the application program doesn’t realise is that the Window Manager is being asked the same question repeatedly by every other active Wimp application program - and they all share the available time, usually without any apparent difficulty from the user’s viewpoint. Thus more than one application can run on the desktop at the same time.

The mental process

Seen from the viewpoint of a single application, the action can therefore be considered to proceed in a series of short responses (by the application) made after positive answers (from the Wimp) to ‘the question’ (asked repeatedly by the application).

Mentally, the programmer has to adjust to this and, by and large, it actually makes things somewhat easier. The main program structure is usually set by the repeated need to ask ‘the question’ and the programmer therefore can focus more on the user interfaces (the window design primarily) and the individual responses required to the range of possible user actions.

For example, a menu selection may be required to open a window into which some keyboard input is to be made into an icon. In Wimp programming terms this would typically break down into:

```
Construct and display the menu (as a response to
         pressing the <menu> button)
Select a menu item and open the window (as a
         response to clicking <select> over the
         menu item)
Display the text in the icon (as a repeated
         response to pressing a keyboard
         character)
```

Thus, this perfectly ordinary small program sequence actually involves three separate journeys round the question-and-answer routine between the program and the Wimp. (In fact, it is likely to involve more than three journeys, because the third action will be repeated for each press of the keyboard.)

How to join in

In order to join in with the Wimp, an application program needs to:

- log on with the Window Manager.
- keep asking ‘the question’.
- respond without delay to a positive answer.
- log off when the program exits.

Thus, the programmer still retains full control over what the program does - but not over precisely when it does it. Having said that, we are dealing with timings in milliseconds and the time-share aspects are usually invisible to the user.

As we shall also see, the Window Manager more than compensates for imposing itself in this way by relieving the programmer of a host of boring tasks.

For example, when the programmer wants a window shown on the screen he/she simply tells the Window manager to do it using a `SYS` call, passing any essential details in a parameter block. *(Appendix 1 reviews the structure and usage of SYS calls, in detail).* The Window Manager then does the difficult bits - and you will be surprised how much is done and how relatively easy it is for the programmer.

The Wimp poll

The particular control process used by the Window Manager to control all the active applications is a polling mechanism, called the Wimp Poll. The heart of all Wimp programs is usually just a fairly simple `PROC`, normally placed inside a `WHILE ... ENDWHILE` loop, which repeatedly ‘asks the question’ and then branches the program along an appropriate path when a positive response is received.

Thus, the outline structure of a typical Wimp program is:

```
PROCinit :REM** Sets up parameter blocks (such as
         DataBlock%, see below) and logs on to
         Window Manager **
WHILE (Quit% = FALSE)
      PROCpoll :REM** Calls PROCpoll repeatedly
            **
ENDWHILE
PROCexit: REM** Logs off from Window Manager **
END
DEF PROCpoll
SYS "WimpPoll" , 0 , DataBlock% TO Reason%
           :REM** 'Asks the question' and
           designates variable and data block for
           answer **
CASE Reason% OF : REM** Deals with a range of
           possible different answers **
       WHEN 0 : PROCaction0
       WHEN 1 : PROCaction1
       .
       .
       etc.
ENDCASE
ENDPROC
```

What is often initially surprising to the learner is that the core of every Wimp program tends to look the same. The differences come in the `PROCactionX` area.

The heart of the program is the line:

```
     SYS "Wimp_Poll" , 0 , DataBlock% TO Reason%
```

which 'asks the question' and puts the response into the variable `Reason%` and often also puts, into `DataBlock%`, detailed data associated with the response.

For instance, if the Wimp is reporting that a mouse click has occurred, it would put the 'mouse click has occurred' code into `Reason%` and put into `DataBlock%` the details of which mouse button was used, whether it was clicked once or double-clicked or dragged, which icon the pointer was over, etc. The programmer can then use these data to deal with the application's response to the mouse click.

Reason codes

The name of the variable used to receive the answers has deliberately been called `Reason%` because the Wimp Poll gives its answers by using "Reason Codes", which currently can take any of the values 0-19. The table in Appendix 2 gives the full description of all these codes and we will be returning to this subject in more detail later. However, for now, a brief glance shows that the reason codes cover all the usual user-input actions.

For instance, Reason Code 6 means that a mouse click has occurred - and therefore, in the above example, `Reason%` would be given the value 6.

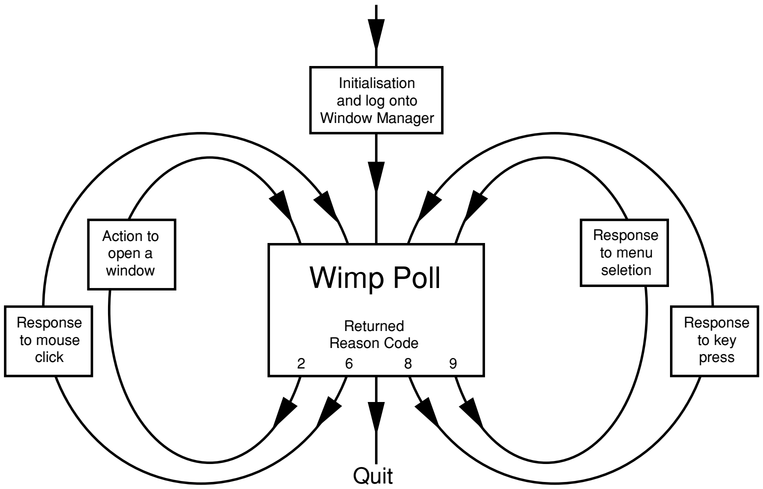
Similarly, Reason Code 9 means that an item from a Wimp menu has been chosen. Thus, the Wimp would put the value 9 into `Reason%` and, this time, `DataBlock%` would be used to hold data on which item, on which menu, etc.

So, the Wimp programmer's main task is to develop the responses to the various reason codes - potentially all 20 of them. Further, as most applications will involve more than one window/icon/menu, each reason code response will normally need to cater for many different window/icon/menu combinations. (So, `CASE ... ENDCASE` constructions are heavily used.)

Reason Code 0 needs a special mention because it is the one returned by the Wimp if the answer to ‘the question’ is “No”. The problem is that this is very much the most frequent response - so much so that it can sometimes slow an application down. For this reason it is often ‘masked out’ by the programmer so that it is not returned. There is more on ‘masking’ later.

Flow diagram

If we link the repeated action of ‘asking the question’ with the use of reason codes, it will be no surprise that a flow diagram of a typical Wimp program tends to comprise multiple main loops, all entering and leaving the Wimp Poll. The figure below represents this. Each loop (only four shown) represents a response to a different reason code - noting that, as was just said, each such response is likely to comprise many different sub-responses.



Schematic Wimp program structure

Parameter blocks

Wimp programming makes considerable use of parameter blocks and a few examples will be worthwhile.

In order to open a window, an application must make a sys call which tells the Wimp all the necessary information about the particular window e.g. its size, screen position, scroll offsets etc.

The programmer's procedure is, in effect, to place all this data in a parameter block and then pass the address of the block with the SYS call - as one of the `SYS` call's own parameters. A typical call might be:

```
SYS "Wimp_OpenWindow",,MainWindow%
```

and the required details will have been previously loaded into `MainWindow%` by the programmer. The window will then be duly opened by the Wimp.

Sometimes the Window Manager needs to prod the program to take some action, as a result of a user action not apparently directly associated with the application. For instance, if you drag an open window of your program to a different part of the screen the Window Manager will start issuing Reason Code 2 (request to open a window) because moving a window is essentially a repeated deletion and reopen exercise. In this case, the Wimp will helpfully and automatically place all the necessary data into the parameter block for you, but you will need to ensure that your program issues the `SYS "Wimp_OpenWindow"` call.

So, nearly every Wimp program needs, as a bare minimum, a routine to respond to Reason Code 2, otherwise the window simply will not drag.

Similarly, if you click on the window 'close' icon the Window Manager will issue Reason Code 3 and place the necessary data into the parameter block - but you have to ensure that `SYS "Wimp_CloseWindow"` is called in response to Reason Code 3.

In practice, as we will show, loading parameter blocks does not need to be a tedious task, because there are various tools to 'automate' it. And this is one of many areas where using Dr Wimp will make it really simple.

‘Shell packages’ and similar programming tools

Many Wimp programming actions need to use a relatively small number of SYS/PROC/FN calls over and over again - albeit with minor changes to the parameters being passed to them. Similarly, although heavy use is made of parameter blocks, it is usually by using relatively few blocks over and over again, for many different purposes.

Also, we have already seen that the basic structure of many Wimp programs is identical and centred on the Wimp Poll, as indicated in the earlier flow diagram.

Because of these factors, it becomes perfectly feasible to produce two things:

- a skeleton Wimp application which will serve as the backbone of many applications, and
- a library of `PROC/FN`s which will serve many programs, particularly to help with the constant loading of the parameter blocks followed by the right `SYS` call.

Thus, packages are available to do just this - sometimes called ‘shell’ packages - and maybe with other useful ‘tools’ included.

Dr Wimp can perhaps be best described as a ‘second-generation shell package’ because it extends the idea even further - as we will see.

None of these packages remove the need for the programmer to devise particular routines specific to each program nor the need to understand the above-described nature of Wimp programming. However, they can certainly relieve him/her from much of the repeated drudgery of the process and they can eliminate many of the keyboard inputting errors.

Naturally, you do not get something for nothing - the resulting programs from these tools are likely to need more memory than if you tackled each one from scratch and they may not run quite so fast. Fortunately, for many applications, these are not usually significant penalties

2. Why use Dr Wimp?

To convert the principles of the previous chapter into practice - and at the same time demonstrate the advantages of Dr Wimp - this chapter shows a small Wimp application produced in three different ways, but achieving exactly the same end results. The three ways are:

a) ‘from scratch’ i.e. using the necessary SYS calls directly and without using any libraries;

b) essentially the same as in a), but using window definition templates to define the windows and their icons;

c) using Dr Wimp.

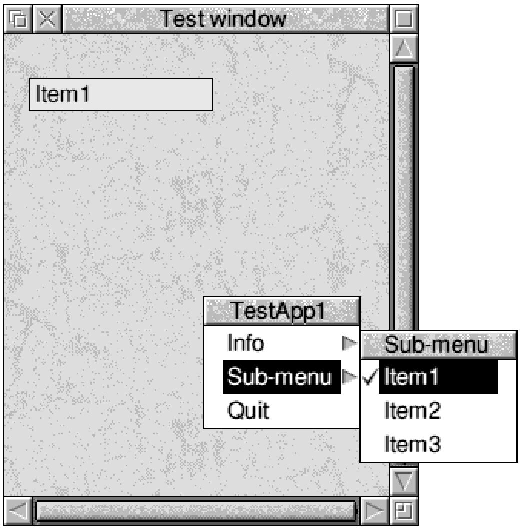
The three applications are on the book disc and are called `!TestApp1`, `!TestApp2` and `!TestApp3` respectively. The rest of this chapter examines them in more detail, enabling comparisons to be made and also providing an introduction to typical Wimp programming methods in Basic.

At this stage, the examination does not look in detail at the ‘application resources’ i.e. the **`!Boot`**, **`!Run`**, **`!Sprites`** etc. files within the application directory. These are covered in Appendix 3,

The demonstration applications

The three demonstration applications produce identical results.

Each ‘loads onto the iconbar’ in the usual Wimp way and pressing `<select>` over the iconbar icon then produces a small main window containing just one icon, carrying text, as shown in the following screenshot:



Pressing **`<menu>`** over the iconbar icon produces a menu with three items in it: the first and third menu items are the traditional ‘Info’ and ‘Quit’ items, respectively, and the second item is ‘Sub-menu’.

The ‘Info’ item has the usual type of **`Info`** box associated with it and the ‘Sub-menu’ item leads to a sub-menu containing three items “Iteml”, “Item2” and “Item3”. At the start, ‘Iteml’ of the sub-menu is ticked - and the icon in the main window shows the text “Iteml” accordingly.

Selecting any item from the sub-menu list causes that item to attract the tick and the text in the main window icon changes to reflect the selected sub-menu item i.e. “Iteml” or “Item2” or “Item3”.

As usual, the application is ended by selecting ‘Quit’ from the iconbar menu.

‘!TestApp1’

As `!TestApp1` involves using fundamental Wimp programming methods ‘from scratch’ it is appropriate to look at it in some detail. Thereafter, the comparisons with `!TestApp2` and `!TestApp3` can be handled more quickly.

Here is the complete !RunImage listing of `!TestApp1`:

```
10 REM** Wimp test "!TestApp1" **
20 REM** Example program defining window etc. from
            scratch. **
30
40 PROCinit
50
60 SYS "Wimp_Initialise",310,&4B534154,App$,
            WimpInitWord%
70
80 ON ERROR PROCerror:END
90
100 REM———————————————
110 REM** Define main window. **
120 TestWindow%=FNcreateWindow(400,400,500,600,
            0,0,&FF000012,"Test window")
130
140 REM———————————————
150 REM** Define icon within main window. **
160 $text%="Item1"
170 $valid%=""
180 IconTextLength%=17
190 Icon1%=FNcreateIcon(TestWindow%,32,-100, 240,
            44,&C7000135,"",text%,valid%,
            IconTextLength%)
200
210 REM———————————————
220 REM** Define info window. **
230 Info%=FNcreateWindow{0,0,456,204,
            0,0,&84000012,"Info")
240
250 REM———————————————
260 REM** Define icons within info window. **
270 RESTORE 3010
280 FOR N%=1 TO 4:READ Text$
290     Dummy%=FNcreateIcon(Info%,0,-N%*48-4,
            128,44,&17000211,Text*,0,0,0)
```

```
300 NEXT
310
320 FOR N%=1 TO 4:READ Text$
330 Dummy%=FNcreateIcon(Info%,128,-N%*48-4,
             320,44,&1700003D,Text$,0,0,0)
340 NEXT
350
360 REM———————————————
370 REM** Define iconbar icon - which also
             displays it. **
380 IconBar%=FNcreateIcon(-1,0,0,68,68,&3002,
             "!testappl",0,0,0)
390
400 REM———————————————
410
420 REM** Define iconbar menu and sub-menu. **
430 RESTORE 3030
440 PROCsetUpMenu(MenuBlock%)
450
460 RESTORE 3040
470 PROCsetUpMenu(SubMenuBlock%)
480
490 REM———————————————
500
510 WHILE NOT Quit%
520     PROCpoll
530 ENDWHILE
540
550 REM———————————————
560
570 SYS "Wimp_CloseDown"
580
590 END
600
610 REM*****************************************
620 REM*****************************************
630
640 DEF PROCinit
650
660 DIM WimpInitWord% 4
670 !WimpInitWord%=0
680
690 DIM DataBlock% 255,MenuBlock% 127,
             SubMenuBlock% 127
700
710 DIM text% 20
```

```
720
730 DIM valid% 20
740
750 Quit%=FALSE
760 App$="TestAppl"
770 SelectedSubItem%=0
780
790 ENDPROC
800
810 REM****************************************
820 REM****************************************
830
840 DEF PROCpoll
850 SYS "Wimp_Poll",&31,DataBlock% TO Reason%
860 CASE Reason% OF
870     WHEN 2:SYS Wimp_OpenWindow",,DataBlock%
880     WHEN 3:SYS "WimpCloseWindow",,DataBlock%
890     WHEN 6:PROCclick(DataBlock%!12)
900     WHEN 9:PROCmenuSelect
910 ENDCASE
920 ENDPROC
930
940 REM****************************************
950 REM****************************************
960
970 DEF PROCreport(Err$,Flag%)
980 Title$=App$
990 IF Flag% AND 16 THEN Title$="Message from "+
             Title$
1000 !DataBlock%=255
1010 $(DataBlock%+4)=Err$+CHR$0
1020 SYS "Wimp_ReportError",DataBlock%,
             Flag%,Title$ TO ,ErrorClick%
1030 ENDPROC
1040
1050 REM***************************************
1060
1070 DEF PROCerror
1080 PROCreport(REPORT$+" at Line "+STR$ (ERL),1)
1090 SYS "Wimp_CloseDown"
1100 ENDPROC
1110
1120 REM***************************************
1130 REM***************************************
1140
```

```
1150 DEF FNcreateWindow(x%,y%,w%,h%,extx%,exty%,
             Flags%,Title$)
1160
1170 REM visible work area
1180 !DataBlock%=x%
1190 DataBlock%!4=y%
1200 DataBlock%!8=x%+w%
1210 DataBlock%!12=y%+h%
1220
1230 REM scroll offsets
1240 DataBlock%!16=0
1250 DataBlock%!20=0
1260
1270 REM handle behind and window flags
1280 DataBlock%!24=-1
1290 DataBlock%!28=Flags%
1300
1310 REM window colours
1320 DataBlock%?32=7
1330 DataBlock%?33=2
1340 DataBlock%?34=7
1350 DataBlock%?35=1
1360 DataBlock%?36=3
1370 DataBlock%?37=1
1380 DataBlock%?38=12
1390
1400 REM work area extent
1410 DataBlock%!40=0
1420 DataBlock%!44=-h%-exty%
1430 DataBlock%!48=w%+extx%
1440 DataBlock%!52=0
1450
1460 REM title bar and work area flags
1470 DataBlock%!56=&19
1480 DataBlock%!60=3<<12
1490
1500 REM sprite area pointer and minimum size
1510 DataBlock%!64=1
1520 DataBlock%!68=0
1530
1540 REM window title
1550 $(DataBlock%+72)=Title$
1560
1570 REM number of icons
1580 DataBlock%!84=0
1590
```

```
1600 SYS "Wimp_CreateWindow",,DataBlock% TO
             WindowHandle%
1610
1620 =WindowHandle%
1630
1640 REM**************************************
1650
1660 DEF FNcreatelcon(Window%,x%,y%,w%,h%,
             Flag%,Text$,Ptr1%,Ptr2%,Ptr3%)
1670
1680 !DataBlock%=Window%
1690 DataBlock%!4=x%
1700 DataBlock%!8=y%
1710 DataBlock%!12=x%+w%
1720 DataBlock%!16=y%+h%
1730 DataBlock%!20=Flag%
1740 IF Ptr1%=0 THEN
1750    $(DataBlock%+24)=Text$
1760 ELSE
1770    DataBlock%!24=Ptr1%
1780    DataBlock%128=Ptr2%
1790    DataBlock%!32=Ptr3%
1800 ENDIF
1810

1820 SYS "Wimp_CreateIcon",,DataBlock% TO
             IconHandle%
1830
1840 =IconHandle%
1850
1860 REM*************************************
1870
1880 DEF PROCsetUpMenu(Menu%)
1890 REM** Defines a menu by loading parameters
             into a data block. **
1900
1910 READ Title$,Items%
1920 $Menu%=Title$
1930 MenuWidth%=16*LEN(Title$)
1940
1950 Menu%!12=&70207 :REM** Colours. **
1960 Menu%!20=44 :REM** Height of menu items. **
1970 Menu%!24=0 :REM**Vertical gap between
             items.**
1980
```

```
1990 REM** Sub-block of 24bytes for each menu
             item, starting at Byte 28. **
2000 Ptr%=Menu%+28
2010 FOR N%=1 TO Items%
2020    READ MenuFlag%,SubPtr%,MenuItem$
2030    !Ptr%=MenuFlag%
2040    Ptr%!4=SubPtr%
2050    Ptr%!8=&7000021
2060    $(Ptr%+12)=MenuItem$
2070    W%=16*(LEN(MenuItem$)+1)
2080    IF W%>MenuWidth% THEN MenuWidth%=W%
2090    Ptr%+=24
2100 NEXT
2110
2120 Menu%!16=MenuWidth%
2130 ENDPROC
2140
2150 REM*****************************************
2160
2170 DEF PROCclick(Window%)
2180 REM** Responses to mouse clicks over windows/
             icons. **
2190
2200 CASE Window% OF
2210
2220    WHEN -2
2230    PROCiconBar(DataBlock%!8)
2240
2250 ENDCASE
2260 ENDPROC
2270
2280 REM*****************************************
2290
2300 DEF PROCiconBar(Button%)
2310
2320 CASE Button% OF
2330
2340    WHEN 1,4
2350    !DataBlock%=TestWindow%
2360    SYS "Wimp_GetWindowState",,DataBlock%
2370    DataBlock%!28=-1
2380    SYS "Wimp_OpenWindow",,DataBlock%
2390
2400    WHEN 2
2410    PROCshowMenu(MenuBlock%,!DataBlock%-64,
             228)
```

```
2420
2430 ENDCASE
2440 ENDPROC
2450
2460 REM****************************************
2470
2480 DEF PROCshowMenu(Menu%,x%,y%)
2490 REM** Displays menu whose definition is
             already held in a data block. **
2500
2510 FOR N%=0 TO 2
2520    Tick%=SubMenuBlock%+28+N%*24
2530    IF N%=SelectedSubItem% THEN
2540          !Tick%=!Tick% OR 1
2550    ELSE
2560          !Tick%=!Tick% AND 254
2570    ENDIF
2580 NEXT
2590
2600 SYS "Wimp_CreateMenu",,Menu%,x%,y%
2610
2620 ENDPROC
2630
2640 REM****************************************
2650
2660 DEF PROCmenuSelect
2670 REM** Responses to menu/sub-menu selection.
             **
2680
2690 MainSelectItem%=!DataBlock%
2700 SelectedSubItem%=DataBlock%!4
2710
2720 SYS "Wimp_GetPointerInfo",,DataBlock%
2730 Button%=DataBlock%!8
2740
2750 CASE MainSelectItem% OF
2760
2770    WHEN 1
2780    Ptr%=SubMenuBlock%+28
2790    ItemStart%=Ptr%+24*(SelectedSubItem%)+12
2800    Item?=$(ItemStart%)
2810    $text%=Item$
2820
2830    !DataBlock%=TestWindow%
2840    SYS "Wimp_GetWindowState",,DataBlock%
2850    DataBlock%!28=-1
```

```
2860    SYS "Wimp_CloseWindow",,DataBlock%
2870    SYS "Wimp_OpenWindow",,DataBlock%
2880
2890    WHEN 2
2900    Quit%=TRUE
2910
2920 ENDCASE
2930
2940 IF Button%=1 THEN
        PROCshowMenu(MenuBlock%,0,0) :REM Keeps
             menu open if <adjust> used.**
2950
2960 ENDPROC
2970
2980 REM****************************************
2990 REM****************************************
3000
3010 DATA Name,Purpose,Author,Version
3020 DATA TestAppl,Wimp demo 1,Ray Favre,1.0
3030 DATA TestAppl,3,0,Info%,Info,0,
             SubMenuBlock%,Sub-menu,&80,-1,Quit
3040 DATA Sub-menu,3,0,-1,Item1,0,-1,Item2,
             &80,-1,Item3
```

Initial actions

(As was said earlier - a lot of lines for a small task!)

`PROCinit` sets up various memory blocks and global variables (particularly **`App$`** and **`Quit%`**) needed by the program.

In most Wimp programs, one memory block in particular, **`DataBlock%`** here, is generally used time and again as a temporary block for passing parameters to/from **`SYS`** calls - and it is usual to create this as one ‘page’ in size i.e. 256 bytes. (Some **`SYS`** calls do not need all this space, but some do - and occasionally more is needed.)

In this example, unique separate blocks are also set up for each menu and sub-menu used. This is a simple approach which is sensible when only a few menus are involved. However, to save memory space, some programs with many menus use/re-use fewer menu blocks - which is perfectly OK but requires stricter management, because a menu block needs to be unique to a menu whilst it is visible on the screen.

Logging on to the Wimp

The first significant action in the listing is to log on with the Window Manager ("the Wimp") and Line 60 does this. Note that this is not straightforward; not only do you have to give the application name, but you need to specify the lowest OS version the application is designed for. There is also a filter flag to ensure that Wimp messages are passed to the application as needed and a special task flag. The PRM contains the details.

It is worth noting here that many (but not all) `SYS` calls are meaningless outside the Wimp environment, so it necessary for the logging on to take place in the program sequence before such calls are used.

Error handling

Immediately following the logging on is the main error trap - at Line 80. The particular `SYS` call it uses (via `PROCerror` and `PROCreport`, see below) is `SYS "Wimp_ReportError"` and this is definitely one which cannot be used outside the Wimp.

Error handling in Wimp programs is very important - not the least because, in a multi-tasking environment, an error in one application could easily cause other applications to 'crash'. So, the general aim is to trap all errors occurring within an application and, at worst, close down just that one application. Hopefully, in many cases the error does not need to force this extreme and the error messages may be able to guide the user to correct the problem and try again.

The `SYS` call provided for error handling within the Wimp is pretty comprehensive, in that its parameters allow a choice of buttons to appear in the box for user action e.g. "OK" and "Cancel" and also allow the error box title to be changed to let the user know the error comes from the application and not elsewhere. The application icon can also be shown in the error box in later RISCOS versions.

Thus, this `SYS` call can be used to report both real errors and also helpful warnings to the user e.g. that a keyboard input needs to be a number rather than a letter; or unsaved data exists, etc.

The immediate practical consequences of this are that it is quite normal to use a pair of error functions, such as those at Lines 970 and 1070:

- firstly, at Line 970, a general `PROCreport` is constructed to call `SYS "Wimp_ReportError"` in such a way that its parameters determine the message to be carried in the error box and which types of buttons are presented for the user's action i.e. 'OK' or 'Cancel'.

  `PROCreport` can therefore be used both for helpful warnings to the user and for real errors. For helpful warnings, it can be called like any other `PROC` from anywhere in the program, as many times as you wish.

- then, `PROCerror` (at Line 1070) is called only on real errors - by the `ON ERROR` statement at Line 80 here. This calls `PROCreport` in a specific way - using `REPORT` and `ERL` to provide the error box text - and then exits from the Wimp before ending the program.

Defining windows, icons and menus

With the initiation tasks out of the way, the program can now start to look at its main visual features - its windows, icons and menus. These are all dealt with in a similar way, which is that they each have to be defined in detail and identified to the Wimp before they can be used. Once this is done they can be brought into play by the program as required.

There is a SYS call for defining each item:

```
SYS"Wimp_CreateWindow"(see Line 1600)
SYS"Wimp_CreateIcon"(see Line 1820)
SYS"Wimp_CreateMenu"(see Line 2600)
SYS"Wimp_CreateSubMenu"
```

and although they work in similar ways, there are some differences which are explained below.

Their common feature is that they all require the details of the window/icon/menu/sub-menu to be entered into a parameter block before making the `SYS` call (and the block is identified as input parameter `R1` of the call).

We will be looking at window, icon and menu design in greater detail in later chapters but at this point it is worthwhile to look at the general processes involved.

Windows

As was said above, windows need to be defined before they can be used. To define a window, a parameter block needs to be loaded with quite a lot of data in a very specific way. The usual programming practice is to construct at least one `DEF FN` to do this - so that it (they) can be used time and again for as many windows (and programs) as you like.

(The reason why more than one such `DEF FN` *might be needed is a pragmatic one. Many windows, for instance, are very similar apart from a few parameter block differences, so a simple* `DEF FN` *with few parameters can be of real use. However, sometimes this simple approach will not suffice and hence an additional* `DEF FN` *might be desirable - to offer wider choices.)*

In the above listing, such a `DEF FN` starts at Line 1150 and is called at Lines 120 and 230. It would, of course, be more normal to hold this type of `DEF FN` in a Library.

By following the `REM` comments in the `DEF FN`, you can see that a wide range of a window's characteristics are simply loaded into the 'temporary' block `DataBlock%` at the right places - as detailed in the PRM of course. For instance, the first 48 bytes of the parameter block are mainly taken up with the work area and window sizes and on-screen position, together with the colours to be used for the background, title bar etc.

The parameter block entries at bytes 28 and 60 (at Lines 1290 and 1460 in the listing above) concern the 'Window flags' and 'Work Area flags'.

'Flags' of this type are a very common feature of `SYS` call parameters. It is simply a shorthand description of the technique of using the individual bits of a data byte (or more) to decide whether a certain feature is present or not.

A 4-byte 'word' is often used for this purpose and then the particular combination of set/unset bits is, of course, directly represented by a specific integer number. Hence it is common to see flag entries simply shown as a number - as with `&FF000012` in Line 120 and `&84000012` in Line 230 - both passed to Line 1290. *(There are library routines or small utilities available to construct the required flag numbers.)*

The 'Window flags' use the four bytes (32 bits (0-31) starting at Byte 28 of the parameter block. The PRM details the meaning of each bit when set and it covers such things as: Is there a title bar/scroll bars/ back icon? Can the window be moved? and many more.

The 'Work Area flags' use a similar process to define the reaction of the window work area (i.e. its background) to mouse-clicks. In essence, precisely how mouse-clicks are reported by the Wimp to the task via the Wimp Poll. This is called the window's 'button type' and is an important feature of both window and icon design. Appendix 4 gives detailed information.

Having done all the work to load the parameter block, a very simple call to `SYS "Wimp_CreateWindow"` is made at Line 1600 - identifying the parameter block and telling the Wimp to put its output parameter `R0` into the integer variable `WindowHandle%` which is used here as the return value for the `FN`.

In common parlance, the `SYS` call has returned, in `R0`, a 'handle' for the window. This handle is, in fact, simply the start address of the Wimp-assigned memory block now containing the window definition.

Thus, the Wimp has read all the data in the 'temporary' block `DataBlock%` and created the window definition in a new memory block - unique for that window - and then tells us this location by returning the address in output parameter `R0`.

`DataBlock%` can therefore be used again to create other windows (or for other purposes). In our case, we create two windows and assign their handles to `TestWindow%` and `Info%` respectively. *(It is normal to give handles meaningful names. It helps considerably.)*

These handles are then used for all future references to the windows in the program. *(It is a very similar concept to the 'channel' returned in Basic when opening a file for access.)*

Handles are also commonly used for icons and menus as will be seen below.

Don't forget that the window creation process only defines a window - it doesn't display it.

Icons

Icons are defined in an entirely similar process, including the use of a 32-bit parameter block location for 'flags' - this time called the 'Icon flags' and which now include the 'button type' (see above).

The only additional aspect is that the programmer has the option of including icon definitions one by one at the end of the definition of their parent window, or to define them separately. It makes no difference - other than that, in the latter case, you have to tell the Wimp the window in which to create an icon, by passing the corresponding window handle.

In the above listing, they are created separately. The corresponding `DEF FN` starts at Line 1660 and this is called once at Line 190, several times in the loops between Lines 270-340 and once again at Line 380. Note that the first parameter passed is the window handle.

We will return to Lines 270-340 a little later but, for now, it is clear that this particular program only intends to make use of the returned icon handles in the first and last cases (`Icon1%` and `IconBar%`) - because the handles for the icons in the `Info%` window are all, in turn, assigned to `Dummy%` and hence all but the last in the loop will be lost.

With icons, the Wimp only makes their handles unique within any one window and, in fact, the icons are simply numbered from 0 upwards in each window and these numbers are the handles. *(Indeed, it is far more common to use the term 'icon numbers' rather than 'icon handles', but they are synonymous.)* So, an icon needs both its window handle and its icon handle/ number for unique identification.

With window handles, the programmer is usually content to assign a handle to a variable and use that variable name to refer to the window - probably not even knowing the actual value which has been assigned by the Wimp as the handle. However, with icons, the programmer is often very interested to know the actual icon number (handle) and also needs the means to change it (icon renumbering) to help ease the programming task. Appendix 7 looks at this in detail.

You may be a little surprised by the use of `FNcreateIcon` to define the application's iconbar icon. The iconbar is merely a special type of window (permanently open and whose handle, for icon creation purposes, is -1 for the right-hand side and -2 for the left-hand side). Because the iconbar window is always open, the icon creation of Line 380 will cause that icon to be displayed immediately i.e. the icon will 'install itself on the iconbar' in the usual program-loading way.

Finally, note that the same parameter block, `DataBlock%`, is used for all the window and icon creations in the above. *(If you include the icon definitions within the window definition you might well need a parameter block larger than 'one page'.)*

Menus

Both menus and sub-menus are defined with a very similar process initially i.e. their detailed description is loaded into a parameter block which is then identified in a SYS call. ‘Menu flags’ are also involved.

However, there are some important differences from windows and icons:

- creating a menu or sub-menu also displays it on the screen;
- the parameter block(s) used need to be available exclusively for each menu and any submenu(s) **whilst they are being displayed.**

These differences usually mean that separate parameter blocks are set up for the exclusive use of menus/sub-menus - with the number of blocks being at least equal to the maximum number of menu/submenus intended to be visible at the same time i.e. at least the number of the largest menu ‘tree’. This also means that the addresses of these blocks can act as the menu/sub-menu handles.

In the above listing, only one menu is used - from the iconbar - and only one sub-menu can appear (either from the ‘Info’ or ‘Sub-menu’ items on the main menu).

We can therefore afford the luxury of two exclusive parameter blocks - `MenuBlock%` and `SubMenuBlock%` dimensioned at Line 690 - which are loaded with the required data by a suitable `DEF PROC` at Line 1880 (called at Lines 430-470) and left loaded permanently. The data block names (which are integer variables containing the memory addresses, of course) then serve as the menu/sub-menu handles.

When the menus need to be displayed, `PROCshowMenu` (at Line 2480) is called and its first parameter is the required menu handle - followed simply by the required screen position.

In a program with many menus using the same blocks and/or with menus that may need last-minute (i.e. ‘dynamic’) changes before display, the parameter block loading would need to take place just before display each time - but the principles are the same.

Putting text in icons and menus

There are two ways of putting the text into icons and menus.

If the text is no more than 11 characters (plus a terminator character, added automatically) **and** it does not need to be changed during the program run, then this ‘fixed’ text must be included during the icon/menu creation stage.

However, if the text is longer than 11 characters **or** it needs to be changed during the program run, then ‘indirected’ icon/menu text is needed.

This second method requires the text to be put into a separate parameter block (often called a ‘buffer’ for this usage) and then the Wimp is simply told, in the icon/menu definition, the address of this buffer and the maximum number of characters allowed - which can be up to 256 including a terminator. Having done this, you can then alter the contents of the buffer at any time - up to the limit of the defined maximum size - and the icon text is changed correspondingly the next time the appropriate part of the window is displayed/redrawn.

The `!RunImage` listing of `!TestAppl` uses the second method for the single icon in `TestWindow%` (at Lines 150-190) - and uses the first method to put text into the `Info` window icons (at Lines 270- 340). The particular way of using the first method - via `DATA` and `READ` statements - is fairly easy to follow in the listing. This method is quite popular for the Info window, because the text will not change during the run of the program. Should, for instance, a version update occur, it will then be easy just to change the `DATA` lines.

For simplicity, the first method is used again for all the menu text. However, the detailed process for menus is somewhat more circuitous because each item on the menu needs to have its own menu flags and any sub-menus need to be specified.

Again, a very common way of handling this is with `DATA` statements detailing each menu item as a set of three values - two integers and a string each time. These are then incorporated into a parameter block prior to the `SYS` call to construct the menu.

The `DEF PROC` actioning this starts at Line 1880 (called at Line 440 and Line 470) and the `DATA` statements are at the end of the listing (Lines 3030 and 3040). From this you will see that, even in this simple case, you will need your wits about you to construct the parameter block - and, even with a decent library function to do that, you would need to understand fully what is needed for the `SYS` call.

Use of libraries

Clearly, with so many parameters to handle, the creation of windows, icons and menus form a major reason for producing libraries of `DEF PROC/FN`s for the purpose. You will run across many different ones, each with their own pros and cons.

Generally speaking, if you are going to program this way, it doesn't matter greatly which library you use as long as you are familiar with it. Each library usually comes as a consistent set of functions covering quite a wide range of needs - but it is very important that you don't mix and match libraries in the same program!

As we shall shortly see, Dr Wimp takes the idea of libraries at least one stage further, and better.

The Wimp poll

After all this setting up, the Wimp poll loop at Lines 510-530 looks like an anti-climax! But a peek at `PROCpoll` itself (starting at Line 840) shows that it is indeed the heart of the program.

The first of the parameters passed with the call to `SYS "Wimp_Poll"` itself is a ‘mask’ whose role is to filter out any Reason Codes (see previous chapter and Appendix 2) not required by the program.

Not all Reason Codes can be filtered, but several can and it is sensible to tell the Wimp not to send them to your application if you are not going to use them. In particular, the Null Code (Reason Code 0) is frequently a prime candidate for filtering *(and Dr Wimp automatically does this in certain circumstances - see Chapter 4).*

Just as with the ‘Flags’ described a little earlier, the individual bits of a 32-bit ‘word’ are set/unset to determine which particular Reason Codes are masked out or not. Thus the mask translates into an integer number.

In the listing, Codes 0, 4 and 5 have been chosen to be masked - and setting bits 0, 4 and 5 (and unsetting the rest) gives a mask value of `&31`.

As indicated, some Reason Codes cannot be masked and therefore some of the 32 bits must always be zero (see Appendix 2).

The second parameter passed to the `SYS` call simply tells the Wimp the address of the data block to be used for passing any detailed information.

Dealing with the Reason Codes

Lines 860-910 deal with specific Reason Code returns in a typical `CASE ... ENDCASE` construction. Only Codes 2, 3, 6 and 9 are dealt with in the listing. There is nothing wrong from a program viability viewpoint in simply ignoring other codes even if they are not masked out.

For each Reason Code there is some defined action. Thus, the program action is exactly as shown in the earlier flow diagram - `PROCpoll` is called repeatedly and, each time, the Wimp will send back a Reason Code if appropriate (and if it hasn’t been masked). The specific action (if any) for that Code only is then carried out by the program and `PROCpoll` exits - back into the loop, normally to be called again. Thus, conceptually, a Wimp program can be said to proceed in short bursts of activity.

Main program operations

In our listing, action can only start by clicking a mouse button over the iconbar icon. The Wimp will then duly return Reason Code 6 - “mouse click”. More than that, the Wimp will also load up the data block (identified to it in Line 850) with a lot of useful information e.g. the pointer x/y coords, which button was pressed, window handle and icon handle (over which the button was pressed).

The programmer then extracts any of this data from the parameter block and makes use of it as required. In our case, starting at Line 2170, whatever button is pressed, the reaction is only to call `PROCIconBar`, but passing as the parameter the contents of `DataBlock%!8` - which (the PRM tells you) is the information on which button was pressed. A value of 1 means `<adjust>`, 2 means `<menu>` and 4 means `<select>`.

Moving on to `PROCiconBar` at Line 2300, we see that for `<select>` or `<adjust>` we want to open the main window, whilst for `<menu>` we want to call `PROCshowMenu` (which displays the menu). Note that only one of these things will happen this time round the loop. After opening the window or the menu, the response to Reason Code 6 has finished and that particular pass of the `PROCpoll` loop ends.

If we clicked with either `<select>` or `<adjust>` the sequence at Lines 2350-2380 would be followed . Before the `SYS` call to open the window, the data block needs to be loaded. This is done by telling the Wimp the window handle to use and then calling `SYS "Wimp_GetWindowState"` (Lines 2350 and 2360). This loads the block with the ‘last known state’ of the window - which, in this case, is the state it was created in (but it might not always be that). To be absolutely sure, the block then needs to be modified (at Line 2370) to tell the Wimp to open the window on top of any others, before the actual `SYS` call to open it. When displayed, you will see that our previously created icon is duly in the window.

This may seem a tedious routine, but it is the only way to tell the Wimp precisely which window (of many it may be handling) is to be opened - and how.

Had we clicked with `<menu>`, the iconbar menu would now sit on the screen awaiting further action - hopefully a mouse click over one of its items/sub-items in order to make a menu selection. A mouse click over a displayed menu generates the Reason Code 9 - “menu selection” and this time Line 900 takes us to `PROCmenuSelect` at Line 2660.

Once again, the Wimp will have loaded the Wimp poll data block with useful information - in this case, which item(s) of the menu tree have been selected. In our listing - which, at most, can only have one menu item and one sub-menu item selected - these two items are first extracted at Lines 2690 and 2700 before further processing. *(Had we merely slid to the right on the 'Info' item, the* `Info` *box would have been displayed and we would not normally need to make a selection on this tree.)*

If we had slid to the right across the 'Sub-menu' item, the sub-menu would have been displayed and let's assume that we had clicked on 'Item 3' in this sub-menu. Lines 2690 and 2700 would therefore extract main menu item 1 (first in the list is 0) and sub-menu item 2. These values are then used in Lines 2780-2810 to extract the submenu item **text** from the sub-menu data block which was set up at Line 470. This text is then put into the indirected text buffer of the single icon in the main window - but this, in itself, does not cause the changed text to be displayed.

The sequence in Lines 2830-2870 then merely causes that window to be displayed/re-displayed to show the changed icon text. *(There are much better ways of re-displaying changed icon text - but a full window closing/opening is as good as any for a small window with only one icon.)*

Finally, had we selected the 'Quit' item from the menu i.e. item 2, the sole action is to change the `Quit%` flag to `TRUE` - which means that the `PROCpoll` loop at Line 510-530 will be bypassed and the application closed via Line 570.

(You may also wish to note that this application cannot be closed from the Task Display. The significance of this is commented on a little later.)

Review of process

The above description of `!TestAppl` has been given in some detail deliberately to demonstrate both the Wimp process and its typical programming needs. Bear in mind that the application is very simple and we have not explained every nook and cranny - and yet there is an awful lot of parameter block loading and extraction needed, both in the initial window/icon/menu definitions and in the run of the action.

You will also possibly have begun to notice in the listing that some of the parameter block loading needs are very similar to each other e.g. the similarity between the window and icon creation parameter blocks in some places. It goes much deeper than this: the patterns of data entry/ extraction are kept identical wherever possible - merely all being copied, perhaps, between different calls. This doesn't relieve the programmer from the need to keep track of what is going on but it makes the tedium of parameter blocks at least logical and the patterns capable of being assimilated as time goes by.

The sense of using libraries of proven functions also becomes obvious.

!TestApp2

Now let's compare the `!RunImage` listing of `!TestApp2` (on the supplied floppy disc or CD-ROM) with the above. The only difference in the approach is that this time the window and icon definitions have been prepared in advance outside the program and stored in a window template file - called "Templates" - held in the application directory.

Producing window definitions from a template editor is looked at later in the book - so accept for the moment that the same windows as used in `!TestApp1` are held in `Templates`.

A quick glance at the `!RunImage` listing of `!TestApp2` will reveal that the main area of change is in the sequence in Lines 110-200 and the consequential elimination of the long `DEF FN`s to create the windows and icons. Indeed, the `!RunImage` is now some 60 lines (20%) shorter.

The `SYS` call strings starting at Line 110 are almost self-explanatory. The template file is first opened and then, in turn, each window definition is loaded into the application and created as before. In effect, the loading from the template file automatically loads the 'temporary' parameter block `DataBlock%` with all the data before the window is created. As before, the output from the creation is the unique window handle returned in `R0`.

The process is much simpler but it is not entirely painless. Separate blocks are needed to hold any indirected window/icon data during the loading process - and these need to be large enough. Handling these is not without its pitfalls (particularly if outline fonts are to be used) and reference to the PRM is essential to steer the way through. In our case, the program is simple and exclusive extra data blocks have been set up in `PROCinit`.

There is also one change in the program action which is consequential on using window/icon templates and it is worth noting because it is a typical consequence.

It arises because, by preparing a window template, we no longer know the name of the buffer which holds the indirected text of the icon in the main window. Such a buffer necessarily still exists, but it is created automatically by the Wimp on loading the template.

So, how do we get at this buffer to change the indirected text? Have a look at `PROCmenuSelect` in the `!RunImage` of `!TestApp2`:

```
2040 DEF PROCmenuSelect
2050 REM** Responses to menu/sub-menu selection.
2060
2070 MainSelectItem%=!DataBlock%
2080 SelectedSubItem%=DataBlock%!4
2090
2100 SYS "Wimp_GetPointerInfo",,DataBlock%
2110 Button%=DataBlock%!8
2120
2130 CASE MainSelectItem% OF
2140
2150      WHEN 1
2160      Ptr%=SubMenuBlock%+28
2170      ItemStart%=Ptr%+24* (SelectedSubItem%)+12
2180      Item$=$(ItemStart%)
2190
2200
2210      DataBlock%=TestWindow%
2220      DataBlock%!4=0
2230      SYS "Wimp_GetIconState",,DataBlock%
2240      Pointer%=!(DataBlock%+8+20)
2250      $(Pointer%)=Item$
2260
2270      SYS "Wimp_GetWindowState",,DataBlock%
2280      DataBlock%!28=-1
2290      SYS "Wimp_CloseWindow",,DataBlock%
2300      SYS "Wimp_OpenWindow",,DataBlock%
2310
2320      WHEN 2
2330      Quit%=TRUE
2340
2350 ENDCASE
2360
2370 IF Button%=1 THEN
          PROCshowMenu(MenuBlock%,0,0)
2380
2390 ENDPROC
```

Line 2230 has the answer. We now have to locate that buffer by calling `SYS "Wimp_GetIconState"` which loads up the data block with full information about the icon - and the text buffer location is held at Byte 28 (so the PRM tells us).

You will also see, a few lines on, that there is a similar fact-finding `SYS` call for windows - and you will not be surprised to know that there is an equivalent for menus as well.

So, in summary, the window/icon definition process is easier using templates - and this is almost universally the practice. However, there are some consequential prices to pay (some of which do not appear in our simple example).

(This application cannot be closed from the Task Window either!)

!TestApp3

This is the demonstration application produced using Dr Wimp (Version 3.80) to achieve the same results.

Note first that the `!RunImage` is much shorter - only just over 200 lines. But now the `DrWimp` library is an essential partner - and this is very long. So, as you might expect, we are paying for ease of use by needing more program space. *(Using some of the supplied utilities - after completing a program - the required program space can usually be very much reduced, as Chapter 16 will show.)*

Looking at the `!RunImage` in more detail, the main program structure is contained within Lines 10-120 - a dozen lines only (and half of those are non-executive!).

The DrWimp library and 'wimp-functions'

In particular, Line 40 calls the `DrWimp` library, which normally resides (as here) in the same directory as the `!RunImage` - but it doesn't need to. This library mainly comprises a large number of `DEF PROC/FN`s whose names all start with `wimpxxxxxxxx` (all in lower case). In Dr Wimp parlance, these are called 'wimp-functions' and nearly all of them are available to be called from your `!RunImage` - just like any other library. *There are also some subsidiary 'internal'* `DEF PROC/FN`*s in this library i.e. routines supporting the wimp-functions but not intended to be called directly by the user/programmer.*

The 'user-functions'

Then, from Lines 170-1970 of the `!RunImage`, there are over thirty `DEF PROC/FN`s all starting with `user_xxxxxxxx` (again, all lower case) - and most of them empty. These are called the 'user-functions' and they work the other way round **i.e. the user-functions are called by one or another of the 'wimp-functions'** in the `DrWimp` library.

This means that it is essential that all the Dr Wimp user-functions are always present in your `!RunImage` even though any or all of them may remain empty. *(It is perhaps now also obvious that the list of user-functions in the* `!RunImage` *must be kept compatible with the particular version of the* `DrWimp` *library being used.)*

In fact, as will be explained later, **all** the programming undertaken by the programmer will normally only involve filling these user-functions - but not necessarily all of them.

The 'app-functions'

After the user-functions is a `DEF PROC` starting with `app_xxxxxxxx` (and not all in lower case). This is a function which has been created by the application programmer (as opposed to the Dr Wimp author) for this particular application - as a result of filling one of the user-functions. In anything other than a simple application there would be many 'app-functions' - to structure the program sensibly. These app-functions are likely to make heavy use of the many wimp-functions available.

However, the use of the prefix `app_` is purely a whim of this book's author, to help distinguish which things are from Dr Wimp and which are not. It is suggested that you adopt a similar policy when using Dr Wimp.

Don't forget: the order of the user- and app-functions in the `!RunImage` *listing is not important - but all user-functions must always be present.*

Brief program description

For this chapter, we need only give an overview of the Dr Wimp process - leaving more detail to later.

The early action takes place within `PROCuser_initialise` - which, remember, is called automatically from somewhere within the `DrWimp` library. (In this case, within `DEF FNwimp_initialise` - which is itself called at Line 80 of the `!RunImage`.)

From the meaningful `PROC`/`FN` names used, you will be able to follow the action easily. For instance, `FNwimp_loadwindow` (at Lines 240, 250) looks as if it returns a window handle and therefore you will probably guess that it combines the template loading and window creation steps into one - and you would be right! Its parameters are simply the template file path and the particular window name on that file. It really is as simple and painless as that.

Again, `FNwimp_iconbar` (at Line 320) puts a named icon onto the icon bar - no fuss!

Even better, `FNwimp_createmenu` (at Lines 360, 370) does the same for menus, and if you want to attach a submenu - or, indeed, a window - to a menu item then `PROCwimp_attachsubmenu` (at Lines 410, 420) does that. *But note that these menu actions only define the menus/sub-menus and their structure. With Dr Wimp, the display of the menus is dealt with separately - by* `FNuser_menu`*, see below.*

These few lines have already set everything up - without a mention of parameter blocks!

Returning to the user-functions, you will see that - in addition to `PROCuser_initialise` - three others contain additional program lines. They are:

```
DEF PROCuser_mouseclick
DEF FNuser_menu
DEF PROCuser_menuselection
```

and, as their names suggest, all of the actions for the user to operate the program are now contained in these.

`FNuser_menu` is called by somewhere within the `DrWimp` library every time the `<menu>` button is pressed - and note that the call will be passing to you, the programmer, active values in its parameters. In this case, it will be the window and icon handles over which the button press occurred.

If you, the programmer, leave `DEF PROCuser_menu` empty then nothing more will happen - it will return zero to whatever called it in the `DrWimp` library. However, if you put some action into the `DEF FN` and cause it to return a valid menu handle, then that menu will be duly displayed when you press the `<menu>` button. As you can see at Lines 750-780, the listing checks that it is the iconbar window and then returns the iconbar main menu handle. *(No need to check which icon in this case as it can only be one.)*

Very similar action occurs in `PROCuser_mouseclick` - which is called from the `DrWimp` library whenever the `<select>` or `<adjust>` button is clicked. As you can see, the listing checks that the window is the iconbar and, if so, it opens the main window with the wimp-function **`PROCwimp_openwindow`** passing it the required window handle. No bother about getting the window state or loading a parameter block!

The listing has a little more in **`PROCwimp_menuselection`**, but it is equally easy to follow. The call from the **`DrWimp`** library passes the handle of the actual menu from which the selection was made and the number of the item selected from it i.e. if the selection was from a sub-menu it is the sub-menu handle and item which will be passed.

The listing, at Lines 1010-1180, needs to cope with both menu and sub-menu selections. It does this by using a straightforward **`CASE ... WHEN ... ENDCASE`** construction which checks which menu and item is involved and takes action correspondingly.

For example, if ‘Quit’ was selected from the main menu (Item 3 - the top item is 1 in Dr Wimp) then **`PROCwimp_quit(0)`** is called - to quit the program.

However, if the selection had been from the sub-menu then the tick against the selected sub-menu item is first changed to reflect the selection and the text of that item is retrieved and put into the main window icon. All these actions are carried out with simple wimp-function calls.

In particular, look at **`PROCwimp_puticontext`**. Not only does this change the text in the icon but, if the window is open, it updates the display as well - again, a lot of hassle done automatically for us.

At Line 2020 is the one and only app-function **`DEF`** in this listing, which is called at Line 280 in **`DEF PROCuser_initialise`**. This app-function also uses **`PROCwimp_puticontext`**: this time in a loop to put the text into the **`Info`** box. Compare this with what we needed to do in **`!TestAppl`**.

(Finally, **`!TestApp3`** *does close from the Task Display properly! To put you out of your misery, this means that, unlike the first two examples, a Dr Wimp program automatically incorporates the Wimp Messaging system - see Appendix 10.)*

Review of the three methods

Even with this simple application, comparing the three methods of achieving the same end shows the main advantage of using the Dr Wimp package - its sheer ease of use compared with ‘do-it-yourself Wimp programming.

Gone is all the grind of dimensioning the various memory blocks, repeatedly loading them accurately and then making the right `SYS` call properly. All this detail is carried out automatically behind the scenes in the `DrWimp` library.

Similarly, all involvement in the Wimp Poll, Reason Codes and extracting and interpreting what is put in the parameter block by the Wimp is entirely invisible to the programmer. This is hidden behind the user-functions, which are much more programmer-friendly.

Another advantage is that the `!RunImage` contains only higher-level code - primarily comprising wimp-functions with their self-explanatory names and parameters. Understanding and keeping track of the program is therefore also much easier.

This demonstration has used only a handful of the many available wimp-functions in the `DrWimp` library - enough to do most things you are likely to need. *See Section 3 of the Dr Wimp manual for a complete list - or use* `!Fnc'n'Pre`. But if you do ever have a need for something which is not covered by the wimp-functions, then there is no difficulty in adding your own Basic routines - and the later tutorial program contains an example. In this respect, Dr Wimp is ‘non-limiting’.

Note also that in comparing `!TestApp1` and `!TestApp2` the big advantage of using a template file to define the windows/icons separately was somewhat offset by the added complication of handling the consequential indirected data. When we came to `!TestApp3` the advantages of templates was enjoyed without the disadvantages - with the difficult outline font handling also taken care of into the bargain.

For the newcomer to Wimp programming, perhaps the biggest advantage of using Dr Wimp is that something tangible can be reliably produced on the desktop screen within a few minutes of dabbling. The tutorial which comes with the package demonstrates this very effectively.

This is a great confidence-builder. The newcomer may not fully understand what is happening at that stage but it is much easier to find out when a successful step (however small) has been achieved.

By the same token, the more you use Dr Wimp the better will become your understanding of Wimp programming in general. With the coding of the `DrWimp` library fully accessible, you will be able to see how things are done.

Finally, whichever of the three methods is used, you do have to know either which `SYS` calls or which wimp-functions exist, what they do and how to use them. With `!TestApp1` and `!TestApp2` the only real source of the necessary information is the (somewhat daunting) PRM. In contrast, for `!TestApp3` the equivalent items are the wimp- and user-functions, of which there are relatively few and they all have meaningful names.

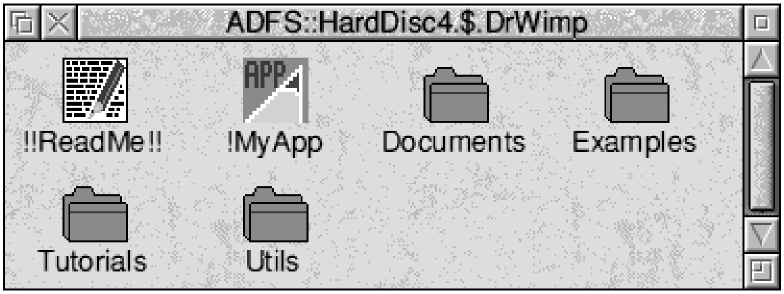
On this aspect, Dr Wimp wins hands down. You will be surprised how quickly you will remember all the common wimp-functions and if you do need to jog the memory, the package comes with an excellent on-screen index of all the wimp- and user-functions, with built-in browse and search facilities - via the utility `!Fnc'n'Pre`. Alternatively, if you prefer, Section 3 of the Dr Wimp manual contains the same information in list form.

"O.K. You've convinced me!" - I hear you say!"So tell me more - and how do I start to use Dr Wimp?" Read on…

3. What's in the Dr Wimp package?

The Dr Wimp package is a Freeware Public Domain (PD) package originally authored by Andrew Ayre and, from May 1999, supported by Ray Favre. A disc containing the latest version of the package comes with this book and can also be obtained by post, Email or the Internet. *(The contact addresses are given in the front of the book.)*

When you 'open' the disc root directory you should see something like this:



These folders etc. contain the following:

!!ReadMe!! Contains a summary of the package contents and the important distribution conditions of the author.

!MyApp The Dr Wimp standard 'blank' application - explained later in this book. Most importantly, this application contains the `DrWimp` library i.e. what the fuss is all about!

Documents Primarily contains two versions of a tutorial manual - one in plain text format and one in Impression Publisher format. Also contains `!Fnc'n'Pre` (the onscreen reference utility for the up-to-date list of user- and wimp-functions) and its StrongHelp equivalent `FuncProc`. In addition there are notes on the version history of the package, upgrading instructions for the various versions, a note on the importance of using certain utilities and, finally, the contact addresses for the package author together with certain accreditations.

Examples Contains almost 30 example applications prepared using Dr Wimp - all of them with their `!RunImage` well-commented to help you follow what has been done.

Tutorials Contains window template files and progressive listings for the tutorials in the on-disc manual (see “Documents” above).

Utils Contains the utility applications which are regarded as part of the total Dr Wimp package and each is described in more detail within this book. They are:

`!StrongBS`	(by Mohsen Alshayef)
`!CodeTemps`	
`!Crunch`	(by Bernard Jungen)
`!Fabricate`	
`!Linker`	
`!MakeApp 2`	(by Dick Alstein)
`!TemplEd`	(by Dick Alstein)
`!UnCrunch (or !CrunchFix)`	(by Jim Hawkins)

Please note that all of these utilities are PD of one form or another and each has its own distribution conditions - which are contained in files within each utility application.

The DrWimp library

The heart of the package is the `DrWimp` library - on the surface, a conventional Library in the Basic sense i.e. simply a listing of `DEF PROC/FN`s any of which may be called from another Basic program once an appropriate `LIBRARY` (or `INSTALL`) call has been set up.

However, the `DrWimp` library is far from conventional in other respects and it is essential to realise that **it does not 'stand alone'**. As will be explained, it can only be used in partnership with a program which itself contains certain specific, corresponding `DEF PROC/FN`s i.e. the user-functions. All will become clear in due course - but it is this feature which makes it so powerful and sets it apart from normal Wimp 'shell' programs.

This book is based on Version 3.80 of the Dr Wimp package, issued in June 1998. *(If a later version is available at the time of sale, a copy of this will be included.)*

4. Getting started with Dr Wimp

The Dr Wimp package comes with this book, so the first step is to copy (and de-archive) the disc contents across to your hard disc.

At the time of writing the current version of the package was Version 3.80.

(Note that it is the `DrWimp` *library version which defines the package version)*

Tutorial in Dr Wimp Manual

A very useful tutorial sequence is described in the disc-based user Manual in the package. Inevitably, this book will be overlap some of that tutorial activity, but to gain familiarity with the basic package some overlap will do more good than harm.

The main parts of Dr Wimp

As Dr Wimp is primarily an aid to help you make Wimp applications, this book is going to do just that. It will develop a complete application step by step over several chapters, as a vehicle to show how Dr Wimp is used in practice and to introduce its main facilities comprehensively.

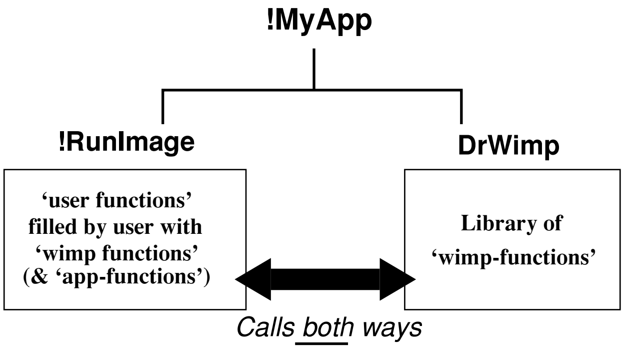
To get started, we need to get to grips with three parts of the package. They are:

The "DrWimp" library (a special library of Basic **`DEF PROC/FN`**s).

The skeleton Wimp application called "!MyApp". This contains a skeleton **`!RunImage`** Basic program plus the above library. (The **`!RunImage`** will be expanded by the programmer to form the main application program.)

The Window Template editor **`!TemplEd`** (supplied with the package, in the "Utils" directory).

As has been stressed already, it is vital to appreciate that the first two items are an inseparable pair. The library and the **`!RunImage`** program both contain **`DEF PROC/FN`**s and **each calls `PROC/FN`s held in the other**. The following diagram shows the relationship schematically:


!MyApp
!RunImage
DrWimp
'user functions' filled by user with 'wimp functions' (& 'app-functions')
Library of 'wimp-functions'
Calls both ways

So, the **`DrWimp`** library is not a conventional, free-standing library which can be used independently with any program - and therein lies the power and ease of use of the package.

*When the **`DrWimp`** library is upgraded, if you already have a completed application authored with Dr Wimp (and you wish to upgrade the* *application) it is quite likely that the* `!RunImage` *of the existing application will need to be upgraded correspondingly. For example, an upgrade might introduce a new user-function or change the parameters of an existing user- and/or wimp-function. (The* `Upgrading` *file in the* `Documents` *folder always tells you exactly how to upgrade an existing application. It is usually very simple.)*

You will therefore appreciate immediately that an application will inevitably come to grief if it was prepared with an earlier version of the Dr Wimp package and an attempt is made to run it with a later version of the `DrWimp` *library without upgrading its* `!RunImage` *in accordance with the Upgrading file.*

For this reason, all upgrades of `DrWimp` *are issued as a complete new Dr Wimp package - together with instructions about how to upgrade existing applications, if necessary.*

As we have already seen, it is best to use a template editor to design an application's windows/icons. Any RISCOS-compatible template editor will do, but as `!TemplEd` comes with the package and is very good, this book will be using it to prepare the project application.

Format of Dr Wimp's functions

As has been touched upon briefly in an earlier chapter, the user-functions and wimp-functions provided in the Dr Wimp package are all named meaningfully and consistently, using lower-case entirely.

User-functions defined in the skeleton `!RunImage` program (within the `!MyApp` application directory) are all in the form:

```
    PROCuser_dosomething e.g. PROCuser_menuselection
```

and wimp-functions in the `DrWimp` library are all of the form:

```
    PROCwimp_dosomething e.g. FNwimp_createmenu
```

Broadly speaking, the wimp-functions do all the tedious work and we call them from within the user-functions - to steer the application at a higher (and simpler) programming level.

Inside the `DrWimp` library you will find a listing of numerous wimp-function definitions. Section 3 of the Dr Wimp Manual (and the utilities !Fnc'n'Pre and FuncProc) describes all of the user- and wimp-functions available to the programmer for the Dr Wimp version for which they are relevant. (Not all wimp-functions are intended to be called by the programmer - a few are designed only to make the Dr Wimp author's task easier. These do not appear in the Manual etc.)

Inside the skeleton `!RunImage` file there is a very short main program listing (which we will look at soon) followed by all the user-function `DEF PROC/FN`s - all of which start off empty. (Appendix 8 specifically examines user-functions.)

Closer examination (if you are so inclined) will show that each user-function defined in `!RunImage` is called from within at least one of the wimp-function definitions in `DrWimp`. This provides the inseparable linking in the package and, remembering that all the user-functions start off as blank, it is the contents of the user-functions which are built up by the programmer to achieve the required end result.

A corollary to this is that although there is a need to become familiar with what most of the wimp-functions do and, indeed, to call them - it is very important that you **do not attempt to alter the `DrWimp` library coding** or to add/delete functions to/from it. This is because:

(a) the package author has constructed the wimp-functions to serve a wide range of possibilities which may be demanded by different applications. The result is that the coding is often not simple and a change to a wimp-function needs to be done with full knowledge of the consequences to the whole package; and

(b) later versions of the library will not be aware of any changes you make - nor will your other programs made with the unaltered library.

However, our need to know what each wimp-function does is very well catered for by good documentation, including an on-screen browser - called `!Fnc'n'Pre`. In the early stages, it is worth having this active while you are working with Dr Wimp and it is useful 'homework' to start browsing it whenever you can. Section 3 of the Dr Wimp Manual contains the same information, if you prefer.

(Notwithstanding the above points, the `DrWimp` *library listing is in an easily readable form, so you can still examine the "how did he do that?" if you want to. Further, the example programs in the package are well commented to show what has been done.)*

Book tutorial project

We are now going to start constructing an application with Dr Wimp in a tutorial style - so as to introduce and examine its practical usage in a clear and comprehensive way.

The accompanying disc contains the full listings (and successive updates) of each step, as identified in the text.

The chosen tutorial application is not ambitious, but it is one we can all relate to and it will certainly give enough scope for its purpose. It is an application to allow a user to keep track of the fuel used by a car and to show the average consumption etc. in graph form, which can be printed out.

So, let's define the objectives:

Initiate a file record for a vehicle;

Enter, update and save fuel consumption data;

Show appropriate consumption averages;

Show consumption data graphically;

Enable the graph to be printed.

The above features to be available for more than one vehicle.

Initial planning

So, how will we want the program to operate?

To make this project representative of most applications, we will start it up by double-clicking on a directory icon and see an application icon appear on the iconbar. We will then use mouse-clicks and an iconbar menu to open up appropriate windows on the screen, leading to the main actions.

So let’s start the ‘hands-on’ and see how these steps emerge.

First ‘hands-on’ steps

We need to start with a few house-keeping matters. Firstly, open a new directory in a convenient place. Let’s call it “CarFuel”. Now copy **`!MyApp`** from the Dr Wimp package across to it. From now on we’ll be working in this new directory.

Open the copied **`!MyApp`** directory and note that it contains the following files:

```
!Run
!RunImage
!Sprites
DrWimp
```

The first three items are found in virtually every Wimp application and are part of the standard ‘application resources’ described in Appendix 3 in detail. You will need to become familiar with their respective roles and we will be looking at some of them below.

Load **`!RunImage`** into your favourite Basic program editor. As supplied, the program listing is headed by the few lines below *(but the details of the version and date, etc. may be different)*:

```
10 REM>!RunImage - for DrWimp Library Version 3.80
         **
20 LIBRARY "<MyApp$Dir>.DrWimp"
30 :
40 appname$="MyApp"
50 ver$="0.1 (31-Mar-03)"
60
```

```
70 ON ERROR PROCwimp_error(appname$,REPORT$+" at
         line "+STR$(ERL),1,1): PROCuser_error:
         PROCwimp_closedown:END
80 task%=FNwimp_initialise(appname$,7000,300,0)
90 PROCwimp_poll
100 END
```

followed by the user-function definitions - all of which are 'empty' (except for one case which we need not worry about yet). By 'empty', we mean that if they are **`DEF PROC`**s they contain nothing other than **`ENDPROC`** - and if they are **`DEF FN`**s they contain nothing but a return value of either 1, 0, -1 or a null string, according to their intended usage.

As we have already seen in **`!TestApp3`**, the programmer's task is to fill any of these, as needed in the particularly application - **but always being careful to leave the others untouched**.

From the above short listing you can see that the skeleton program merely loads the **`DrWimp`** library, sets an application title in **`appname$`**, and then calls three wimp-functions (held in the **`DrWimp`** library, of course).

Line 20 loads the **`DrWimp`** library and you'll notice that, conventionally, it refers to a 'system variable' **`MyApp$Dir`**, which is actually created in the **`!Run`** file (and the **`!Boot`** file) - before the instruction to run the **`!RunImage`** file. So, load the **`!Run`** file into a text editor also, to confirm this *(again, the details may vary)*:

```
Set MyApp$Dir <Obey$Dir>
IconSprites <MyApp$Dir>.!Sprites
WimpSlot -min 256k -max 256k
Run <MyApp$Dir>.!RunImage %*0
```

The first line sets the system variable **`MyApp$Dir`** and the last line runs the **`!RunImage`**. *(We need not worry about the other two lines at the moment. They are covered in Appendix 3 - but it is worth noting that the second line, indirectly, allows us to choose which sprite will be used to represent the application in the directory display.)*

Looking at Line 40 of the `!RunImage`, note that, as usual, the title given to the application lines up closely with the system variable name, the application directory name and, in the `!Sprites` file, a sprite name - all are variations of “MyApp”. So, our first action is to decide what to call our tutorial application and then modify Lines 20 and 40 accordingly (and the sprite, if we wish).

We have chosen “Fuel” for the (finished) application title. So, it makes sense to name this version `!Fuel4a` - because this is Chapter 4 and it is the first version of the application we have developed in this chapter. The system variable can be `Fuel$Dir` and - when we get to it - the application sprite will be called, correspondingly, `!Fuel4a`.

The `!Run` file thus needs to be modified to become:

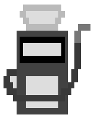
```
Set Fuel$Dir <Obey$Dir>
IconSprites <Fuel$Dir>.!Sprites
WimpSlot -min 256k -max 256k
Run <Fuel$Dir>.!RunImage %*0
```

Then the two lines of the `!RunImage` program need to be changed correspondingly to:

```
20 LIBRARY "<Fuel$Dir>.DrWimp"
40 appname$="Fuel4a"
```

Now save the new application as `!Fuel4a`, rather than `!MyApp`.

Finally, on the listings disc, the tutorial applications are all displayed with their own sprite (meant to be a petrol station fuel pump!).



So the housekeeping also involves adding this sprite to the `!Sprites` file and giving it exactly the same name as the application directory, but using lower case i.e. `!fuel4a` in this case.

The housekeeping is finished but, at the moment, our customised application will probably still appear with the old default RISCOS application sprite, If you now <shift-double-click> on that it should change to our unique petrol pump sprite.

It is best to carry out these house-keeping changes afresh at the start of the development of every new application - and as it is the first time we have come across them, we have changed them above ‘by hand’.

However, for later use, you may wish to note that the Dr Wimp package comes with a helpful utility application called `!Fabricate` which does automatically exactly what we have just done (except for providing a new icon, of course) - and it can do a lot more.

`!Fabricate` is in the ‘Utils’ folder of the `DrWimp` package and is also described in Appendix 6.

Error trap caution

You may have already noticed in the above short listing that the general Wimp error trap (at Line 50) is located before Wimp initialisation takes place (at Line 60). This means that you would be well advised not to alter (or put anything between) lines 50 and 60 during your use of the package. Any error occurring between or in those lines is very likely to lock your program into an endless loop - requiring a reset.

First run

Although our new application currently does nothing useful, it is important to understand and believe that the provided skeleton `!RunImage` is already a complete Wimp program.

To prove this and also to check that our few changes are functioning as they should, double-click on the `!Fuel4a` directory icon - and wrinkle your brow for a moment at the apparent absence of any action! But if you now open the Task Display (from the Task Manager's iconbar menu), you'll see that the application is actually up and running and occupying the space set in the WimpSlot line of the `!Run` file.

The only way to quit the application at the moment is to do so from the Task Display. Do this. (Click `<menu>` over the "Fuel4a" entry on the Task Display, then move off the menu to the right at the `Task 'Fuel4a'` item to get 'Quit'.) *With most RISCOS versions you can also quit using the* `<Alt-Break>` *facility.*

Logging on to the Wimp

Trusting that you are now a believer, let's have a slightly closer look at the listing so far. The all-important thing that happened on this first run, apart from loading the `DrWimp` library, was in line 60:

```
      60 task%=FNwimp_initialise(appname$,7000,300,0)
```

The wimp-function called in this line carries out a whole bunch of initialisation tasks as well as logging on to the Wimp (the Window Manager if you prefer, see Chapter 1).

The function returns a task identifier, or 'handle', to **`task%`**. Note that **`appname$`** is the first parameter - the Wimp requires to know the title of your application when you log on, *and this will be the name shown in the Task Display when the application is running.*

The number passed in the second parameter is to reserve memory space for window and icon definitions. *(The number is also used to dimension the general wimp parameter block in the program - and is always made to be at least one 'page' - 256 bytes - in size.)* At this stage the value 7000 (bytes) is safely large and we can ignore it for some time yet.

The third parameter is the minimum RISCOS version usable by the program, multiplied by 100. Here, version 3.00 is the minimum.

Note that this is not necessarily the RISCOS Version that you happen to be using yourself - although it could be. It is the lowest RISCOS Version that you want your users to be able to use when running your application on their computers. The significance is that a few of Dr Wimp's facilities can only be used if the user has a RISCOS Version higher than a given value. There are very few of these and the required RISCOS Version is clearly noted in the Manual. *(E.g. You need RISCOS Version 3.50 or higher to use Dr Wimp's 'colour picker' facilities - so you would need to use 350 in the third parameter above in that case.)*

As **`FNwimp_ initialise`** is the first wimp-function we have looked at in any detail, it is worth noting that it perhaps typifies the way Dr Wimp works. It is a simple Basic statement as far as we are concerned. But, behind-the-scenes, it carries out out a lot of essential, tedious stuff on our behalf.

If you look at the actual function definition in the **`DrWimp`** library, you will be able to detect that, among other things, it sets up various parameter blocks, including one for outline fonts, ensures the application gets service from the Wimp messaging system and checks that the right OS is operating. A lot of hair tearing avoided!

Finally, you will see that it also calls **`PROCuser_initialise`** - back in the **`!RunImage`** and, as yet, empty.

This peek also serves to demonstrate the point that the package author has designed a system to cater for many applications and this sometimes results in some complexity in the wimp-function coding. *(So leave well alone!)*

The Wimp poll

Line 90 contains the other main action of the skeleton application. It calls `PROCwimp_poll`, to start ‘asking the question’ repeatedly - see Chapter 1. A quick look at the definition of this `PROC` in the `DrWimp` library will again show that the package has a built-in structure to respond to the majority of the Reason Codes that will be returned.

Whilst peeking at this, also note that, just before `SYS "Wimp_Poll"` is called, a check is made on the state of three items to see if the application currently has any need for Reason Code 0 - the very frequently occurring Reason Code which merely reports ‘no action needed’ (see Chapter 1 and Appendix 2).

It is sensible to ‘mask out’ this code if possible and in a Dr Wimp application there are only three which need it - none of which apply to our Tutorial exercise. If none of these circumstances exist then Reason Code 0 is masked out automatically.

Listing reference: The few steps we have taken so far merely customise the skeleton application `!MyApp` supplied with the package - but it is our starting point, so is worth keeping for reference. It is `!Fuel4a` on the listings disc - indicating that it is the first complete application arising in Chapter 4. (See ‘Conventions’ in the Introduction)

Getting ready for successive versions

As we are going to develop several successive versions of the tutorial application it will be sensible to take account of this at the outset, before adding to the program functionally. The following points need to be catered for:

- It would be wasteful to include the `DrWimp` library in every version of the developing application. A single version will suffice, which will mean that the `LIBRARY` call (at Line 20) will need to be modified correspondingly. A sensible way to implement this is to copy what the Dr Wimp author has done - for the same reason - in the `Examples` directory of the Dr Wimp package i.e. create a 'mini- application' called `!DrWimpLib` which holds just the `DrWimp` library and a `!Boot` and `!run` file both containing the single line:

```
Set DrWimpLib$Dir <Obey$Dir>
```

Once this application is 'seen', the system variable `DrWimpLib$Dir` is set and holds the location of the `DrWimp` library - which can then be referred to in our application's `LIBRARY` call. It is best if `!DrWimpLib` is in the same directory as the various `!Fuel` application versions. It will then be 'seen' when you access these versions.

-A !`Boot` file should be added - to ensure that the application sprites are 'seen' (see Appendix 3). It needs only have the single line:

IconSprites <Obey$Dir>.!Sprites

-It will be helpful to put an extra copy of the application sprite - called, simply, fuel - in the sprite files. This will be used as the common iconbar icon throughout.

-It will be convenient to use line numbers for references in the text and, as the tutorial will add to the `!RunImage` progressively, **it is best if we use from the start the line numbers from the final**

listing. This will ensure that any references will remain constant throughout the chapters. ***(However, please note that some modern text/Basic editors use 'auto-renumbering' and this may not allow you to choose individual line numbers. Worse still, such editors may completely destroy the carefully preserved line numbering of a listing loaded from the supplied disc. Therefore, to follow the tutorial 'hands on', you are strongly advised to use .'Edit which is supplied free with every RISCOS computer.)***

- It will be helpful to see, on the iconbar, which version of the developing application is loaded. If we arrange this, there will then be no need to keep changing the value of `appname$` at the program start. We can use the name "Fuel" for all versions (although the application directory names will still need to be different of course).

All the changes necessary to achieve these points can be seen in `!Fuel4b` - plus the presence of the new mini-application `!DrWimpLib` in the same parent directory.

Listing reference: `!Fuel4b` is therefore the practical starting point for our tutorial.

Subsequent tutorial changes will therefore build from `!Fuel4b`. If you are following the tutorial 'at the keyboard' it is strongly recommended that you keep the supplied listings disc 'write protected' and work on copied versions. (In addition, **the tutorial !RunImage listings on the disc have all been locked** to reinforce this point. You can unlock the copies as required.)

Variable names

In the following chapters we will be adding various routines to the `!RunImage` and it is timely to mention some points about variable names.

Most importantly, if you look inside the `DrWimp` library you will see that nearly all the variables used there are made `LOCAL`. However, some are not - and these usually have names all in lower case and start with a “w”. These steps are deliberate and intended to prevent clashes with variables created by the programmer who is using Dr Wimp.

So the first rule is: don’t create new variables starting with a lower case “w” - leave that letter for the `DrWimp` library.

There are a few special global variables used by Dr Wimp which do not follow this normal pattern, mainly because they sometimes need to be used by the programmer directly e.g. `NULL%` *and* `UNUSED%`*. These are covered in the Dr Wimp Manual - and* `NULL%` *is also covered later in this book.*

Where this book creates other variables, it uses the policy of putting variable names in lower case but starting each ‘word’ or ‘sub-word’ with a capital letter e.g. `Main%`, `IconBar%`. The same policy is used in naming the app-functions (see Chapter 4 also).

These housekeeping details have been much longer to describe than to do but now they are out of the way, we can proceed to do something more visibly useful!

5. The Iconbar icon and its menu

Iconbar sprite

Our next practical step moves beyond house-keeping. When we double-click the application icon to start things up we want an icon to appear on the iconbar, signifying in the usual RISCOS way that the application is loaded and providing a user focus for getting access to the application's facilities.

With Dr Wimp, putting an icon on the iconbar is done very simply by calling one of the available wimp-functions. The one we want is:

```
    FNwimp_iconbar(sprite$,text$,maxlen%,pos%)
```

This function takes a named sprite, places it on the iconbar as an icon and returns a handle. The sprite needs to exist in the application's **`!Sprites`** file (or be in the Wimp sprite pool already - see Appendix 3) and then is identified simply by using its name in the first parameter **`sprite$`**.

If we want to put some text beneath the sprite, then that is the purpose of the parameter **`text$`**. (Make sure it is a null string if no text is required.) If we do want to use text we might also want to change it to longer text during the program run. In that case, the third parameter, **`maxlen%`**, allows us to specify the maximum length of text required.

If the fourth parameter **`pos%`** is 1 the sprite will appear on the right of the iconbar - if it is 0 it will be to the left.

So, starting from **`!Fuel4b`** (or a copy of it, called **`!Fuel5a`**), update the application directory name, sprites files and Lines 20 and 120 to '5a'.

Then add the following line to the !RunImage, within
`DEF PROCuser_initialise`:

```
     480 IconBar%=FNwimp_iconbar("fuel","V "
               +Version$,0,1) :REM** Creates and
               displays sprite on iconbar. **
```

This line also shows the practical effect of the housekeeping changes made between `!Fuel4a` and `!Fuel4b`. We now have two sprites in our `!Sprites` file: one whose name ending we have to change for each version (to keep the different versions correctly displayed in the application window) and one called `fuel` - which does not change its name. It is the latter one we have put into the first parameter here i.e. this sprite is always the iconbar icon.

The string manipulation used in the second parameter is to change the text beneath the iconbar icon in sympathy with our version numbering - it puts a “V” (with space) in front of the version number (from Line 120).

You may be surprised to see that the third parameter has been given the value 0 when the string in the second parameter will certainly be longer than zero. This is a small example of how Dr Wimp helps ‘behind-the-scenes’. An automatic check is always made to ensure that the third parameter is at least the same as the length of the actual string in the second parameter. Thus, as in this case we will not want to change this string during the program run, it is safe to use 0 in the third parameter. Dr Wimp will sort it out.

Finally, the handle returned to `IconBar%` in this case is, in fact, the window handle of the iconbar - which is always -2. This is used a little later.

If you now save and run the updated application, you will see the sprite duly installed on the iconbar, with “V 5a” written beneath it. It’s as easy as that! *(But you will still have to Quit via the Task Display, at the moment!)*

To emphasize that any sprite already in the Wimp pool could also be named in Line 480, the Tutorial in the package Manual suggests using `"!draw"` instead. Try it to prove the point to yourself.

Iconbar menu

Once we have the iconbar icon displayed, it is usual to offer a choice of user activities via an iconbar menu. So the next task is to create and display a menu.

Dr Wimp provides several ways to create menus (see Chapter 17) and at the moment we will use the simplest - which is by using the wimp-function:

```
FNwimp_createmenu(menu$,size%)
```

The parameter `menu$` needs to be used in a special way, which is best shown in the actual line we are going to use. Add this line, again within `DEF PROCuser_initialise`:

```
620 IconBarMenu%=FNwimp_createmenu("Fuel/Info/
        Quit",0)
```

As you can see, `menu$` is constructed as a list of the actual menu items you want to appear - each separated with a slash. Just note that the first item in this string is the menu heading and item No.l in the menu list will be the second item in the string i.e. "Info" here.

The second parameter sets the maximum number of items allowed in the menu - in case you want to add more later - and 0 means just allow space for the items listed. We will come back to this.

That is all we need to do to create a menu. The wimp-function returns the menu handle. It doesn't display the menu though - we want that to happen when we press the `<menu>` button over the iconbar icon. To arrange this takes us into our first use of the user-functions.

First use of a user-function

Whenever `<menu>` is pressed over one of the application's active windows, `DrWimp` calls `FNuser_menu(window%,icon%)` automatically behind-the-scenes.

Moreover, it makes the call with the actual 'live' parameter values of the window and icon handles over which `<menu>` was pressed.

If, in response, the user-function returns a valid menu handle then that menu is immediately displayed. Very simple but very powerful.

Initially, `FNuser_menu(window%,icon%)` is empty and returns 0 to any call from the `DrWimp` library - which results in no action. We therefore need to alter the contents of the user-function so that, when the passed window/icon parameter values are the ones we want, it returns the corresponding menu handle instead of 0.

It is much easier to show than describe. `FNuser_menu` is at Line 1590, and it is amended as follows:

```
1590 DEF FNuser_menu(window%,icon%)
1600 REM** Responses to <menu> mouse clicks. **
1610
1620 ReturnHandle%=0
1630
1640 CASE window% OF
1650
1660     WHEN IconBar%
1670     ReturnHandle%=IconBarMenu% :REM**
             Displays menu 'IconBarMenu%' when
             menu button clicked over iconbar
             icon. **
1680
1690 ENDCASE
1700
1710 =ReturnHandle%
```

Although this is very simple coding, we will nonetheless describe it in detail because it is typical of how nearly all the user-functions work and therefore important that it is fully understood early on.

Don't forget, this user-function will be called (from somewhere within the `DrWimp` library) every time the `<menu>` button is pressed over one of the application's active windows. When this happens, `DrWimp` automatically substitutes the current window and icon handles (over which `<menu>` was pressed) into the `window%` and `icon%` parameters before it makes the call. So 'live' parameter values are being passed to you, the programmer, to use as you wish.

In this particular case, we are concerned solely with the iconbar icon and the application has only one of these. Hence the passed icon handle parameter is not important this time - the window handle is enough.

Thus, when the window handle is the iconbar, we want the menu handle returned to be the one we created in the preceding section i.e. `IconBarMenu%`. Otherwise we return 0.

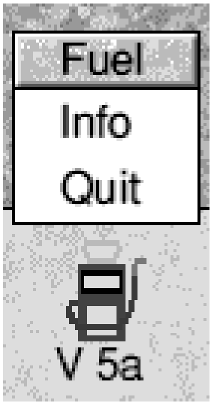
In Line 1660 therefore, we could have used:

```
      WHEN -2
```

but as the iconbar window handle was previously assigned to the variable `IconBar%` (see earlier) it makes easier reading to use this instead.

What our `CASE ... ENDCASE` amendment says is "When the window is the iconbar, the handle of the menu to display is `IconBarMenu%`". That's all there is to it!

Now run the application again and confirm that the menu duly appears, as shown below, in the normal place when you press `<menu>` over the iconbar icon.



If, instead or in addition, we wished to show a menu in response to a **`<menu>`** click over a particular icon in one of the application's ordinary windows, then we would simply need to use/add other **`WHEN`** statements to capture the appropriate window handle(s) and typically nest another **`CASE ... WHEN ... ENDCASE`** construction within each to capture the icon handle(s) required.

A more general structure might therefore be:

```
DEF FNuser_menu(window%,icon%)
ReturnHandle%=0
CASE window% OF
          WHEN WindowHandle1%
          CASE icon% OF
               WHEN IconNumber1%
               ReturnHandle%=MenuHandle1%
          ENDCASE

          WHEN WindowHandle2%
          CASE icon% OF
               WHEN IconNumber2%
               ReturnHandle%=MenuHandle2%
          ENDCASE
ENDCASE
=ReturnHandle%
```

Finally, it is of course possible that the user will press **`<menu>`** over a window's background, rather than over an icon. In this case, **`FNuser_menu`** is called (by the **`DrWimp`** library) with an icon handle value of -1.

Menu selection

By now, it will not surprise you to learn that whenever you make a selection from a displayed menu, **DrWimp** automatically makes a call to another user-function, this time:

```
PROCuser_menuselection(menu%,item%,font$)
```

which is at Line 2610. Change this **DEF PROC** to become:

```
2610 DEF PROCuser_menuselection(menu%,item%,
         font$)
2620 REM** Responses to menu selections. **
2630
2640 CASE menu% OF
2650
2660      WHEN IconBarMenu%
2670
2680      CASE item% OF
2690
2700           WHEN 2
2710           REM** "Quit" is menu item 2. **
2720
2730      PROCwimp_quit(0)
2940
2950      ENDCASE
2960
2970 ENDCASE
2980
3550 ENDPROC
```

(Note from the numbering gaps that quite a lot of lines will be added to this user-function in due course.)

With the previous detailed explanation of **DEF FNuser-menu**, the workings of **DEF PROCuser_menuselection** ought to be fairly self-explanatory. This time our amendment says "When item number 2 (second in the list, the top item being 1) is selected from the menu called **IconBarMenu%**, carry out **PROCwimp_quit(0)**."

We will look at `PROCwimp_quit` later but, for now, setting its parameter to 0 means that the application will quit from the desktop i.e. the usual way to end a Wimp program.

OK, so now run the application again - and try it.

> **Listing reference:** This is another useful milestone, so we will save the application at this stage as `!Fuel5a`.

Note that we have ignored the third parameter (`font$`) of:

```
PROCuser_menuselection(menu%,item%,font$)
```

This will be covered later in Chapter 17 under ‘Font menus’ - but in the above it will be a null string.

Take stock

As was explained, this chapter has deliberately covered a few fundamental features of Dr Wimp in some detail, because the working of the user-functions - albeit quite simple - is a little unusual i.e. they are called **from** the `DrWimp` library and automatically passing ‘live’ parameter values to **you**, the programmer.

Once the penny drops, the real strength of Dr Wimp is appreciated.

The chapter has also introduced the importance of identifying the right wimp- and/or user-function and then using it (in a very simple manner). This is typical of 90% of the hands-on work of using Dr Wimp. So it is important to get browsing among the function list in the Dr Wimp Manual (Section 3) or by using `!Fnc'n'Pre` (its on-screen equivalent) to get to know what is available.

Because of the meaningful and consistent naming of the wimp- and user-functions, the learning curve to acquire some fluency and confidence in using the package is surprisingly short.

You need to be reasonably adept at Basic, but not expertly so - because all the difficult stuff goes on behind the scenes.

6. Adding windows and icons

To develop the tutorial application further, we need to create/design some windows with icons and then start to use them in the program.

Designing windows/icons

In Chapter 2, we saw that the creation/design can be done directly from within the program or by using window definition templates - and that, generally speaking, the latter is far more convenient. Dr Wimp provides facilities for both methods and, whichever approach is chosen, the creation/design process is a step which is largely independent of the more mechanical process of loading/displaying/managing the resulting design.

At this stage, we are going to use the template method and, whether using Dr Wimp or not, window templates are created in a template editor which prepares the window-plus-icons definition in a file format specified by RISCOS in the PRM. Traditionally, the template file becomes one of the ‘application resources’ - see Appendix 3 - and is normally held in the application directory.

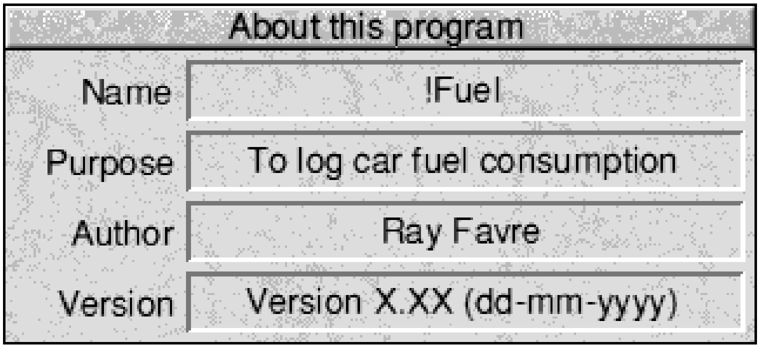
There are several template editors available - all of them graphics editors - and the output of any of them (for a given window) will be the same. Thus, the one to use is a personal choice. But the Dr Wimp package includes a very good one (`!TemplEd`, a PD utility by Dick Alstein) and this book shows how it is used in Appendix 7 - specifically taking one of our tutorial windows as its descriptive vehicle.

That Appendix is closely linked with Appendices 4 and 5 which look at the important subject of window and icon ‘button types’ and icon validation strings.

So now is the time to divert to those appendices - because the following words will assume that you already have a templates file in the same directory as the `!RunImage` and that it contains, at least, the definition of the following Info window - and that you understand the concepts of icon and window button types, icon numbers and ‘indirected’ text.

These are vital topics: do not proceed further into this book without taking them on board!

The Info window


About this program

Name	!Fuel
Purpose	To log car fuel consumption
Author	Ray Favre
Version	Version X.XX (dd-mm-yyyy)

As you can see, this window is typical of the information box that you see when you move the mouse pointer to the right across the top item (invariably called ‘Info’) of the iconbar menu of nearly every application.

Examining it in more detail - as it is the first window of our tutorial - it is a small window with a title bar but no scroll bars, close icon, back icon, size toggle etc. In total, there are eight icons holding text - all of button type ‘Never’, because we need no interaction with the mouse for this window.

There are four icons on the left (icon numbers 0-3, from top to bottom) with only short, single words as their text. These are without a border or a background - so the effect is as if the text is written directly onto the window. The text in these acts as labels for the other four icons (4-7), which have 3D effect borders and contain the program information.

In this example, the text in the four label icons (0-3) is fixed text included as part of the icon definition and cannot be changed by the program. However, the main information icons (4-7) contain ‘indirected’ text and so can be altered by the program.

The maximum length of the indirected text allowed in icons 4-7 is, in this case, just sufficient to hold the items shown in the above screenshot i.e. 6, 28, 10 and 26 characters for icons 4-7 respectively - each one being one more character than the visible text, to allow for the string terminator. There is no reason why you cannot alter these values (in the template editor) if you wish.

Similarly, provided that they are created as indirected text icons of sufficient maximum length, there is no need to put any actual text in the template. It has been done here solely to show the intended text formats at the maximum lengths.

With this Info window in place within a file called “Templates” we can now start to use it. What we want to do with this particular window is fairly modest - just to display it from the iconbar menu.

Loading a window

The first thing we have to do with any window held as a template, is to load it into our application. Provided this is done after **`FNwimp_initialise`** and before **`PROCwimp_poll`** (and before you want to display the window!) the precise location in the program is not too critical. However, it makes sense to include it in **`PROCuser_initialise`** and the following addition does this:

```
520 Info%=FNwimp_loadwindow("<Fuel$Dir>.
     Templates","Info",0)
```

The first parameter of this wimp-function is the full path name string of the template file in which the required window is stored, and the second is the specific window definition name string.

The third parameter tells the program where to find any sprites which may be used in the window. To the `DrWimp` library, 0 actually means “use the Wimp sprite pool” but as we are not using any sprites here, 0 is also a convenient default value. If we were using our own sprites, we would need to create a ‘sprite area’ - which Dr Wimp has facilities to do (see Chapter 19) - and we would then put the address of the sprite area in this third parameter.

The function returns a handle for the window and the meaningful name **`Info%`** has been chosen in the above line to store this.

This simple wimp-function hides an awful lot of routine hassle behind the scenes in the `DrWimp` library. Not only are the straightforward physical window parameters loaded (e.g. size, colours etc.) but also any outline fonts used in the template are correctly loaded in and ‘opened’ ready for use (one ‘opening’ per font per size) and in a manner which will not conflict with other template loadings or other separate uses of fonts in the application. Similarly, all icons in the window template are correctly loaded and identified to the Wimp. Font handling, in particular, can be tricky to get the hang of, so Dr Wimp really does a good job in making it painless.

Attaching a window to a menu

As is normal with Info windows, we want it to be displayed - in the same way as a sub-menu is displayed - whenever a user presses <menu> over the iconbar icon and then moves the pointer off to the right of the “Info” item on the menu (menu item 1, the topmost item).

Again, this is painless with Dr Wimp as there is a wimp-function to do it. It is:

```
    PROCwimp_attachsubmenu(menu%,item%,submenu%)
```

which is almost self-explanatory.

This function attaches a sub-menu (or a window) to an already defined menu. The first and second parameters are, respectively, the handle of the menu and the specific menu item to which the attachment is to be made - the top menu item is 1. The third parameter is the handle of the sub-menu (or window) which is to be attached.

The required line in our case is therefore:

```
760 PROCwimp_attachsubmenu(IconBarMenu%,1,Info%)
```

That is, after the menu IconBarMenu% has been created.

If you now save `!RunImage` and run it you will find that the top menu item now has the usual right-pointing arrowhead and the window will appear as required - albeit still with the text from the template.

Putting/changing text in icons

Yet again, a simple wimp-function is in the Dr Wimp arsenal to change the text in any indirected icon. The wimp-function is:

```
PROCwimp_puticontext(window%,icon%,text$)
```

This really is self-explanatory!

(If you are new to Wimp programming, now is the time to note that icons can only hold text or sprites or both. They do not hold numbers: all numbers need to be converted to string form first before being displayed in an icon.)

So, in our case, we need to call it four times - once for each information line (i.e. icon) in the Info window, whose handle is `Info%`. You will remember that the relevant icon numbers are 4-7 from top to bottom,

There are several ways we could make these four calls. A popular way, for the `Info` window, is to put the required text into a `DATA` line and `READ` it for each icon using a loop. This method, you will recall, was used in `!TestApp1` and `!TestApp3`.

This time, however, it will serve the tutorial purposes better simply to call the wimp-function four times directly. The lines to add are therefore:

```
800 PROCwimp_puticontext(Info%,4,"!Fuel")
810 PROCwimp_puticontext(Info%,5,"To log fuel
         consumption")
820 PROCwimp_puticontext(Info%,6,"Ray Favre")
830 PROCwimp_puticontext(Info%,7,"Version
         "+Version$+" ("+VersionDate$+")")
```

The fourth line automatically copes with our version numbering changes. You can make the text changes here to suit your own wishes, of course.

The only thing you need to watch is the indirected text length limits built into these icons in the window template. If you exceed them, some strange effects may happen - typically the desktop starts to revert to 'system font' if you have it set to an outline font.

Save and run the program again to check that the new text is displayed correctly.

`PROCwimp_puticontext` has some other important points which need to be taken on board:

- In the use we have just made of it, the corresponding window was not open when the call occurred. It could not be because the window happens to be linked to a menu. The point to note is that this is perfectly OK. After it has been loaded/created, the text of an icon can be changed at any time, provided the icon text is 'indirected' of course. The new text will then show when the window is next opened.
- The corollary is that, if the corresponding window is open when the call is made, the text in the icon will be changed immediately in the display. This is another example of Dr Wimp shielding you from all the complication.

- If an icon is indirected and contains a sprite instead of text (e.g. like the draggable file icon in the standard Save window) then PROCwimp_puticontext can be used to change the sprite, in exactly the same way as for changing text. In this case, the sprite name - in string form - is put in the third parameter e.g. “file_ffb” to change the icon sprite to the standard Basic file icon.

Listing reference: This is another useful milestone, so we will save the application at this stage as `!Fuel6a`.

7. More windows

Although nearly all applications have one, the Info window is not a typical window. It is generally only used as an information display and does not itself become involved with other user interactions. So now is the time to start moving into the main uses of windows.

The windows needed

Our tutorial will need three other windows to carry out the tasks we outlined in Chapter 5:

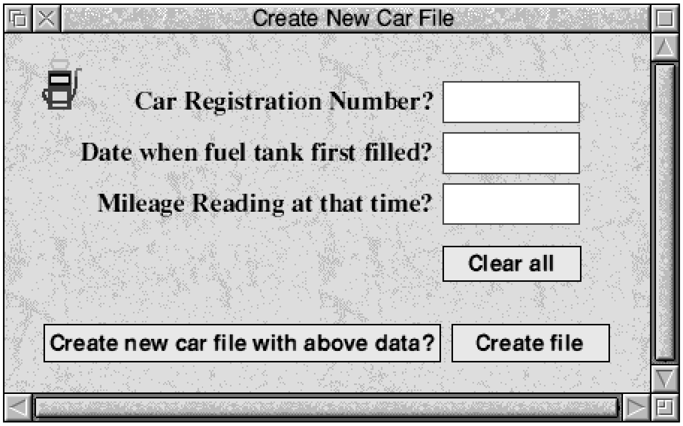
- one to create separate fuel records for each vehicle;
- one to permit each vehicle fuel consumption to be updated
- one on which to draw a selected vehicle’s fuel consumption graph.

We will call these windows NewCar, FuelUpdate and FuelGraph respectively and we will introduce them in this same order.

This chapter will deal with the NewCar window.

The NewCar window

This window has been added to the **`Templates`** file in **`!Fuel7a`** and looks like this:


Create New Car File
Car Registration Number?
Date when fuel tank first filled?
Mileage Reading at that time?
Clear all
Create new car file with above data?
Create file

Again, this was designed in **`!TemplEd`** - see Appendix 7. Its main characteristics are:

- The window work area button type is ‘Never’.
- There are ten icons (0-9), which carry all the text (indirected) and the decorative fuel sprite.
- Outline fonts are specified for the text in all text icons, using the F-command in the validation strings (again, see Appendix 7). *(Outline fonts supplied with all RISC OS computers are used.)*
- The three writable icons (0, 1 and 2) use the A-command in their validation string to restrict the allowable user-entry characters (see later also). Their button type is ‘Writable’, of course, and the pointer changes to a thin blue cursor as it moves onto them (via the P-command in the validation string).

- The `Clear all` and `Create file` icons are of button type 'Menu', which selects the icon (highlights it) as the pointer moves onto them and responds to mouse-clicks.
- Background and foreground colours have also been used to highlight action icons more prominently (icons 0, 1, 2, 6, 7 and 8) - again using the F-command.
- It is essential to remove any K-command validation string from the three writable icons (icons 0, 1 and 2), otherwise the caret movement may not work as later arranged.
- It is helpful, for programming, to number the writable icons 0, 1 and 2, as per the template and for the other three coloured text icons to be numbered consecutively (6, 7 and 8 here).
- If you play with the template, make sure none of the icons is selected in the template display before it is re-saved otherwise it will not look right when first opened by the application.

Intended operation

This window will be opened from the iconbar menu and its three writable icons will start off empty with the caret in the top one. The other three coloured text icons will start off disabled i.e. they will be 'greyed out' and give no response.

The vehicle details are to be entered by the user into the three writable icons using keyboard entry. The `<return>` key or `<up/down>` keys will be used to cycle the caret among the three icons at will whilst they remain empty. However, once an entry is made in any of the three icons, a validation process on that entry will commence as soon as `<return>` or `<up/down>` is pressed and the caret will not be allowed to leave that icon until a validated entry is completed (or the user once again empties it).

Only when all three writable icons hold validated entries will the three other coloured icons be enabled, allowing the user to confirm that a new file should be opened - or, alternatively, that all the icon entries are to be cleared. (Individual entries will still be able to be amended during this stage.) Once a new file has been opened, all window entries will be cleared and the display will revert to its starting state.

Displaying the window

We need start the action by displaying the new window on the screen. This is straightforward once we have it saved in the Templates file. First, the window needs to be loaded, exactly as with the Info window earlier and this is best done in the very next line after that, thus:

```
530 NewCar%=FNwimp_loadwindow("<Fuel$Dir>.
         Templates","NewCar",0)
```

Secondly, because we are going to activate this window from the iconbar menu, we now need to revisit that menu and make a modification to our existing program.

So, change Line 620 to:

```
620 IconBarMenu%=FNwimp_createmenu("Fuel/Info/New
         car/Quit",0)
```

which just adds an extra menu item "New car". However, in doing so, it changes the menu position of 'Quit' from item 2 to item 3, so we also need to change our existing menu-selection code to reflect this. Therefore, change Line 2700 from `WHEN 2` to `WHEN 3`.

So, our new menu item 2 is "New car" and when this is selected we want the new window to open. This is achieved by adding a new `WHEN 2` sequence to `PROCuser_menuselection`, using yet another wimp-function, whose general form is:

```
PROCwimp_openwindow(window%,centre%,stack%)
```

where `window%` is the handle of the window to be opened and `centre%` is 0, 1 or 2 and determines where the window is placed on the screen. 0 means where it was last opened (or as in the template definition if the window has not yet been opened); 1 means centred on the screen and 2 means centred on the pointer.

`stack%` is a handle of another window - the one to open the new window behind. Alternatively, -1 is used to open the new window on top of all the rest, or -2 to open it at the bottom, or -3 for the current stack position.

Therefore, the new sequence to be added is:

```
2770 WHEN 2
2850 PROCwimp_openwindow(NewCar%,1,-1)
```

(You may think that things are now out of order in the `CASE ... ENDCASE` *construction, but all will become clear later.)*

You should now run the new program to check that the “New car” window opens as expected, but there is a lot more to do before it can be used.

Enabling/disabling icons

From using other applications you will be familiar with the fact that icons (and menu items) are often ‘greyed out’ to prevent their use at particular stages in the application and/or if a feature is not available in the version you are using. Dr Wimp provides a wimp-function for this:

```
PROCwimp_iconenable(window%,icon%,state%)
```

The first two parameters are self-explanatory. The third parameter, `state%`, is set to 0 to disable the icon or 1 to enable it.

We said earlier that the starting state of our new window would have three of the coloured text icons disabled - so we need to effect this as soon as the window is opened. Immediately after the above `PROCwimp_openwindow` line (i.e. still in the `WHEN 2` sequence).

We could simply add:

```
PROCwimp_iconenable(NewCar%,6,0)
PROCwimp_iconenable(NewCar%,7,0)
PROCwimp_iconenable(NewCar%,8,0)
```

However, we shall soon want to do a few other things at this point also, so it makes structural sense to put all such actions into a new and separate procedure - our first app-function - as follows:

```
7600 DEF PROCapp_ClearUserIcons(Window%)
7650 CASE Window% OF
7670     WHEN NewCar%
7680     FOR Icon%=0 TO 2
7700     PROCwimp_iconenable(Window%,(Icon%+6),0)
7730     NEXT
7880 ENDCASE
7940 ENDPROC
```

and we call this new app-function with:

```
2870 PROCapp_ClearUserIcons(NewCar%)
```

Again, you may feel that the coding is somewhat convoluted to produce the icon numbers 6, 7 and 8, but there is a reason which will become clear.

Now when you run the program you will see the effect - the window opens with the three lower coloured icons ‘greyed out’.

Caret placement

We also said that the caret would start off in the top writable icon, which is icon 0, and yet again a wimp-function is provided. Add the next line:

```
2880 PROCwimp_putcaret(NewCar%,0)
```

which says “Put the caret into icon 0 of the `NewCar%` window”.

In operation, this action will also give the window the ‘input focus’ - which will be shown by the window’s title bar becoming pale yellow (see later, also).

Don’t forget that you can only use this wimp-function if the window is open.

As we said under ‘Intended operation’ above, we are also going to make the caret cycle round the writable icons in order to help the user ensure that all three writable icons get valid entries before further action can take place.

> *Please note that different RISCOS versions have handled caret movement among writable icons in different ways, with later versions being easier for the programmer. But our tutorial will adopt a method which should work identically for all RISCOS versions.*

Key presses

With the window now in its initial state, the user is required to use the keyboard to make data entries. The `<return>` and `<up/down>` keys are used to signify the end of an entry and/or to move the caret and will also trigger our specific entry-validation and caret control processes.

All Wimp programs need a routine to handle user keypresses and Dr Wimp makes does this via the user-function `FNuser_keypress`.

This user-function is called automatically from within DrWimp every time a key is pressed - if our window has the ‘input focus’.

The general form is:

```
DEF FNuser_keypress(window%,icon%,key%)
=0
```

and the live values of the relevant window and icon handles and the ASCII code of the keypress are passed to you in the three parameters.

If we don't need to use the keyboard at all - or only want to use it to fill a writable icon - we simply leave this function in its default state, which means it is 'empty' and returns 0. However, if we want to specify some other action to be taken on a keypress (e.g. like some specific caret movement or extra validation) then we define that action in the user-function and arrange for it to return 1 when the appropriate keypress occurs.

In our case, we want to use the `<return>` key (ASCII 13) and the `<up/down>` keys (ASCII &18E and &18F) to initiate a special validation check and to control the caret movement in a particular way.

Therefore, the following generalised sequence is appropriate:

```
DEF FNuser_keypress(window%,icon%,key%)
used%=0 :REM** Needs to return 0 if keypress not used.
CASE window% OF
   WHEN NewCar%
   CASE icon% OF
       WHEN 0,1,2
       CASE key% OF
          WHEN 13,&18E,&18F
          used%=1 :REM** Needs to return 1 if keypress
            is used for some action. **
          PROCapp_Validate:REM** Validate current
            entry.
          PROCapp_MoveCaret:REM** Cycles caret.
          PROCapp_CheckAllEntries :REM** Checks if all
            three entries validated. **
       ENDCASE
   ENDCASE
ENDCASE
=used%
```

Note that this routine responds only to the `<return/up/down>` keys - anything else is either entered into the icon (if it passes the icon's validation string criteria) or is ignored.

Using this structure for guidance, a specific routine is entered at Line 2180 for your examination. You will note that it is structured to be used by more than the `NewCar%` window and we will return to this later.

(Chapter 26, Appendix 5 and the Dr Wimp manual contain more detail about using keypresses and caret control among writable icons.)

Also needed is the coding for the new app-functions represented above by `PROCapp-Validate`, `PROCapp_MoveCaret` and `PROCapp_CheckAllEntries` - which will carry out the keyboard entry validations.

For these there is first a need to set up some flags and arrays, which are appropriate to `PROCuser_initialise` and more neatly done by calling a new app-function from within it. This new app-function is `PROCapp_FuelInit`.

`DEF PROCapp_FuelInit` is started at Line 4590 and the single call to it is placed at Line 450. It would risk confusion to list the complete `DEF PROC` here, so we will mention the various important lines as we need them. The first are:

```
4640 DIM Valid%(2)
4650 DIM Valid$(4)
```

The elements in array `Valid%()` hold either `TRUE` or `FALSE` and are flags indicating whether or not the user entries into icons 0, 1 and 2 have been validated. They are initially set to `FALSE` by other lines in the `DEF PROC`.

`Valid$()` holds the validated strings from each user-entry icon. This array has more elements than `Valid%()` because we are also going to use it for the `FuelUpdate` window which has extra icon strings to hold. The elements are 'sized' to 10 characters and then set as null strings, by other lines in the `DEF PROC`.

Both these arrays will also need to be reset to their starting states after successful creation of a new vehicle file - ready for another possible new vehicle. `DEF PROCapp_ClearUserIcons` is changed to do this.

With these arrays in place we can start to construct a validation sequence.

User-input validation

Validation of user keyboard entries is an important task for nearly all programs.

With Wimp programs, validation of user entries in writable icons is made very much simpler by the very comprehensive ‘first line of defence’ facilities offered via the icon’s ‘validation string’ (see Appendix 5). This enables us to restrict at entry which characters can be accepted from the keyboard and also the maximum number of characters allowed - for each icon individually.

This is a considerable help but, even so, our application will still need a fair amount to be done on the ‘second line of defence’.

As this book is about Wimp programming and Dr Wimp in particular, we will not be going through the specific validation processes needed for our tutorial application in detail. However, using `!TemplEd` you will be able to see the validation strings used for each icon via the `Templates` file on the disc.

Similarly, the (fairly extensive) programming of the ‘second line defence’ can be seen in the `!RunImage` listing for `!Fuel7a` which is well-commented. Several new app-functions have been added - in accordance with those indicated a little earlier.

Nonetheless, it is sensible to give a brief overview of the validation needs here as they are quite typical of what might be needed in any program:

Vehicle Registration Number (VRN) (icon 0 of NewCar window). We can’t do much validation here, because the valid variations are considerable. VRNs usually comprise small groups of capital letters and/or the numbers 0-9, separated by single spaces - so we can usefully allow only these characters with the validation string (so don’t forget that `<shift>` will be needed to enter letters).

Beyond that, we will eliminate any leading and trailing spaces and also any occurrences of more than one space consecutively.

Further, we will set an arbitrary rule that the total length of the VRN must not be **less** than 6 letter/number/space characters in total - but this is easily changed if you own “HRH 1”!

Date (icon 1). We may well later want to extract numerical data from the date, so there are many advantages if we fix the date text format rigidly at the user-entry stage. We will use the form `dd/mm/yyyy` - but will allow the user to enter single numbers for day and month and will automatically add leading zeros to keep the format constant.

We can therefore use the validation string to restrict the date entries to the digits 0-9 and the “/” symbol, but we need to check for at least two “/” characters, correctly disposed. For the year, we will limit it to the range 2000-2050 and, of course, appropriate date and month range limits.

(All this takes a lot of program lines, as you can see from the listing starting at Line 10430.)

Mileage (icon 2). Here we are talking about the odometer reading, so we can easily restrict ourselves to integer numbers with no more than six digits. It is worth noting that 0 can be a valid entry and needs to be distinguished from an empty (null string) icon.

Don’t forget (see earlier in this chapter) - the aim is not to let the user leave an input icon (i.e. leave `PROCapp_Validate`) until a valid entry (or an empty icon) exists. So, each time `<return/up/down>` is pressed, the program needs to evaluate the string currently in the icon. If it is not valid, the user needs to be warned and told why and the caret needs to be put back into that same icon - without deleting the invalid entry.

The programming therefore makes copious use of calls to `PROCwimp_error` to provide helpful explanations of why the entry is not valid e.g. `"Too many days for the month"` etc.

Some specific wimp-programming points

Although we will not be looking into `PROCapp_Validate` in detail, there are a few wimp-programming items that need comment.

Reading text from an icon

The routines include the use of the wimp-function:

```
FNwimp_geticontext(window%,icon%)
```

This is a `FN` and it returns the text from the specified icon - again, it is that simple.

Like its counterpart `PROCwimp_puticontext`, an icon's text can be read at any time after it has been loaded/created - but in the reading case it does not have to be 'indirected' text.

As mentioned earlier, remember that all 'numbers' read from an icon will actually be text strings and hence conversion between the two will often be needed e.g. using `VAL` and `STR$`.

Line 9450 shows a typical call to read the text from an icon.

Moving the caret

After a valid entry has been achieved in any one of the three icons, we want the caret to move in a logical way to the next icon and to cycle around the three potentially available. As we said earlier, the method adopted here should work for all RISCOS versions.

`PROCapp_MoveCaret` at Line 7980 does this and it is straightforward - although it has been generalised to allow it to be used in more than one window and for any three consecutively-numbered icons. This routine may not work correctly for all RISCOS versions if any of the writable icons have a K-command in their validation string (see Appendix 5). so please ensure that none is present.

It should be noted that the corresponding window needs to be open before a `PROCwimp_putcaret` call will work - as, unlike text, a caret cannot be put into an icon unless the window has the input focus.

Enabling icons

As seen in the earlier outline structure of `DEF FNuser_keypress`, after every `<return/up/down>` keypress we need to check to see whether all three user input icon entries are yet validated. If they are, then the user can be allowed to proceed to creating a new vehicle file - and we duly signal this to him/her by enabling icons 6, 7 and 8 to present the corresponding options.

`PROCappCheckAllEntries` at Line 8350 does the necessary actions. Note that it needs to 'enable' and 'disable' the icons in an `IF THEN ELSE ENDIF` construction. This is a very common type of need.

Checking the changes

If you run the application as it now stands, and open the `NewCar` window from the iconbar menu, you will be able to make entries into the three input icons to try out the validation processes in each one.

When you have a validated entry in all three user-entry icons the `Create file` and `Clear all` icons will become enabled, inviting the user to click on one of them to take the next step.

At the moment, nothing will happen if you click on them and we will address that in the next chapter.

Listing reference: We have added quite a lot of lines in this chapter and reached another useful milestone So we will save the application at this stage as `!Fuel7a`.

You will probably need to spend some time on this particular listing before continuing, because we have not given detailed descriptions here.

8. Mouse clicks

(This chapter concerns mouse-clicks over windows and icons only. Mouse-clicks over menu items have already been introduced under menu selection in Chapter 5)

The principles

Mouse clicks over windows and icons are, of course, a major part of the user interactions in Wimp programs and, before continuing with our tutorial program, it may be helpful to explain how Dr Wimp handles them and how this differs from the fundamental Wimp treatment.

(However, this explanation is not critical to an understanding of Dr Wimp and you can skip straight to 'The practice' - on the next page - if you prefer!)

The Wimp itself treats clicks with the **`<menu>`** button in a different way to **`<select>`**/**`<adjust>`** clicks. As Appendix 4 indicates, the window/icon button type is used to define the Wimp's precise response (which can be 'no response') to **`<select>`**/**`<adjust>`** actions whereas the Wimp always responds to **`<menu>`** clicks.

Dr Wimp takes a similar but simpler approach. Firstly, it handles **`<menu>`** button presses just like the Wimp and we have already seen how this is done via **`DEF FNuser_menu`** in Chapter 5. This was very simple and needs no more explanation here.

However, for **`<select>`** or **`<adjust>`** actions, it takes the view that, in the vast majority of cases (whether clicks on windows or icons), the key factor is which button has taken the defined mouse-click action, rather than what, precisely, that action was.

Thus, if an icon is of button type 'Double-click' the Wimp will ensure that there will only be a response from it if `<select>` or `<adjust>` is double-clicked over it. Similarly, an icon of button type 'Click' will give a response to a single click from `<select>` or `<adjust>`.

Dr Wimp provides a single user-function to handle both these cases - telling the programmer which button took the action but not distinguishing between the single- or double-click actions.

It would be a similar result if button type 'Release' was used, A response would only occur on button release, but Dr Wimp's single user-function would be used just the same.

In the vast majority of applications this is not a limitation. In fact, it is usually a welcome simplification.

When it comes to drag actions, Dr Wimp provides facilities for saving or loading file data (see Chapter 18) - which are the most likely uses of dragging - but does not offer a general dragging facility.

The practice

As we shall see, the practical restrictions are very few indeed and the implementation is extremely easy.

The user-functions provided is:

```
DEF PROCuser_mouseclick(window%,icon%,button%,
     workx%,worky%)
```

This is called by the `DrWimp` library whenever a `<select>` or `<adjust>` mouse click is made and it is up to the programmer to fill the `DEF PROC` as required, using the 'live' values passed in the parameters.

The parameter `button%` holds 1 if `<adjust>` was pressed and 4 if `<select>` was pressed. These may seem odd values but they are those the Wimp uses and they make sense in binary if you visualise the three buttons and represent each by a single bit in a three-bit number: 100 for the left button and 001 for the right - get it? *Yes, the Wimp does produce 2 for the middle button* `<menu>`*, but this is 'tied off' in* `PROCuser_mouseclick` *and activates* `PROCuser_menu` *instead.*

The last two parameters, **`workx%`** and **`worky%`**, give the mouse pointer position when the click was made - relative to the top left corner of the window. These values can therefore be used by the programmer, if required, and they are probably of most use when the window work area button type is not zero, rather than for icons.

As has been stated in an earlier chapter, if the pointer is not over an icon when the button action occurs, the icon number in the second parameter will be -1.

Back to the tutorial application ...

To see how easy it is in practice, in our tutorial application we only need to add a typical and simple mouse-click routine, as follows:

```
1070 DEF PROCuser_mouseclick(window%,icon%,
         button%,workx%,worky%)
1100 CASE window% OF
1120     WHEN NewCar%
1140     CASE icon% OF
1160          WHEN 8 :REM** 'Create File' icon. **
1170          PROCapp_CreateNewFile
1180
1190          WHEN 7 :REM** 'Clear All' icon. **
1200          PROCapp_ClearUserIcons(window%)
1210          PROCwimp_putcaret(NewCar%,0) :REM**
                Put caret in top user-entry icon.
                **
1220     ENDCASE
1530 ENDCASE
1550 ENDPROC
```

That is, if icon 8 is clicked a new car file is created and, if icon 7 is clicked, all the icons are cleared of text - and for the latter, we can re-use `PROCapp_ClearUserIcons`.

That is all there is to handling mouse-clicks.

The routine would have been identical if icons 7 and 8 had been, say, of button type ‘Double-click’ or ‘Release’. The only thing that would be different is that there would be no response from the icon unless a double-click had occurred, or when the button release occurred, respectively.

We now need to create `DEF PROCapp_CreateNewFile` and the first task is to make some enabling additions to `PROCapp_FuelInit`:

```
4980 SourceDir$="<Fuel$Dir>.Vehicles"
4990 OSCLI "CDir "+SourceDir$
5010 FileHeader%=32
5020 RecordLength%=48
5040 NewCarWindowOpen%=FALSE
```

Lines 4980 and 4990 ensure that a known directory exists to hold the vehicle file records - which, for good reasons, we are going to save directly without using a ‘Save box’. The use of the star command `CDir` is such that it only creates the directory if one does not already exist - so there is no difficulty in using it every time the program is started.

Lines 5010 and 5020 are housekeeping tasks to keep the vehicle files in the same format: each will have a small header and each record will be of a fixed length. *(These features are not essential and have been chosen merely for simplicity.)*

Line 5040 is a flag to keep track, during the program run, of whether the `NewCar` window is open or closed. Similar flags will be created for the other windows in due course.

`PROCuser_openwindow` and `PROCuser_closewindow` are used to toggle these flags between `TRUE` and `FALSE`. The former is called from the `DrWimp` library just after a window has been opened - passing the window handle, the screen coordinates of the top left corner of the window (in OS units) and the window stack position. The latter does the same thing just after a window has been closed. The necessary additional lines are shown in the `!Fuel8a` listing.

Creating the file

`PROCapp_CreateNewFile` is fairly straightforward. Firstly, because there is no limit to the craftiness of users, it is necessary to run a quick check before creating a new file to ensure that the user hasn't altered the validated entries after the **`Create file`** icon has been enabled but before it has been clicked.

So, **`DEF FNapp_DoubleCheck(Window%)`** at Line 7120 does this and raps the user's knuckles if this effrontery occurs. All the **`Valid%()`** array elements are then put to **`FALSE`** - but the icon entries are not deleted though. This explains the need for the array **`Valid%()`** - the validated entries are stored there and the current contents of each user-entry icon are compared with their corresponding array element as the double-check. You will note that, as with some other cases, this **`DEF FN`** has been prepared for later use in other windows.

If a change in the icon text is detected, the **`FN`** itself returns **`FALSE`** and **`PROCapp_CreateNewFile`** exits (at Line 5510) before doing anything else.

Next, **`PROCapp_CreateNewFile`** decides what to call the new file. Anticipating a little ahead, we want to be able to handle more than one vehicle, so all vehicle files will be named sequentially as **`Car1`**, **`Car2`** etc. The routine starting at Line 5610 tries to **`OPENIN`** each existing **`"Carx"`** file in turn and stops when the channel number is 0 i.e. when no such file exists. The first one that doesn't exist is hence the name of the new one, the next in the sequence.

This new file is then **`OPENOUT`** and the items from user-entries in the **`NewCar`** window are put into the start of the file as a header and the header is padded out to the value **`FileHeader%`** now defined in **`PROCapp_FuelInit`** - and the file is closed. When we later add fuel consumption data to the file it will start after this header.

> **Listing reference:** This is another useful milestone, so we will save the application at this stage as !Fuel8a.

> *This chapter has only introduced a few new user-functions but has actually covered the basics of the most frequently used of all the user interaction, namely mouse-clicks - and how simple Dr Wimp makes it.*

9. Dynamic menus

Having created a new car file, we need to give the user access to it for adding fuel data. This is best done by putting the vehicle's VRN into the iconbar menu and it would be sensible to allow for more than one vehicle on file. Thus we have a need for a 'dynamic' iconbar menu i.e. its size and content will be determined by how many vehicle files are held at the time the menu is displayed.

Dr Wimp has several wimp-functions to help us effect this in different ways and Chapter 17 is devoted to menus in more detail. However, for now, it is sufficient to note that there are wimp-functions to modify a single item on an existing menu. For example:

```
PROCwimp_putmenuitem(menu%,item%,item$)
PROCwimp_removemenuitem(menu%,item%)
PROCwimpputmenutitle(menu%,title$)
PROCwimpmenutick(menu%,item%,state%)
etc.
```

And there are others to recreate an existing menu entirely:

```
PROCwimp_recreatemenu(menu%,menu$)
PROCwimp_recreatemenuarray(menu%,array$())
PROCwimp_recreatemessagemenu(menu%,
            messagefilehandle%,token$,title$)
```

There is also the user-function:

```
PROCuser_overmenuarrow(RETURN nextsubmenu%,
            parentmenuitem%,x%,y%)
```

which allows us to alter a sub-menu just before it is opened.

At this stage, to demonstrate dynamic menus in our tutorial application, we will only be using:

```
FNwimp_createmenu(menu$,size%)
PROCwimp_putmenuitem(menu%,item%,item$)
```

and we have already used the first of these in Chapter 5.

The programming sequence we will adopt is:

```
On program start, count number of existing
            vehicles on file and extract their
            VRNs into an array.
Create iconbar menu using 'slash separated' list.
If extra vehicles are created during program run,
            use PROCwimp_putmenuitem to add them
            to the menu.
```

To effect this we need, firstly, another visit to `PROCapp_FuelInit` and to take a decision on the maximum number of vehicles we are going to allow. In this tutorial a maximum of 6 vehicles has been chosen, but is easily changed:

```
4820 MaxCars%=6
```

An array to hold the VRNs is then created, using this chosen maximum number of vehicles:

```
4860 DIM CarReg$(MaxCars%)
```

and then the elements of this array are sized immediately after.

The existing iconbar menu creation at Line 620 then needs to be changed:

```
620 IconBarMenu%=FNwimp_createmenu("Fuel/Info/New
     car/Quit",MaxCars%+3)
```

That is, the initially-created menu items are exactly as before but the second parameter is altered to make the maximum allowable menu size sufficient to take the maximum number of vehicles to be allowed plus the three non-varying menu items in the 'slash separated' list (i.e not including the menu title).

With these additions in place we can construct a new app-function to count the cars on file at program start up.

`FNcountAndGetCarRegs` is defined at Line 5210 and called at Line 640 and needs little explanation. Remember from the previous chapter that the vehicle files are named and numbered in the sequence `Car1`, `Car2`, etc. The extracted VRNs are placed in the new array using identical element numbering and the overall number of vehicles is returned to the new global variable `Cars%`. *(The* `UNTIL` *statement at Line 5360 guards against there being more vehicles on file than is set by* `MaxCars%` *- in case you 'play'!)*

A `FOR ... NEXT` loop (at Lines 680-720) then calls the wimp-function `PROCwimp_putmenuitem` the required number of times to add each VRN as a new menu item.

There are three parameters to this wimp-function: namely `menu%`, `item%` and `item$`. The first is self-explanatory. The second and third specify where the new menu item is to be placed in the menu list (1 at top, remember) and the required additional item string itself. Any existing items **at and below** this position will be shuffled down. By calling the VRN array elements in the reverse order (Line 630) the routine places the most-recently-added VRN in the lowest position on the menu - just above 'Quit'.

There is one more consequential thing to do. We need to change `PROCuser_menuselection` to allow for the fact that 'Quit' may now not be item 3 on the menu list.

There are several ways of coping with this, but as ‘Quit’ is always the last item and as Dr Wimp offers the wimp-function **`FNwimp_menusize(menu%)`**, which returns the number of existing menu items on the menu, we can simply use:

```
2700 WHEN FNwimp_menusize(barmenu%)
```

instead of the previous:

```
2700 WHEN 3
```

Adding more vehicles

This takes care of vehicles already on file when the program is started, but what about any new ones created after start-up? These are easily coped with by adding a few lines at the end of our existing app-function **`PROCcreateNewFile`**.

Starting at Line 5900 the newly-created VRN is put into the correct element of **`CarReg$()`** and **`PROCwimp_putmenuitem`** is then called to add the new VRN at the bottom of the VRN list - again, just above ‘Quit’. After this (at Line 5950) **`Cars%`** is incremented by 1

The only thing now left is to ensure that the user does not exceed our newly set maximum number of vehicles on file. It will be more user-friendly to stop the user from using the **`NewCar`** window once the limit has been reached, rather than tell him/her the bad news when **`Create File`** is selected after he/she has added all the details into the the window!

Therefore, the way chosen is to put an error trap at Lines 2800-2830 i.e. before **`PROCwimp_openwindow(NewCar%,1,-1)`** in the **`WHEN 2`** case in **`DEF PROCusermenuselection`**. Further, the **`NewCar`** window is closed deliberately (at Lines 6000-6020) after creation of a new vehicle file if this has taken the total number of vehicles to the maximum.

The result is that when the maximum number of vehicles is on file, the user is forced to use the iconbar menu in order to (try to) get to the **`NewCar`** window. The new error trap will then be duly triggered to give a warning message and prevent that.

Pause to reflect

It is worth pausing at this point to see what has been done. You will now be able to create up to six vehicle records via the `NewCar` window, but no more. As you create them, their VRNs will be added to the iconbar menu. At this stage, nothing happens if you select one of these vehicles from the menu, but the other three permanent menu items can be used to give results as before.

Listing reference: Once again, we have come a fair way in this chapter and it is another useful milestone. So we will save the application at this stage as !Fuel9a.

10. Yet more windows and menus

We now need a means of logging the fuel consumption of any filed vehicle i.e. we need a window which will allow the user to record vehicle mileage readings and fuel quantity bought after each visit to the petrol station.

The window itself is straightforward but we need to look ahead a little before starting to implement it.

We have up to six vehicles on the iconbar menu and, eventually, when we select one of them we will wish to have the choice of either:

- updating its fuel record, or
- displaying the graph of that record.

So, a means of making this choice is needed and a convenient way to do it will be to present these two choices by attaching a simple, identical, two-item sub-menu to each vehicle item on the main iconbar menu. We therefore need to effect the necessary sub-menu construction before going further.

More sub-menus

It would look better if the sub-menu which appears after ‘sliding right’ over the VRN of our choice had this same VRN as its title.

We can achieve this in several ways - possibly the most elegant being to use `PROCuser_overmenuarrow` (see Chapter 17). We could then use a single sub-menu and change its title dynamically.

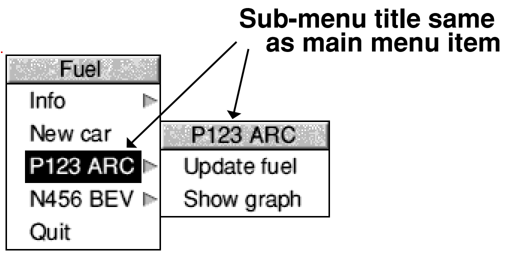
However, our method in this tutorial will be to create a separate sub-menu for each VRN.

We do this simply in five lines.

Firstly, a new array is created at Line 4920, to hold the sub-menu handles. This array is called `IconBarSubMenu%()`. At the start of the program (Lines 700 and 710) - after the VRNs are read in turn from the array `CarReg$()` - a sub-menu is created for each, putting the sub-menu handle into the new array. Each sub-menu is then attached to its corresponding main menu item i.e. to its VRN. If a new car is added after the start of the program, similar action occurs in Lines 5920 and 5930.

Finally, at Line 2920, an `OTHERWISE` statement with nothing in it is added to `PROCuser_menuselection` to ensure that no action takes place if the user presses `<select>` over any of the main iconbar menu items which have a 'slide right' arrow.

If you add these few lines to a copy of `!Fuel9` a you can confirm that all is well so far. The on-screen result is shown here:

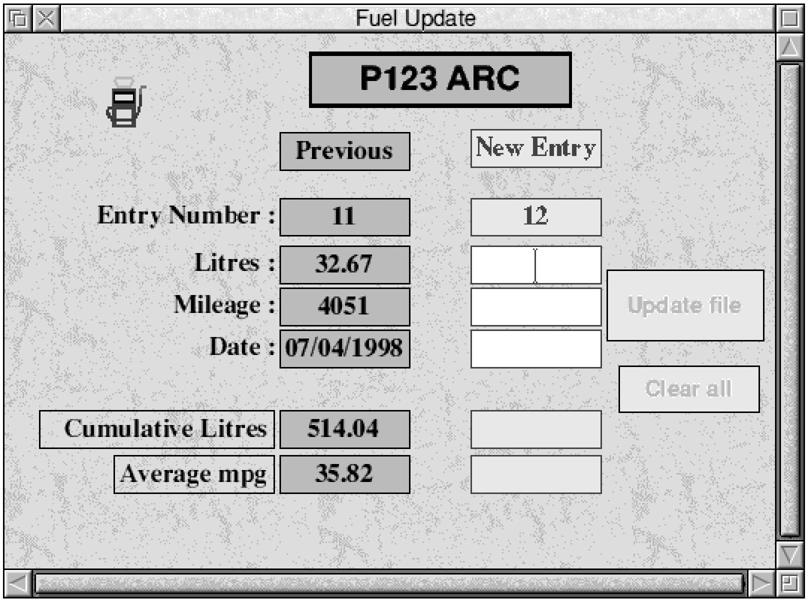


FuelUpdate window

The way is now open to add a window to allow a selected vehicle to have its fuel consumption recorded in its vehicle file. We will call this the `FuelUpdate` window and it will be opened whenever the 'Update fuel' item is chosen on any one of the newly-created iconbar sub-menus, reached by 'sliding right' across a VRN.

We need not introduce the composition of this window in the same detail as was done for the `NewCar` window, because there are many similarities to it. If required, the template should be viewed in an editor to examine the button types, validation strings, etc.

The screenshot below shows this new window as it will typically appear during general use, ready to add the latest fuel consumption update to a vehicle file which has been in existence for some time. *(In this case, the VRN is "P123 ARC" and there have already been 11 fuel entries. The last fuel entry was made at a mileage of 4051 on 7th April 1998 and the average m.p.g. to that date was 35.82)*



	Previous	New Entry
Entry Number :	11	12
Litres :	32.67	
Mileage :	4051	
Date :	07/04/1998	
Cumulative Litres	514.04	
Average mpg	35.82	

Examining this window shows that the user is required to note the car odometer (mileage!) reading and date each time the petrol station is visited - as well has the amount of fuel (in litres) put in.

From a programming viewpoint, we will therefore need routines to:

- open the window;
- show the chosen VRN;
- show its previous record entry (if any);
- validate the user inputs and cycle the caret;
- update the cumulative fuel and average mpg values from the new user-entries;
- update the chosen vehicle file with the entered-and-validated fuel consumption data (when confirmed by the user).

These additions are going to take quite a lot of extra program lines - but they are not going to involve any new Wimp programming feature and only one new Dr Wimp wimp-function is introduced. Therefore, we need only give brief descriptions of the many additions made in this chapter.

Program additions

The first three items on the above list are inter-related. You will recall that the vehicle files are named in numerical order `Car1`, `Car2` etc. We need to know this number in order to open and update the right file. We find it by reading the title from the particular sub-menu which appears and, in turn, use this title to search the array `CarReg$()` - thus giving us the record file numerically.

Therefore, after loading the new window (from template) at Line 540, the new routine in Lines 3010-3530 carries out the first three actions on the above list - leaving a gap to be filled later for the other sub-menu option selection (the graph). As usual, the same result could be achieved in other ways.

Line 3030 is the one which introduces the only new wimp-function in this chapter, `FNwimp_getmenutitle(menu%)`, and its name is fully self-explanatory.

A new app-function `PROCapp_InitialFill(ChosenCar%)` is called in Line 3240 and defined at Line 6090 onwards. Using several calls to `PROCwimp_puticontext`, this app-function fills the `FuelUpdate` window with the previous data from the chosen car file (or preset initial values if the car file does not yet hold a previous fuel entry). A new array `Previous$()` is also created to hold this data, for convenience.

Also introduced (at Line 12500 and Line 12770) are two new general utility app-functions, called `FNapp_TwoDecPlaces(Number?)` and `FNapp_DecimalDate(Date?)`. The first is used frequently in the program to format user-input and other numbers uniformly to two decimal places. It rounds correctly and also deals consistently with zero and whole numbers. Note that it takes a string and returns one.

The second converts a date - which must be a string in the `dd/mm/yyyy` format - into a decimal number, with the years as the integer part. This is used to test whether one date is later than another.

(You may find these two app-functions useful in other programs.)

With modifications, the already-existing caret cycling and user-entry validation app-functions from Chapter 7 can be used for the “Fuel Update” window - provided the three user-entry icons have consecutive icon numbering. The initial coding anticipated this.

One of the user-entry types - fuel quantity in litres - is a new one, so we need an extra validation routine to cover that, as well as a new routine to update the average fuel consumption. These are:

```
PROCapp_UpdateCumLitres (at Line 8910)
PROCapp_UpdateAvMpg (at Line 9040)
```

Further, there are a couple of additional validation steps needed for mileage and date entries, but only in the “Fuel Update” case - to ensure that the mileage is greater than last time and that the date is not earlier than last time.

There are several other consequential, but simple, additions to:

```
PROCuser_mouseclick
PROCuser_openwindow
PROCuser_closewindow
PROCuser_keypress
PROCapp_FuelInit
FNapp_DoubleCheck
PROCapp_ClearUserIcons
PROCapp_MoveCaret
PROCapp_CheckAllEntries.
```

Fuel file record

Once a valid set of new entries has been made, the user is given the option to clear them or update the file with them. `PROCappUpdateFile(Car%)` is the new app-function to do this - starting at Line 6670.

As before, it is useful to store the three validated entries (plus, this time, the two consequential entries for cumulative litres and average m.p.g.) in the array `Valid?()` to assist in checking that the user has not changed the entries after all three are validated.

A simple file structure has been chosen. A single fuel update record comprises the contents of the five icons resulting from the new entry i.e. litres, mileage, date, cumulative litres and average miles-per-gallon. Clearly, other possibilities exist.

Records are added to the vehicle file in sequence after the file header space. It is worth noting that it makes things a little easier if each update record takes up the same amount of space on the file, which is also picked up below. To make the program work for either Basic V or Basic VI a standard record length of 48 bytes has been set in `PROCapp_FuelInit`. (44 bytes would be needed in Basic VI.)

Once the file has been updated, the `FuelUpdate` window is prepared for further updates by transferring the just-made entries to their ‘previous’ counterparts, incrementing the record numbers and clearing the user-entry icons and corresponding arrays.

Pause to reflect

The first main objective of our tutorial exercise has now been reached and further additions concern only the display and printing of graphs of the stored fuel records.

> **Listing reference:** This is obviously an important milestone, so we will save the application at this stage as `!Fuel10a`.

Numbers or strings?

Despite the fact that all displayed entries in our icons are strings, the same items in the new files are all numbers (except for the date).

There is a dilemma here: we need numbers to do the calculations but strings for icon display. So, frequent conversion between numbers and strings is inevitable.

All other things being equal, it seems preferable to file numbers as numbers - simply because they then occupy a known number of bytes, unlike their string conversions.

When converting from a string to a number and then later back to a string, be very careful to truncate the final string every time to avoid the annoying effect of internal ‘rounding errors’ with real numbers, (e.g. 42.00 appearing as 41.9999999...) You will note in the listing that `FNappTwoDecPlaces` is often used to do this.

11. Wimp graphics (The principles)

Before taking our tutorial application further, it will help to have a look at some points about graphics in general in Wimp programs.

Fundamentals

There is more than one way of putting graphics onto the desktop screen and each has its pros and cons. The differences concern the actual screen display method rather than the generation of the graphic itself. For example if we want to show a graph on a Wimp screen we can:

- draw it directly; or
- construct it as a sprite, then plot the sprite to the screen; or
- construct it as a drawfile, then plot the drawfile to the screen.

In each of these methods, the actual graphic construction will be the same and use the familiar Basic `MOVE`, `DRAW`, `PLOT` etc. keywords.

By and large, the first method - direct drawing - is the one most likely to be used and this is what our tutorial application will do.

Coordinates

There is no great difficulty in designing graphics for direct display into in a desktop window once it is appreciated that, effectively, there are two overlapping coordinate systems in play simultaneously.

Firstly, there is the **screen** graphics coordinate system (which you may well know from non-Wimp programs). This, by default, uses the bottom left corner of the screen as its graphics origin. It is important to realise that the final instructions given to the computer to produce something on the screen are going to be in these screen coordinates.

But a few moments thought will show that this screen reference is not going to be very convenient for the programmer to specify points in a Wimp window which, at any point in time, may (due to scrolling, sizing, etc.) show only part of its defined total window work area - and, as if that wasn't enough - can be moved all over the screen by dragging.

Clearly, for window design and programming we need to be able to specify points with reference to the window itself - and the only sensible reference is one of the corners of the total defined **work area** of the window. The corner chosen - you may be surprised to learn - is the **top** left corner of the work area. (Yes - this means that all y-axis values are going to be negative if they are to fall within the work area.)

This choice is not so odd when we reflect that (in English, anyway) we normally regard the top left corner of a page as the starting point.

The diagram on the next page shows the relationships.

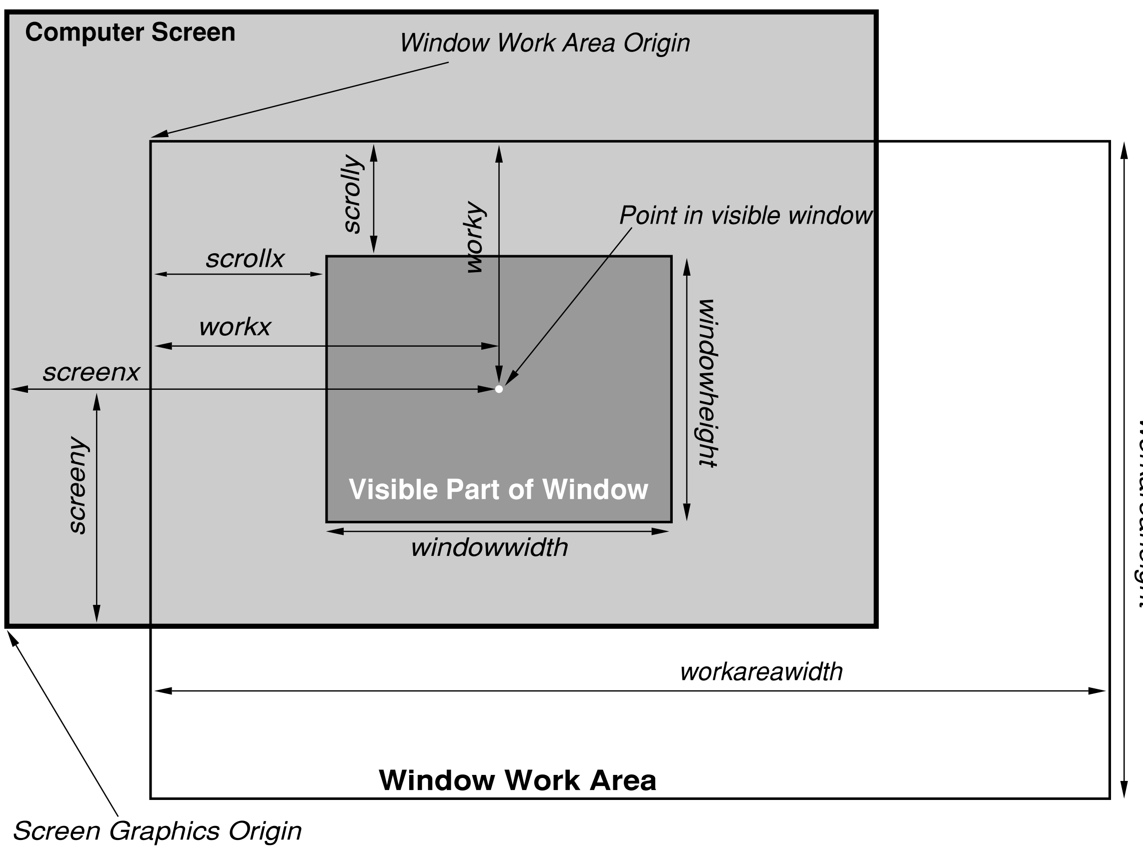
Thus, any point can be defined **with respect to its window work area position** by specifying its `workx` and `worky` values - **and these values will not change**, whatever we do to the window.

The same point can also be defined **with respect to its position on screen** by `screenx` and `screeny` - but **these values will change frequently**, whenever we move, scroll or re-size the window, in fact.

So, for convenience, we normally do all our window design using 'work units' and all the actual plotting to screen is done in 'screen units'.

How (and when) do we convert between the two? Well, the answer is that - by and large - we don't. We let the Wimp do it for us, and it is specifically designed to do this via `SYS` calls - particularly one which provides full information on the current state of any window.

As usual, Dr Wimp makes it even simpler, with a series of wimp- and user-functions optimised for plotting text, graphics, sprites and drawfiles onto the screen or window - with colour and font control where applicable. Also provided are wimp-functions to do any coordinate conversions which may become necessary.


Computer Screen
Window Work Area Origin
scrolly
worky
Point in visible window
scrollx
workx
screenx
screeny
windowheight
Visible Part of Window
windowwidth
workareaheight
workareawidth
Window Work Area
Screen Graphics Origin

The Redraw process

So far in our tutorial application, all our text and sprite ‘plotting’ has been into icons. As we have seen, this is extremely simple and the Wimp automatically takes care of all the display management necessary when we move and/or scroll any window containing icons - and also when icons are revealed again after being covered by another window.

However, when it comes to displaying graphics designed by the user/ programmer - or putting text directly into a window - the Wimp cannot take all the load. In these cases, the Wimp needs help when that window is initially opened and subsequently whenever it is moved/scrolled/resized etc.

So, taking as an example our wish to display a graph, we have to **redraw** that graph (or perhaps only part of it) often - and the same ‘redrawing’ action is also used to display it initially.

This explains why the Wimp calls this the redraw process. This process applies to whatever method we use actually to plot the graphic to the screen i.e. whether sprite, drawfile or directly.

The Wimp activates the redraw process using Reason Code 1, the Redraw request - and this is linked with the SYS calls:

```
SYS "Wimp_RedrawWindow"
SYS "Wimp_GetRectangle"
```

These `SYS` calls are used in a special way to ask the application program to redraw the graph (or whatever) for only those parts of the window necessary to produce the new screen. (This may be the whole of the window but might be just a part of it that was previously - or is now - hidden by another window, or perhaps the same portion as before but now in a different place on the screen.)

We do not have to worry about any icons in such windows; they are not regarded as part of the redraw process.

Whenever a redraw is requested by the Wimp it breaks the required new area up into rectangles - maybe only one - and calls for each of these rectangles to be redrawn in turn, until all have been completed. (On initial opening of a ‘redraw window’ Reason Code 2 causes the non-redraw parts of the window to be displayed and then Reason Code 1 is automatically generated to call for the redraw parts.)

Programming this activity ‘from scratch’ is not instinctive and it takes time to understand the process properly. However, once the penny drops it is different rather than difficult. The important thing to note is that all our custom graphics/direct text programming can take place within the response to Reason Code 1 - the Wimp takes care of the rest.

It is worth noting in this process that although, generally, the Wimp asks only for a smaller rectangular part to be redrawn each time, there is nothing to stop you responding by redrawing the whole graphic window each time - even though some of it may be unnecessary. Obviously, this approach has a speed implication for a very large and/or complex graphic, but there might not be a significant time penalty for a simple one.

Finally, one very important advantage of the redraw process is that once you have got it right, you can usually use exactly the same coding for producing hard copies of the (wysiwyg) graphics from a printer - but more on that in a later chapter.

Dr Wimp’s approach to graphics

Dr Wimp removes nearly all the pain - mainly due to the user-function:

```
PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
    printing%,page%)
```

Every time the Wimp issues Reason Code 1, the **`DrWimp`** library calls this user-function at least once, passing to the program each time live values for the long list of parameters shown. **`DrWimp`** will then recall this user-function for each rectangle requested by the Wimp.

As usual, the programmer’s job is to fill the initially-empty **`DEF PROC`** using these parameters as and if required.

At the moment in our tutorial we are only interested in the **`minx%`**, **`miny%`**, **`maxx%`** and **`maxy%`** values. These are simply the corners (in screen OS units) of the rectangle of `window%` that the Wimp wants redrawn. As we shall see in the next chapter, all the complication of the redraw process and its `SYS` calls is completely hidden behind the Dr Wimp scenes. *(We still have to grapple with the two coordinate systems though!)*

12. Wimp graphics (The practice)

With the general introduction provided by Chapter 11, we can now start things rolling to produce a graph of a vehicle's fuel consumption history.

This is fairly straightforward, but because of our later intention to be able to print the graph we need to do some preparations before we actually get to the graph drawing itself. The sequence we will use is:

-Design and load a new window for the graph;

-Re-size this window to reflect the paper size in the currently loaded printer driver;

-Draw the graph.

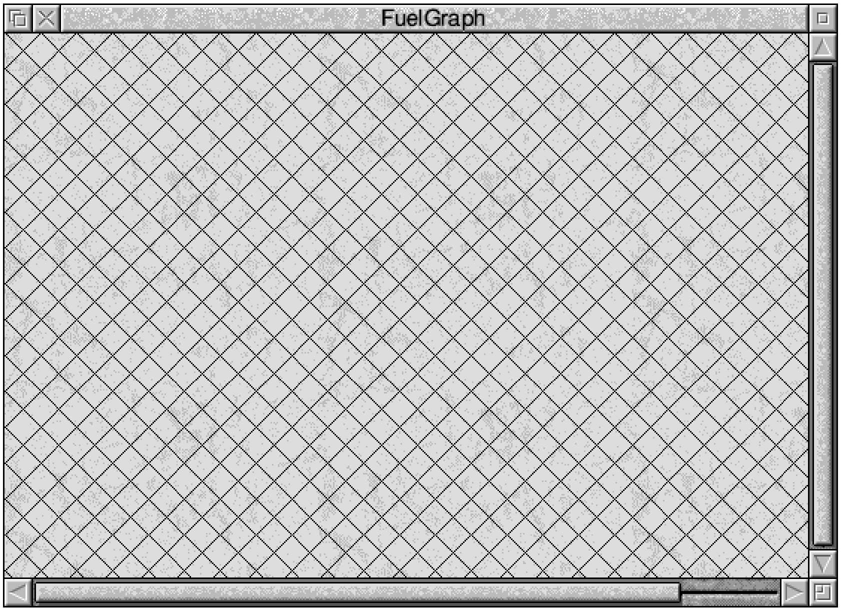
Redraw windows

Our new window for the graph will be called 'FuelGraph' and, initially, it is simply an empty window of any convenient size to show on the screen.

However, this blank window needs to have one very important characteristic built into its definition: the 'auto redraw' option must be disabled (in the template editor - see Appendix 7). This tells the Wimp that Reason Code 1 is going to be needed i.e. the program needs to assist in displaying the window contents - as described in the preceding chapter.

When you disable 'auto redraw' it is likely that, in the template editor, the window will be displayed with its background cross-hatched - as below.

(The cross-hatching will not appear in the program use of the window.)



We don't happen to want to add any icons to this window definition, but there is no reason why they couldn't be added as usual. The Wimp would then manage the screen display/updating for the icons in the usual way - the need for a redraw process applies only to putting our custom graphics directly onto the window background. *(Note that any icons would be displayed 'on top of' the redraw graphics.)*

Our first step is to load the new window - at Line 550. Having done this, we need to add to **`PROCuser_menuselection`** so that the new window is initially opened in response to selecting **`Show graph`** on the sub-menu from any of the VRN items on the main iconbar menu

On an interim basis, just add the following to allow you to check that the blank new window is opening correctly:

```
3300 WHEN 2
3310 PROCwimp_openwindow(FuelGraph%,1,-1)
```

In principle, we could now proceed to draw the graph in the displayed window, merely by adding something to PROCuser_redraw. To confirm this, add the following temporary line:

```
950 PROCwimp_plotwindowrectangle(window%,50,-300,
            300,200,1)
```

and you will get a black square appearing in the new window if you select ‘Show graph’ from the iconbar submenu.

Regard this as only a confidence building step and delete this temporary line after you have tried it.

Currently, the size of our new window, as created in the template, is arbitrary and may not be a good size for showing our graph. With our eye on the future need to produce hard copy, it would be nice to try to produce a ‘what-you-see-is-what-you-get’ display i.e. the displayed graph window ought to look like the sheet of paper on which we are later going to print the graph.

So, we need to resize the graph window and Dr Wimp has several helpful wimp-functions for this.

(We cannot sensibly do this in the template editor simply because the paper sizes and print margins will vary considerably among different printers - and the printer driver may well be changed whilst the program is running).

Re-sizing windows

Firstly, let’s have a look at the range of wimp-functions available here. They are:

`FNwimp_getwindowvisiblesize`
: gives currently displayed width/height

`FNwimp_getwindowworksize`
: gives defined work area width/height

`FNwimp_getwindowvisiblework`
: gives work area coords of currently visible sides i.e. takes current scroll values into account

`FNwimp_getwindowvisiblescreen`
: gives screen coords of currently visible sides i.e. gives window position

`FNwimp_getscroll`
: gives scroll values

```
PROCwimp_resizewindow
PROCwimp_resizewindowvisible
PROCwimpscroll
PROCwimp_scrollto
PROCwimp_openwindowat
```

As you can divine, the first five read information about the existing size and the last five allow us to alter some aspect of the window size or position on screen. With them we can produce a window of any size we want and place it precisely where we want, with whatever scroll values we want.

Note that all these functions - except `PROCwimp_openwindowat` of course - can be used whether the window is open or closed. If closed, the first five return values from the window as it was when last open (or as loaded/created if it has not yet been opened).

Back to the tutorial …

For our purposes, we are going to resize the graph window work area to represent the printer paper: then resize its visible area so that none of the window is off-screen when we open it initially.

Paper size and printer driver

This leads directly to the question “How do we know the paper size?” Well, that’s the easy bit. If you have a printer driver loaded it incorporates a paper definition - and there is a wimp-function to read this. It is:

```
FNwimp_getpapersize
```

which allows you to read the paper dimensions and the print margins.

“But what if a printer driver isn’t loaded?” I hear you ask! Once again, there is a wimp-function to help us! This time it is:

```
FNwimp_pdriverpresent
```

and also:

```
FNwimp_getpdrivername
```

to find the driver name if you want it.

Our sequence is therefore to call `FNwimp_pdriverpresent` to see if a printer driver is loaded.

If one is, then we call `FNwimp_getpapersize` to find the needed measurements. If one isn't, then we supply some default paper sizes. Either way, we end up with a paper size.

Further, if the printer driver is changed at any time while our application is running - or a driver is loaded for the first time while it is running - then the `DrWimp` library once again comes to our rescue by calling automatically yet another user-function:

```
    PROCuser_printerchange
```

As usual, the `DEF PROC` of this user-function is empty by default, so we can utilise it simply by putting some coding of our choice into it. In this case we will repeat our check of the printer driver status and change the paper size accordingly.

As you can see, Dr Wimp really makes printer driver management very easy.

By the way, if you load a printer driver by running `!Printers` and then subsequently (in the same session) quit `!Printers`, the printer driver still stays installed. This is sometimes the source of a little confusion until you know.

(You may wish to note, in passing, that a change of printer driver status is one of the events which is notified to applications by the Wimp using the Wimp Messaging system and Reason Codes 17, 18 and 19. See Appendix 10.)

Program changes

A little earlier, a temporary addition to `PROCuser_menuselection` was made. We can now change that, initially, to:

```
3300 WHEN 2
3330 ChosenCarFile$=SourceDir$+".Car"+
          STR$(ChosenCar%) :REM** Needed in
          'PROCuser-redraw' also. **
3360 CarFile%=OPENIN(ChosenCarFile$)
3370 Records%=(EXT#CarFile%-FileHeader%)/
          RecordLength%
3380 CLOSE#CarFile%
3460 PROCapp_SizeAndDisplayWindow
```

This sequence constructs - in `ChosenCarFile$` - the exact file-path for the chosen vehicle - using the value of `ChosenCar%`, which was found a little earlier in `PROCuser_menuselection`.

`ChosenCarFile$` will be needed a little later to draw the correct graph.

Although not essential, `ChosenCarFile$` is also now defined in `PROCapp_FuelInit` by adding:

```
4940 ChosenCarFile$="":REM** Global variable. **
```

The window resizing

Line 3460 then calls a new app-function which is defined at Line 13620 and within which our re-sizing etc. takes place.

It is worth having a look at this in more detail.

```
13620 DEF PROCapp_SizeAndDisplayWindow
13710 PROCapp_GetPaperSizes
13730 WindowWidth%=PaperWidth%*ScreenScale
13740 WindowHeight%=PaperLength%*ScreenScale
13760 PROCwimp_resizewindow(FuelGraph%,
             WindowWidth%,WindowHeight%)
```

```
13800 MaxScreenWidth%=FNwimp_getscreensize(0)
13810 MaxScreenHeight%=FNwimp_getscreen size(1)
13830 PROCapp_ResizeVisibleAndDisplay
            (WindowWidth%,WindowHeight%,
            MaxScreenWidth%,MaxScreenHeight%)
13850 ENDPROC
```

Firstly, as just indicated above, we get the paper sizes and print margins. This is done by the new app-function **PROCapp_GetPaperSizes** - defined at Line 13890.

```
13890 DEF PROCapp_GetPaperSizes
13930 Pdriver%=FNwimp_pdriverpresent
13950 IF Pdriver%=0 THEN
13980    PaperWidth%=1488
13990    PaperLength%=2104
14010    LeftPrintX%=180 :REM** Deliberately set
              to 180 OS (1 inch), for effect. **
14020    BottomPrintY%=98
14030    RightPrintX%=1466
14040    TopPrintY%=2082
14090 ELSE
14100    PaperWidth%=FNwimp_getpapersize(0,0)
14110    PaperLength%=FNwimp_getpapersize(1,0)
14130    LeftPrintX%=FNwimp_getpaper size(0,1)
14140    BottomPrintY%=FNwimp_getpapersize(1,1)
14150    RightPrintX%=FNwimp_getpapersize(0,3)
14160    TopPrintY%=FNwimp_getpapersize(1,3)
14210 ENDIF
14230 ENDPROC
```

This app-function first checks to see if a printer driver is loaded. If there is, it calls **FNwimp_getpapersize** several times (with different parameter values) to extract the printer driver paper width, height and print margins. If no driver is present then sensible default values (here, typical of an A4 page) are used. Note (Line 14010) that the left-hand print margin is deliberately made fairly wide to emphasize this situation visually (purely for tutorial purposes) and to make the effect of any subsequent printer driver loading obvious.

At Line 13760 of **PROCapp_SizeAndDisplayWindow**, the work area of the graph window is then resized to the paper width/height, scaled by a new global variable **ScreenScale** (defined at Line 5100 in

`PROCapp_FuelInit`) with a value of 0.75. This value is chosen solely as an attempt to make the displayed graph window approximate to the actual printing paper size. The value will be about right for a 17-inch monitor, but you will need to alter it for other sizes - smaller for bigger screens, or larger for smaller screens.

The maximum screen width and height in the current screen mode are then read using a new wimp-function:

```
FNwimp_getscreensize
```

This is called twice - once to get the width and once to get the height, merely by changing the parameter value. As we have already seen with **`FNwimp_getpapersize`**, *this is a typical Dr Wimp method for extracting values which are actually provided by a* **`SYS`** *call, behind the scenes.*

Another new app-function, **`PROCapp_ResizeVisibleAndDisplay`**, then does any necessary resizing of the **visible** window area (to keep it within the screen) and then causes the result to be displayed.

Within **`DEF PROCapp_ResizeVisibleAndDisplay`** (at Line 14270) a decision is taken about the 'standard' displayed window width. Here, the A4 paper width of 210mm has been chosen - and is converted into OS units at Line 14330 using yet another useful wimp-function:

```
FNwimp_lengthtoOS
```

which allows a physical length (as printed on paper) to be converted to OS units on the screen - either from inches or millimetres - and it can be scaled up or down. It has, of course, a complement:

```
FNwimp_OStolength
```

Armed with this data we can then check whether or not the resized window width is greater than the standard 210mm. If it is, we reduce the visible window width to the standard: otherwise we make the visible width the same as the resized width.

Something similar needs to be done about the visible window height - and here we are pretty sure the resized window height will well-exceed the display maximum. The arbitrary choice here - at Line 14420 - is to restrict the initially-displayed resized window height to no more than three-quarters of the screen height.

Thus, at Line 14520 we use:

```
PROCwimp_resizewindowvisible
```

with the above-calculated visible width and height values.

When it comes to actually displaying the window, we opt simply to redraw it if it is already open - to preserve its current screen position - using the new wimp-function **`PROCwimp_redrawwindow`**.

However, if the window needs to be opened then the resized visible window is centred on the screen, at Line 14620, by:

```
PROCwimp_openwindowat(window%,x%,y%,stack%)
```

This is yet another new, but easy, wimp-function which allows us to open any window with its top left corner at a specified place on the screen i.e. we have to supply screen coordinates (in OS units) for **`x%`** and **`y%`**.

Drawing the graph

At long last (you may feel) we can now concentrate on filling `DEF PROCuser_redraw` in order to draw the graph.

It will be sensible to carry out the detailed work in a new app-function (`PROCapp_DrawGraph` at Line 14710) so we can simply start with the following additions to `PROCuser_redraw`:

```
940 DEF PROCuser_redraw(window%, minx%, miny%,
            maxx%, maxy%,printing%, page%)
970 CASE window% OF
980      WHEN FuelGraph%
990      PROCapp_DrawGraph(ChosenCarFile$,
            window%,minx%,miny%,maxx%,maxy%,
            printing%,page%)
1010 ENDCASE
1030 ENDPROC
```

Thus, we just pass all the parameters through to our new app-function, along with the path of the file of our chosen vehicle.

The four x/y parameters are the bottom left and top right corners of the window rectangle which needs to be drawn/redrawn - as described in the ‘redraw’ process of the previous chapter. They are given in OS units and they are ‘screen units’ i.e. relative to bottom left corner of the screen.

However, although it makes sense to pass all the parameters through - in case we later want to change things - our graph is not complicated and we are therefore going to take the easier alternative of redrawing the whole graph each time a redraw is called for - so we will not actually need to use the x/y parameters.

(The `!Grid` *application which comes with the Dr Wimp package (in its* `Examples` *directory) gives good information on how to use the redraw rectangles in ‘busier’ graphics.)*

Before proceeding, remember that our window design work will need to be in ‘work area coordinates’ but our actual plotting will need to be in ‘screen coordinates’. In the following program changes, the variables are all named with ‘screen’ or ‘work’ in them to show which coordinates are being used.

Showing the paper

Our first graphic step is essentially cosmetic, but also helpful - that is, to display the print margins in the graph window. These margins were obtained earlier (from the printer driver or default values) when resizing the graph window to the paper size. A pleasing visual effect is produced if we display the printing area as a white sheet with the surrounding margins showing as the normal desktop window grey.

The following lines of `DEF PROCappDrawGraph` do this - taking the work area paper coordinates and converting each to screen coordinates using the wimp-function `FNwimp_worktoscreen`, scaled by the previously set global variable `ScreenScale`. A simple `RECTANGLE` call then plots the 'paper' in white onto the normal window grey background:

```
14860 REM** Convert the printing margins to
            rectangle corners in screen coords.
             **
14870 PrintAreaScreenMinX% =
            FNwimp_worktoscreen(FuelGraph% ,
            LeftPrintX%*ScreenScale, 0)
14880 PrintAreaScreenMinY% =
            FNwimp_worktoscreen(FuelGraph% ,
             (-PaperLength%+BottomPrintY%)*
            ScreenScale,1)
14890 PrintAreaScreenMaxX% =
            FNwimp_worktoscreen(FuelGraph% ,
            RightPrintX%*ScreenScale,0)
14900 PrintAreaScreenMaxY% =
            FNwimp_worktoscreen(FuelGraph% ,
             (-PaperLength%+TopPrintY%)*
            ScreenScale,1)
14940 REM** Display the 'paper' rectangle in
            white on the screen, before any
            drawing. **
14950 PROCwimp_setforegroundcolour(255 ,
            255,255):REM**White **
14960 RECTANGLE FILL PrintAreaScreenMinX% ,
            PrintAreaScreenMinY% ,
            (PrintAreaScreenMaxX%-
            PrintAreaScreenMinX%),
            (PrintAreaScreenMaxY%-
            PrintAreaScreenMinY%)
```

```
    14970 PROCwimp_setforegroundcolour(0,0,0)
                   :REM** Reset to Black **
```

If you make just these changes and run the program you will see the ‘paper’ duly plotted when you choose ‘Show graph’ from the iconbar sub-menu.

Note also the two calls to `PROCwimp_setforegroundcolour`, which changes/sets the plotting colour using ‘rgb’ values from 0-255. Here, it is first set to White and, after use, it is reset to Black. It is always a good policy to reset a colour back to the default (Black) after use - particularly if White is being used.

The graph axes

We can now do a similar thing for the graph axes and text labelling. First decide where to draw the graph in the (resized) graph window. You may find a rough sketch helpful here but, arbitrarily, the axes have been set 25mm (1 inch) in from the left/bottom/right **of the printed paper borders** (Line 15020) and 50mm in at the top (Line 15100). Other arbitrary values have been set for the title text.

To make the coordinate conversions very clear, it is done in two steps. Lines 15070-15130 calculate the work coordinates then Lines 15180-15240 do the screen coordinates conversions.

Then Line 15550 simply passes all the values to `PROCapp_PlotAxes` for the actual axes plotting - followed by `PROCapp_PlotGraph` for the graph itself. These two app-functions can thus ignore all the work/screen conversion issues.

Apart from introducing some new wimp-functions for text plotting, there is nothing unusual in `PROCapp_PlotAxes`. The chosen car file is accessed to find the range of fuel consumption and the number of records already logged for that vehicle. From these, the graph scales are calculated. *(The graph is always the same physical size, with the scales being adjusted as necessary for each vehicle.)*

Text plotting

For demonstration purposes, `PROCapp_PlotAxes` deliberately uses more than one method for adding text to the graph axes. Firstly, using ordinary `VDU5` statements, producing System Font. Then using two of several wimp-functions available for text - the first being:

```
     PROCwimp_plottext
```

which takes several parameters to control the font choice, size, colour and position. The second is:

```
     PROCwimp_plottexth
```

which is almost the same but uses a ‘font handle’ for the specific font type and size instead. The handle is obtained from using `FNwimp_getfont` at Line 5150 in `PROCappFuelInit`.

If you compare the first few parameters of `PROCwimp_plottext` and `PROCwimp_plottexth` you will easily see what a font handle is: it simply represents a specific font type and size.

Font handles are the main way of defining outline fonts in wimp programs. One advantage of them is that once a font handle has been set up (simply by using `FNwimp_getfont` as above, in Dr Wimp) the handle can be used as many times as you like within the application programming.

More detail on text plotting and fonts is given later in Chapter 20.

Plotting the graph

After all this, the actual graph plot is somewhat of an anti-climax. It is merely a matter of reading each record in turn from the chosen car file and drawing a short line between each average miles-per-gallon value successively, with a `DRAW` statement in a control loop.

A `MOVE` statement is used for the very first record to set the starting point of the graph plot.

Three horizontal dotted lines are then drawn across the graph, l mpg apart and centred roughly at the final average position, to provide a convenient variation gauge.

As you can see, there are no wimp-related statements in `PROCapp_PlotGraph` i.e. only the usual Basic graphics plotting commands are needed.

Tidying up

There are a few housekeeping jobs to do before we can leave this section.

Firstly, what happens if we try to plot the graph of a filed vehicle which does not yet have a fuel history? At the moment an error will occur as the graph plotting procedure tries to read non-existent records.

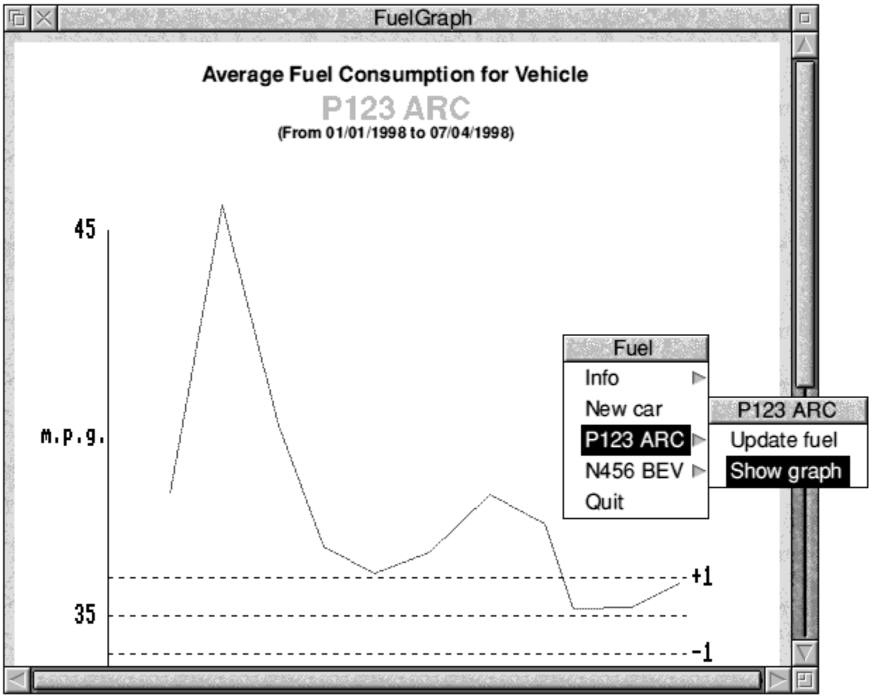
We need to trap this and the best way to do it is to stop the program entering the graph-plotting routines if a file does not have sufficient records to generate a sensible graph - say, less than 2 records.

The routine at Lines 3400-3470 does the necessary. (Note that if this circumstance occurs when the graph window is already open, then the window is closed after the warning is given.)

The second issue concerns the earlier point about the printer driver and what happens if we change (or load/delete) the driver whilst the program is running.

All we need to do - at Line 4280, inside `PROCuser_printerchange` - is call `PROCapp_SizeAndDisplayWindow` again, but only if the graph window is already open.

When run, the screenshot below shows the result of selecting the ‘Show graph’ option from the iconbar sub-menu.



Pause for reflection

We have done quite a bit in this chapter. Although none of it has been difficult, the preparation and organising the window, paper/ printer driver information, plus the work and screen coordinates took much more time than the actual graph drawing itself. Had we not had the help of Dr Wimp in these tasks there would have been a great deal of tedious nitty-gritty - and maybe much hair-tearing!

The key point is to plan the sequence carefully and then take it a step at a time. By and large, by this stage in the application, it is possible to check that each step is working OK before going on to the next.

Listing reference: This is obviously an important milestone, so we will save the application at this stage as `!Fuel12a`.

It is worth mentioning here that - although very suitable for introducing newcomers to wimp graphics - the main disadvantage of plotting a graph directly to the screen and then redrawing the whole graph and its axes (even when only a partial redraw is needed) is that any screen updating (e.g. during window dragging) can take a noticeable time if more than a few graph points are involved.

Further, in our particular tutorial, the emphasis has been on clarity in the listings, and this means that some steps and/or calculations have been included within the umbrella of **`PROCapp_DrawGraph`** when - for speed - they might be better done outside.

13. Creating windows/icons from within a program

Although our next major task in the tutorial application is printing, we will first need to add something to allow the user to trigger the printing process. A convenient way to do this is with a `'Press to print'` button on the graph display and this provides a good excuse to introduce Dr Wimp's facilities for creating windows and/or icons 'on the fly' i.e. from within the `!RunImage` program rather than being loaded after design in a template editor.

You may, at first, wonder why anyone would want to bother about creating windows/icons other than via a template editor - particularly as we took some trouble in Chapter 2 to stress both the great advantage of templates and how easy Dr Wimp makes it to use them.

The reason is that there are some circumstances where - using Dr Wimp or not - a window template cannot be used (or not wholly) and there is also the fact that sometimes you may not want everyone to have easy access to a window/icon design.

For example, the number of icons needed might vary e.g. to report the results of a search. Or the position of the icons in a window might need to change if the window is resized during the program run.

Consequently, Dr Wimp offers a pair of wimp-functions and the utility application `!CodeTemps` to help us.

The two wimp-functions take many parameters - which are similar:

```
FNwimp_createwindow(vminx%,vminy%,vmaxx%,vmaxy%,
               wminx%,wminy%,wmaxx%,wmaxy%,flags%,
               colourflags%,button%,title$,
               titleflags%,maxind%,sarea%)

FNwimp_createicon(window%,wminx%,wminy%,wmaxx%,
               wmaxy%,flags%,esg%,button%,fcol%,
               bcol%,fhan%,text$,sprite$,sarea%,
               maxind%,valid$)
```

The Dr Wimp documentation gives a full explanation of these parameters, but you can see that, essentially, they line up with many of the entries needed in the parameter block for their corresponding `SYS` calls (see Chapter 3 and its `!TestAppl`).

For windows, the parameters determine the work area and visible size and position on screen, the value of `flags%` and `colourflags%` determine whether or not scroll bars, back icon etc. are present and their colour.

The parameter `button%` decides how mouse-clicks made over the window background are to be interpreted (the ‘button type’) - and closely follows the same parameter in `FNwimp_createicon` below.

`title$`, `titleflags%` and `maxind%` determine the titlebar text, characteristics and the allowed maximum length of the title text (if set to indirected).

`sarea%` is the handle of a sprite area holding any sprites used in the window definition - 0 meaning the Wimp sprite pool, the most usual value.

For icons, the parameters determine the window to which they will be added, the size and position in the window, the ‘icon flags’ value, the ESG value (see Appendix 7 !TemplEd), the button type, colours, text font, text string, validation string, etc.

The value of button% (for both windows and icons) lines up exactly with the Wimp’s ‘button types’.

Once the parameters are determined, the use of a call will create the window and/or icon and return handles accordingly.

For a window, the end result is the same as loading a template i.e. the handle can then be used to open the window etc.

For icons, display will occur automatically at the next opening/redraw of the window (or redraw of that part of the window where the icon is to be placed) - and you can force this by using `PROCwimp_redrawwindow` or `PROCwimp_updatewindow` after icon creation.

!CodeTemps utility

This is all well and good, but it doesn't remove the need to have to decide the detailed parameter values for window and/or icon creation.

Once again, Dr Wimp allows you to have the best of both worlds - that is, to use a template editor to design your window/icons and then to convert the result automatically into a set of parameters for the above wimp-functions. This is the role of the utility `!CodeTemps`, which comes with the package.

This is what happens. You design your windows and/or icons in a template editor as usual and save the resulting window template in a normal template file.

If you only want icons e.g. to create in an existing window, you will still have to design them in a dummy window and save the window template. Some simple editing will then be needed, as we will see a little later below when a specific example is described for the tutorial application.

`!CodeTemps` is then loaded onto the iconbar and the template file is dragged to it. This produces a small window with an icon for each window held in the dragged file. (The window will probably look very much like the template file window from your template editor.)

If you then click on one of these icons a Save box will appear showing a Basic file which, as usual, you need to drag to a convenient directory window. If you then load the resulting Basic file into a text editor you will see a set of Basic statements comprising all the calls to `FNwimp_createwindow` and `FNwimp_createicon` necessary to produce your designed window/ icons.

It is as simple as that.

Back to the tutorial …

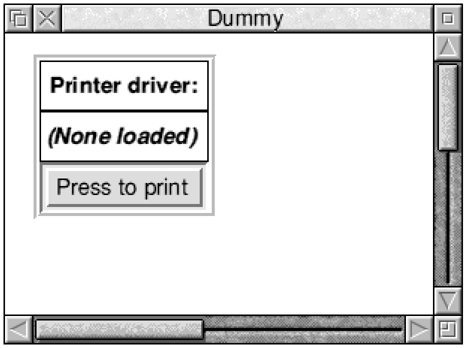
As was said at the start of the chapter, we need to give the user a means of triggering the printing and a `'Press to print'` button will fit the bill.

In fact, we are going to add something just a little more fancy: there will be a small group of icons showing the name of the loaded printer driver as well as a `'Press to print'` button. Further, we will only enable this latter button if a printer driver is actually loaded.

The icons were designed in `!TemplEd` in a small dummy window called Dummy, saved in a templates file called `Create` - and you will find this template file in the folder `CreateIcon` in the Progs directory on the disc which comes with this book.

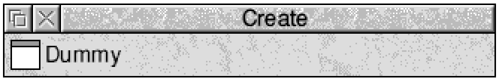
As long as they are grouped together in the way they are to be used, it doesn't matter too much where the icons are located in the `Dummy` window. The only points to watch are:

- the text in the middle icon - the name of the printer driver -needs to be set as 'indirected' and its maximum size should be set with the longest string likely to be used, as shown in the screenshot below.
- the 'Press to print' icon is set to the 'Menu' button type and its ESG value is set to 1.



(In the above figure, the window has been given a white background and the top two icons have been given borders and filled white - solely to show their construction more clearly. In the the `Create` *template, these icons are as they are to appear in the tutorial i.e. they have no border and they are filled with a different colour.)*

If you load `!CodeTemps` and drag the `Create` templates file to it, the following window will appear:



Click on the `Dummy` icon and a Basic file Save box will appear - with the default name `Dummy`. Drag this file to a place of your choosing. *(We need it only temporarily and a copy is also in the Create folder mentioned above.)*

Load this file into a Basic editor to see:

```
10 Dummy%=FNwimp_createwindow(546,772,1118,1148,
        0,936,1236,0,%0000000101111111l,
        &727031C,0,"Dummy",%00001101,6,0)
20 itext$=""
30 icon0%=FNwimp_createicon(Dummy%,38,-276,292,
        28,%0000000100111101,0,0,7,1,0,
        itext$,"",0,1,"R4")
```

```
40 HomertonBold12%=FNwimp_getfont("Homerton.
         Bold",12)
50 itext$="Printer driver:"
60 icon1%=FNwimp_createicon(Dummy%,54,-114,276,
         42,%000000101111001,0,0,0,0,
         HomertonBold12%,itext$,"",0,16,"F17")
70 HomertonBoldOblique12%=FNwimp_getfont
         ("Homerton.Bold.Oblique",12)
80 itext$="(None loaded)"
90 icon2%=FNwimp_createicon(Dummy%,54,-186,276,
         114,%000000101111001,0,0,0,0,
         HomertonBoldOblique12%,itext$,"",0,14,
         "F17" )
100 itext$="Press to print"
110 icon3%=FNwimp_createicon(Dummy%,54,-262,276,
         190,%000000100111101,1,9,7,1,0,
         itext$,"",0,15,"R6,3")
```

As promised, this listing contains the precise statements calling the wimp-functions to produce the **Dummy** template window and its icons. Note the following points:

- arbitrary icon handles are assigned, such as **icon0%** at Line 30. *(We change these a little later)*;
- the icon text is always extracted as a separate line before the call to **FNwimp_createicon** (e.g. Lines 20 and 50);
- where an outline font is used in the icon definition, its handle is obtained before the call to **FNwimp_createicon** in which it is first used (e.g. Lines 40 and 70).

This process takes all the hard work out of it, but there is a little editing to do before we can transfer the lines to our tutorial application.

Firstly, we only need the icons, so the **Dummy** window definition in Line 10 can be deleted. Secondly, we want these icons in our graph window, so all occurrences of **Dummy%** need to be changed to **FuelGraph%**. Thirdly, although not essential, the new icon handles are renamed to be more descriptive than those provided from **!CodeTemps**. We have used **Frame%**, **Label%**, **Driver%** and **Print%**.

Then we need to do some fine adjustment to the icon positions to put them where we want them in the graph window. A detailed look at Lines 30, 60, 90 and 110 above shows that the framing icon surrounds the other three - which latter all have the same size and x-position.

We need to preserve these icon sizes and relative positions and, in case we want to move the icon group at any time, it is clearly best to use variables instead of values.

The new app-function `PROCapp_AddPrintIcons` does the necessary - and it also incorporates the icon text within the `FNwimpcreateicon` calls instead of separately. It is called at Line 580 and defined at Line 13250.

If you make these changes, run the application and select a graph, the new icon group will be seen in the top left corner of the graph screen. This corner has been chosen because it will not be affected by any window resizing.

At the moment, the printer driver icon will still be showing the text used in its creation. We therefore need to add some lines to `PROCapp_GetPaperSizes` (Lines 14060-14080 and 14180-14200) to put the printer driver name (or `'None loaded'`) and enable/disable the `'Press to print'` button.

The result is that, whenever `'Show graph'` is chosen from the iconbar sub-menu and whenever the printer driver status changes when the graph window is already open, the graph window will be resized to the new paper size and its margins and the printer driver name will be updated correspondingly. Also, the `'Press to print'` button will only be enabled if a printer driver is present.

These changes now allow us to proceed with the printing.

Listing reference: We will save the application at this stage as `!Fuel13a`.

14. Printing

The focus for printing from a desktop application is the use of printer drivers loaded by the RISCOS `!Printers` application (maybe enhanced by a ‘turbo’ package).

The Wimp uses the Messaging system via Reason Codes 17 & 18 to interface between an application and the printer driver - and this has to be harnessed to a special control loop sequence involving several `SYS` calls. The RISCOS PRM explains the messaging protocol reasonably clearly but it has to be said that practical implementation of the whole process can be confusing.

The Dr Wimp package employs both wimp- and user-functions to help make the whole process very much easier.

The Dr Wimp printing options

Dr Wimp caters for two types of printing need: firstly, printing something which has already been produced as a window display via the redraw process (i.e. wysiwyg, more or less) and, secondly, printing something which does not necessarily follow anything that is displayed in a window. The Manual calls these the ‘redraw printing’ and the ‘user printing’ methods, respectively.

It will perhaps not be a surprise that the redraw method uses **`FNuser_redraw`** - which was used to produce the screen image in the first place. Alternatively, for ‘user printing’, a new user-function **`PROCuser_print`** is provided.

General points

The process of printing (using Dr Wimp or not) involves several steps which, need to be carried out in the right order. Where Dr Wimp scores, as in its other facilities, is that it makes things a lot simpler.

Dr Wimp provides functions to:

> Check that a printer driver is installed (already introduced);
>
> Check paper size and borders (already introduced);
>
> Convert between paper and screen coordinates;
>
> Check progress of printing, with option to cancel;
>
> Print ranges of pages and number of copies of each;
>
> Fit more than one page to an A4 sheet;
>
> Print in ‘portrait’ and ‘landscape’ orientations;
>
> Declare all fonts to be used, including any in rendered Drawfiles (for Postscript printer drivers).

The Dr Wimp Manual takes you through each of these facilities in a typical sequence, giving sample listings at each step.

As usual with Dr Wimp, the code you have to write yourself relates mainly to the plotting of the unique material you actually want to display/print and getting the sequence of high-level events right. All the tricky interfacing with the Wimp is safely left to the wimp- and user-functions.

There is also an excellent, complete, application called `!PrintTest` (in the `Examples` directory) which takes a textfile and puts it on the screen rather like a word-processor output (with a graphic letterhead on the first page). It allows you to step back and forth through the pages and gives various wysiwyg print options. In fact, this example program amply repays careful examination. It is well commented and gives good insight into the Dr Wimp printing process and several other interesting hints.

Back to the tutorial …

As we produced the graph by the redraw method it is natural that we are going to use 'redraw printing' to get a hard copy of this graph. *(More is said on the 'user printing' method in Chapter 21.)*

In fact, this is good news, because the extra programming to achieve printing is very little indeed. Essentially, all that needs to be done is to carry out another round of coordinate conversions within `PROCuser_redraw`.

You will recall that, to draw the graph, we started off with work area coordinates and changed them to screen coordinates. For printing, we have to change them to 'paper coordinates' instead.

Activating the printing

A single call to the wimp-function `PROCwimp_print`, which has seven parameters, actually kicks things off - for both 'user printing' and 'redraw printing'. The general form is:

```
PROCwimp_print(user%,window%,firstpage%,lastpage%,
     perpage%,copies%,orientation%)
```

The first parameter is used to decide which printing method is to be used: set it to 0 for the redraw method or 1 for the user method. The second parameter is ignored if the first parameter is 1, but otherwise it is set to the window handle of the redraw window involved.

The next three parameters are for use with multi-page documents and will be looked at in a later chapter. For now they can all be set to 1.

The sixth parameter simply determines how many copies (of each page) are to be printed and the final parameter sets whether the printing is to be in portrait (0) or landscape (1) paper orientation.

If the first parameter is set to 1 (i.e. for 'user printing') then the user-function `PROCuser_print` is automatically called and this is where the printing instructions would need to be placed.

If the first parameter is set to 0 (i.e. for 'redraw printing') then `PROCuser_redraw` is automatically called instead - but now with its parameters set to the relevant printing needs.

In particular, the penultimate parameter of `PROCuser_redraw(printing%)` will be set to `TRUE`, and the page number being printed will be placed in the final parameter `page%`. Therefore, in this case, Dr Wimp arranges for `PROCuser_redraw` to be called as many times as is necessary to complete the printing job. All the complication is hidden from us.

The program changes

If the explanation leaves you a bit confused, let's have a look at precisely what is done. You'll find it very easy indeed.

We have already arranged for the `'Press to print'` button in the graph window to be disabled if no printer driver is loaded, so we can now safely kick things off by calling `PROCwimp_print` as our sole reaction to pressing the `'Press to print'` button - whose icon handle is `Print%` (see Chapter 13). Therefore, the first step is to add the following to `PROCuser_mouseclick`:

```
1400 WHEN FuelGraph%
1420 CASE icon% OF
1440    WHEN Print%
1490    PROCwimp_print(0,FuelGraph%,1,1,1,1,0)
1510 ENDCASE
```

This simple routine will set up everything via the PROCuser_redraw parameters and we can now make a few simple changes to respond to these accordingly.

The main change in `PROCapp_DrawGraph` is to add:

```
15280 IF printing%=TRUE THEN
15320     GraphMinX%=FNwimp_worktopaper
             (WorkLeftX%/ScreenScale,0,0)
15330     GraphMinY%=FNwimp_worktopaper
             (WorkBottomY%/ScreenScale,1,0)
15340     GraphMaxX%=FNwimp_worktopaper
             (WorkRightX%/ScreenScale,0,0)
15350     GraphMaxY%=FNwimp_worktopaper(WorkTopY%/
             screenscale,1,0)
15360
15370     GraphTitleCentreX%=FNwimp_worktopaper
             (WorkTitleCentreX%/ScreenScale,0,0)
15380     GraphTitleCentreY%=FNwimp_worktopaper
             (WorkTitleCentreY%/ScreenScale,1,0)
15390
15410     PrintAreaMinX%=FNwimp_worktopaper
             (LeftPrintX%,0,0)+2
15420     PrintAreaMinY%=FNwimp_worktopaper
             ((PaperLength%+BottomPrintY%),1,0)+2
15430     PrintAreaMaxX%=FNwimp_worktopaper
             (RightPrintX%,0,0)-8
15440     PrintAreaMaxY%=FNwimp_worktopaper
             ((-PaperLength%+TopPrintY%),1,0)
15450
15470     RECTANGLE PrintAreaMinX%,
             PrintAreaMinY%,(PrintAreaMaxX%PrintAr-
             eaMinX%),(PrintAreaMaxY%-
             PrintAreaMinY%)
15490 ENDIF
```

This is very much simpler than it looks. It is merely a sequence of calls to the new wimp-function `FNwimp_worktopaper` - converting work values (after ‘un-scaling’) to paper values. (We could have equally well used `FNwimp_screentopaper` to convert from screen values to paper values.)

Line 15470 is an added item. It draws the paper printing margins on the paper for comparison with the displayed ‘white sheet’. This also shows that you can introduce non-wysiwyg items to the printed sheet if required.

(Don’t be concerned about the extra numbers at the end of Lines 15410-15430. They are added just to ensure that the margins actually print. Practical printer driver values can sometimes be rounded to be just outside the printing area rather than just inside!)

Note where these new lines have been inserted i.e. after the conversions of the main graph axes coordinates to screen values and before the calls to `PROCapp_PlotAxes` and `PROCapp_PlotGraph`. Thus, when the `'Press to print'` button has been pressed `printing%` will be `TRUE` and the values passed to the plotting functions will be ‘paper coordinates’ instead of the ‘screen coordinates’.

(If there had not been a good tutorial reason to introduce things in a certain order, it would be better to put the work-to-screen conversions within an `ELSE` *statement before Line 15490 instead of where they currently appear. That way the work-to-screen conversions would not be carried out whether wanted or not - as they are now.)*

The printing will now take place if you try it.

However, you may notice that the text sizes are not quite right - compared with the wysiwyg screen version - and a few additional changes are needed:

```
4890 LargePrintFont%=FNwimp_getfont
        ("Homerton.Bold",24)
16040 IF printing%=TRUE THEN FontPoint%=16 ELSE
        FontPoint%=16*ScreenScale
16130 IF printing%=TRUE THEN FontHandle%=
        LargePrintFont% ELSE FontHandle%=
        LargeScreenFont%
16240 IF printing%=TRUE THEN FontPoint%=12 ELSE
        FontPoint%=12*ScreenScale
```

Taking stock

As you can see, the practice is a lot easier than the written explanation! If you have ever tried to sort out Wimp printing without Dr Wimp you will appreciate just how much is involved and yet it is now rendered fairly painless.

As was said earlier, the example program `!PrintTest`, in the Dr Wimp package, is also really worth delving into to get to grips with printing.

Postscript printers

Before leaving the program in this chapter, it is convenient to mention that Dr Wimp has special facilities to cater for Postscript printers, which require all fonts that are going to be used to be ‘declared’ first.

The Dr Wimp Manual gives fuller details but, essentially, it is done via the user-function `PROCuserdeclarefonts` - note the plural - (at Line 4110 in our tutorial application) which is always called automatically from within `DEF PROCwimp_print` - which, you will recall, is used by both ‘re-draw printing’ and ‘user-printing’.

There is then a choice of two wimp-functions which can be used within `PROCuser_declarefonts` to allow the text-plotting fonts to be declared by specifying the font name or its handle - and a third wimp-function for use in case fonts are defined within a drawfile which is to be printed.

As we will not know whether the user of our tutorial application will be using a Postscript printer or not, it is always safest to declare all the fonts that we are printing - which is only one in this case.

So we can demonstrate the procedure simply by adding the line:

```
4140 PROCwimp_declarefont(”Homerton.Bold”)
```

which uses the wimp-function for declaring the single font by name.

To do the same thing for a text-plotting font whose handle we know then:

```
PROCwimp_declarefonth(fonthandle%)
```

would be used.

And if our application was printing a drawfile - which process is explained in detail in Chapter 19 - we would have used:

```
PROCwimp_declaredfilefonts(dfilehandle%)
```

which would automatically declare any fonts - again, note the plural - within a drawfile which has already been loaded into memory in the manner described in that later chapter. A drawfile will then have a ‘handle’ which is used in the sole parameter above.

Note that we only need to declare the font name for PostScript printers, irrespective of how many different sizes of it might be used. However, if you are using font handles - which, of course, involve both a name and a font size - it is safest to declare all font handles which are used for printing. It will not matter if this action actually declares the same font name more than once.

Listing reference: We will save the application at this stage as `!Fuel14a`.

Specific example

The Examples folder in the Dr Wimp package contains the application `!PrintTest` which shows in further detail how points in this chapter can be implemented in practice.

Error reporting whilst printing

It is a fact of life - rather than any fault of Dr Wimp - that once your program invokes the printing routines (i.e. here, once `PROCwimp_print` is called) you often have difficulty in getting any sensible error reporting if something goes wrong.

This can make it unusually time-consuming to develop a successful sequence if you make even the smallest of errors e.g. a typo in a variable name. The best you are likely to get is the message `"Print job doesn't exist at Line xxxx"` where `xxxx` is the line carrying the error trap rather than the line where the problem is occurring.

You cannot reliably use the old standby of displaying variable values. Possible worse results are a freeze up or quitting without any message.

In these circumstances, the following are offered as possible help:

- use `VDU7` to check that certain stages have been reached.
- report variable values to a file instead (`BPUT`ting them as strings to make them easy to read in the subsequently displayed file).
- send the results of a `TRACE` action to a file, using `TRACE ON/ TRACE TO <file$>/.../TRACE CLOSE/TRACE OFF` sequence. This is usually excellent for showing the line that caused the jump to the error trap, but note that if your program calls a library then it is sensible to number your program and library lines so that - temporarily at least - they do not overlap. This is because the `TRACE` action will faithfully report line numbers in the library if necessary and you may waste time looking in the wrong place.

15. DrWimp library version

We have virtually finished the programming of the tutorial application. All that remains is a sensible housekeeping step.

The Dr Wimp package - like most software - is progressively upgraded from time to time, and you will be well aware by now that the `!RunImage` program and the `DrWimp` library used to develop it are an inseparable pair.

This means that an application produced with `Version X` of the `DrWimp` library may not work with `Version Y` of the `DrWimp` library and it is therefore vital to ensure that the correct version of the `DrWimp` library is used with an application into the future.

If you use !`Linker` (see Chapter 16) you may feel that this will bypass the matter, but this will only be the case as long as you do not want to amend the application later.

It is therefore sensible to keep a copy of each `DrWimp` library version that you use. *If you adopt this philosophy - don't forget to lock each one and hold it in an easily identifiable and unique directory e.g.* `DW365`, `DW380` *etc. It is so easy to overwrite versions when they have the same file name!*

An alternative is religiously to modify each existing application as each new Dr Wimp version is issued. The process is usually very simple and is always described in detail in the supplied documentation (by the `Upgrading` file in the `Documents` directory). But it has to be said that modifying finished applications in this way for no functionality gain requires unusual discipline! It is probably best left until you decide that you need to change/improve the program in some way. By and large, it is then sensible to upgrade to the latest Dr Wimp version at the same time.

Of course, if you are starting a new application then it is **always** best to use the latest version of Dr Wimp. *(Remember that the utility* `!Fabricate`*, which you will probably use when starting the development of a new application - see Appendix 6 - is also matched to a specific* `DrWimp` *library version.)*

Back to the tutorial …

The final step of our tutorial is therefore a practical step to alert us should a mix-up of versions occur.

Dr Wimp has a wimp-function which returns the particular library version number. It is `FNwimp_libversion`, which returns the library version number times 100.

We can therefore add the following routine:

```
80 Library%=3.80*100
420 IF Fnwimp_libversion<>Library% THEN
         PROCwimp_error(appname$,"Wrong 'DrWimp'
         Library Version! It needs to be Version
         "+STR$(Library%),1,1):END
```

If you are not using `DrWimp` Library Version 3.80 you would need to change Line 80 accordingly.

Listing reference: Our tutorial programming is now complete, although there are still some non-programming finishing options to cover. We will save the application at this stage as `!Fuel15a`. *(If you look at the* `!RunImage` *listing you will find several explanatory* `REM` *statements which, to save space, were not shown in the listing extracts in the previous chapters. In most cases they are aide-memoires to the more detailed descriptions in the book.)*

16. Post-programming utilities

The Dr Wimp package comes with a set of very useful utility applications for neatly finishing things off after the programming has been completed. They are mainly aimed at reducing the memory needs of the completed application but they also provide a very good means of protecting your coding from other's eyes - if you have a need to do that. *(But don't forget that these utilities are optional: your application will work quite happily without using them.)*

They are:

```
!Linker       (Freeware)
!StrongBS     (Freeware by Mohsen Alshayef)
!MakeApp2     (Freeware by Dick Alstein)
!Crunch       (Freeware by Bernard Jungen)
```

and they are usually used one after the other in the sequence given (for programs using a library, anyway).

Program size

Before looking at these facilities in more detail, we need to comment on program size in general and - if we are going to reduce the size - it would also be as well to establish exactly what we are starting from.

Running space and WimpSlot

For an application to run successfully, it needs RAM space to:

- hold the `!RunImage`;
- hold any libraries, plus the variables, arrays, parameter blocks, window definitions, workspace, etc. created during the program run.

For the purposes of this book we will call the first item above the "static run space" and the second the "dynamic run space".

The `WimpSlot` allocated to the application is the amount of RAM set aside for running it and therefore needs to be enough to hold both the static run and dynamic run spaces.

It is important to remember that, if we make changes that reduce the static run and/or dynamic run spaces, we will not get the advantage of the reductions unless the `WimpSlot` value is reduced correspondingly after the changes have been made. So, adjusting the `WimpSlot` value is the very last item of action to take.

Storage size

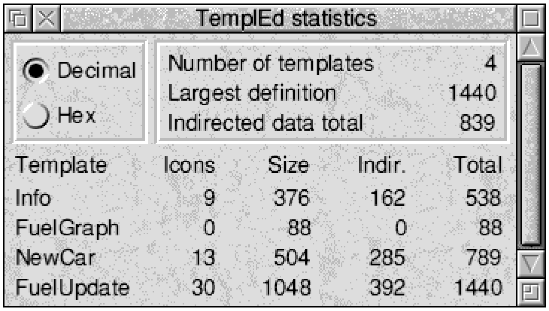
As a separate issue, there is also the disc (or other medium) storage space to consider - which needs to be enough to include all the 'application resources' such as the `!Boot`, `!Run`, sprite and window template files, etc. - as well as the `!RunImage` program itself and any libraries.

Window template statistics

Generally speaking, for given functionality, we can't do a lot about the 'dynamic space' directly, although we have one piece of housekeeping yet to do which can affect that.

The `!TemplEd` utility (see Appendix 7) has a 'Statistics' option available from its iconbar menu which displays the memory needs of any window or template file.

If you load the `Templates` file from `!Fuel15a` into `!TemplEd` and choose this option you will get the following window.



Decimal [✓] Hex

Number of templates	4
Largest definition	1440
Indirected data total	839

Template	Icons	Size	Indir.	Total
Info	9	376	162	538
FuelGraph	0	88	0	88
NewCar	13	504	285	789
FuelUpdate	30	1048	392	1440

For our current purposes, it is the ‘Largest definition’ (1440 bytes) which is of interest.

If you return to Line 170 of our tutorial program you will see:

```
    170 task%=FNwimp_initialise(appname$,7000,300,0)
```

and you may recall (from way back in Chapter 4) that the second parameter is set to the memory space needed to hold the window definition. This is still currently set to 7000 - which is the initial value assigned in `!Fuel4a` (and is also the default value set by `!Fabricate` when you use that).

Clearly, the statistics table shows that we do not need that amount of space set aside. Rather, a value of, say, 1500 would be adequate to cater for the 1440 bytes shown in the table.

To effect this, we change Line 170 to:

```
    170 task%=FNwimp_initialise(appname$,1500,300,0)
```

and this saves 5500 bytes of dynamic run space directly. *(But always check that the program runs OK after such a change. Note the error which occurs if you reduce the value below the 1440.)*

Our final change is to copy the DrWimp library back into the application directory - the place where it would normally sit - and adjust Line 50 accordingly to:

```
50 LIBRARY "<Fuel$Dir>.DrWimp"
```

Listing reference: The program is now, in fact, the true 'final version' of the tutorial application, as far as any further reference to the `!RunImage` listing is concerned. We will therefore save the application at this stage as `!Fuel16a`.

It is fully in the starting state for the use of the 'post-programming finishing facilities' - after which it will not be in a state suitable for reference to the listing.

Starting point values

There is nothing more we can do directly at the moment to reduce the dynamic run space further.

However, we can do a great deal about the static run space directly and, through this, also reduce the dynamic run space indirectly- as well as the disc storage space.

To check where we are starting from though, we can carry out the following routine:

- Add the following temporary line to the `!RunImage:`

```
250 VDU4:PRINT TOP-PAGE;" ";END-TOP:VDU5
```

- Save and run the application and Quit immediately.

You will get a task window display showing the values 56274 and 145190. These are the static run and dynamic run spaces respectively. *(Don't worry if your values are not exactly the same, but they ought to be close to these.)*

PAGE is the memory location in the user RAM of the start of the program. **TOP** is the memory value of its end. So, **TOP-PAGE** gives the program size i.e. the static run space.

END is the highest memory location of the libraries, variables, arrays, etc. loaded/created during the program run. They are automatically placed immediately above the program space, so:

```
END-TOP
```

gives the dynamic run space.

As we Quit without doing anything, the value returned shows the dynamic run space used by the **DrWimp** library plus the initialisation steps only.

(Try it again by Quitting after exercising the program a little - and you will find that the second value is higher - because some more variables, arrays, etc. have now been created.)

Now use 'Count', via the Filer menu, to look at the size of the disc space used to store the **!RunImage** and **DrWimp** files of the application.

At this stage, the static run space of the program - as defined above - will be the same as the disc space used for the **!RunImage** file i.e. 56274 bytes.

The **DrWimp** library size, as counted, is 130459 bytes - so the rest of the dynamic run space is the space taken up by the variables, arrays, etc. set up during initialisation i.e. 145190-130459=14731 bytes.

So, our starting position - using rounded values - is as follows:

1) **!RunImage** length	= 56kbytes
2) **DrWimp** library length	= 130kbytes
3) Initial variables, arrays, etc.	= 15kbytes
4) Static run space (as item 1)	= 56kbytes
5) Dynamic run space (2+3)	= 145kbytes
6) Min. **WimpSlot** needs (4+5)	= 201kbytes
7) Disc storage space (see below)	= 195kbytes

Our current `WimpSlot` size of 256kbytes therefore nicely leaves a margin for the extra dynamic run space that will be needed when the program features are exercised.

At this stage, the disc storage space for the whole application is the static run space plus the size of the `DrWimp` library, `!Run` and `!Boot` files plus the sprites, templates and data files. The counted value is 194155 bytes i.e. say, 195kbytes, as shown.

!Linker

As we said earlier `!Linker` is the first of the special post-programming facilities to use.

What it does is to go through the `!RunImage` of the specific application and the DrWimp library and identifies which wimp-functions have not been used. It then adds all the rest to the `!RunImage` listing to produce a new composite, stand-alone program. As this does not now need the `DrWimp` library, the initial library call can also be deleted. This new program can then become the new `!RunImage` and the `DrWimp` library file can be deleted from the application.

The operation is simple. First, move the `!RunImage` somewhere away from its application directory. Then install `!Linker` on the iconbar and press `<select>` to bring up a window with two file-drag destination boxes. Drag the `!RunImage` to one and the `DrWimp` library to the other. Press "Link" and a Save box appears with a default Basic icon named `!RunImage`.

You should drag this to your original application directory i.e. the one from which the original `!RunImage` came. Then sit back for a few moments and follow the action via the messages which appear. *(If you merely copy your original* `!RunImage` *elsewhere - rather than move it - you will be prevented from dragging the save box icon for the new* `!RunImage` *to the application file and will have to start again. This is a good safeguard to prevent loss of the original* `!RunImage`*. Alternatively, you can drag the save box icon to some other directory and do the changing over later. However, the above suggested sequence is safer.)*

After using `!Linker`, run the application and…… you will get an error!

It points to Line 420 which was our protection against the wrong `DrWimp` library version. This step is clearly now inappropriate and the `!Linker` process has quite sensibly removed the wimp-function. So we need to delete (or `REM`) the contents of Line 420 (and, if you wish, Line 80 - and the `REM` at Line 30 has been changed to remind us for any future development.). In doing this you will note that `!Linker` has already deleted the `LIBRARY` call at Line 50.

The `DrWimp` library itself can now also be deleted from this version of the application.

Listing reference: We will save the 'linked' application at this stage as `!Fuel16b` - noting that the temporary Line 250 is still in place.

The application will now run properly and you can therefore repeat the earlier process of checking the various sizes. You will find the new results are:

`!RunImage` length	= 114kbytes
`DrWimp` length	n/a
Initial variables, arrays etc.	= 15kbytes
Static run space	= 114kbytes
Dynamic run space	= 15kbytes
Minimum `WimpSlot` needs	= 129kbytes
Disc storage space	= 121kbytes

Note that the static run space has increased considerably, because parts of the `DrWimp` library have been added permanently to the `!RunImage` files. However, the dynamic run space needs are reduced by an even greater amount because all of the original `DrWimp` library has now been removed.

The net result is an overall reduction of 72kbytes in both the `WimpSlot` and disc space needs, which represents those parts of the `DrWimp` library that we are not using in this particular application.

(That's over half the `DrWimp` *code - there's still a lot for this book to cover!)*

!StrongBS

This is a Basic program compactor - which means that it may well be impossible to modify the program in its compacted state.

So, before taking the next step, copy `!Fuel16b` as `!Fuel16c` and change Line 120 accordingly. (It doesn't matter about the header `REM`s because they will be eliminated by the process anyway.) Also, if you want to check the sizes detail, make sure the temporary Line 250 is still in place.

Having taken these preliminary steps, the procedure is again straightforward. Load `StrongBS` onto the iconbar and press `<select>` over it to bring up the operating window.

Now press `<menu>` over this window; move across Mode and select Full2001.

The `!RunImage` file is now dragged to this window and "Squash" selected.

After completion, drag the Basic file icon (called `!RunImage`X if you have left things untouched) to the `!Fuel16c` application directory. Delete the original unsquashed `!RunImage` and rename `!RunImageX` as the new `!RunImage`.

Listing reference: We will save the 'linked and squashed' application at this stage as `!Fuel16c` - noting that the temporary Line 250 is still in place.

Check that the application runs OK and then repeat the checks for the sizes. The figures should be:

`!RunImage` length	= 35kbytes
`DrWimp` length	n/a
Initial variables, arrays etc.	= 13kbytes
Static run space	= 35kbytes
Dynamic run space	= 13kbytes
Minimum `WimpSlot` needs	= 48kbytes
Disc storage space	= 42kbytes

This step reduces the static run space considerably - mainly due to the elimination of `REM`s and the long variable names in the `!RunImage`. But note that the dynamic run space is also reduced a little - due to shorter variable/array names.

(Remember that it is unwise - and sometimes impossible - to modify a compacted Basic program. If modifications are needed you need to modify the original program and then run it through !StrongBS again. So always keep a copy of the original!)

!MakeApp2

This utility converts a Basic program into ‘absolute code’ - which effectively means that it ceases to be a Basic program and becomes a sequence of bytes which look like gibberish in a text editor and will not operate as a program without help. So the `!MakeApp2` conversion process gives this help by adding a special header to the file which automatically ensures that, when run in the usual way, Basic mode is entered and the listing restored - and that the Basic environment is quit after the program ends.

The consequence is that the use of `!MakeApp2` on its own actually increases the static (and dynamic) size very slightly. Its main advantage on its own is that it provides an effective security facility by making the `!RunImage` file totally incomprehensible.

To see it in action, first copy `!Fuel16c` as `!Fuel16d` (and change the sprite name, as usual). Then load `!MakeApp2` and simply drag the `!RunImage` file to the iconbar icon. The resulting Save box will show the standard Wimp App directory icon (with a default name of `"Ab"`) which can be dragged to the `!Fuel16d` application directory and renamed to replace the `!RunImage` from whence it came. The new sizes will be very slightly larger than before but when rounded will be:

`!RunImage` length	= 35kbytes
`DrWimp` length	n/a
Initial variables, arrays etc.	= 13kbytes
Static run space	= 35kbytes
Dynamic run space	= 13kbytes
Minimum `WimpSlot` needs	= 48kbytes
Disc storage space	= 43kbytes

!Crunch

However, the second advantage of using !MakeApp2 is that the resulting absolute file (unlike a Basic file) can then be ‘crunched’ further, by !Crunch, to reduce the disc storage space significantly - **but not the actual running needs**.

The process is simple but not quite the same as before. After loading `!Crunch` onto the iconbar and dragging the absolute version of `!RunImage` to it, a Save box with a blank file icon (with default name `"Crunched"`) appears. This needs to be dragged to the application’s directory and the crunching then starts. On completion the file icon changes to the standard Wimp `App` directory icon.

The final sizes are then:

`!RunImage` length	= 20kbytes
`DrWimp` length	n/a
Initial variables, arrays etc.	= 13kbytes
Static run space	= 35kbytes
Dynamic run space	= 13kbytes
Minimum `WimpSlot` needs	= 48kbytes
Disc storage space	= 27kbytes

We therefore end up with a file which needs less space to store on disc than it's static space when loaded for running.

Revisit WimpSlot

Having carried out all the above processes, it is time to have a final look at the `WimpSlot`.

Our final result indicates a minimum `WimpSlot` need of 48kbytes, but remember that this figure is not quite sufficient because it only took account of the dynamic run space needs of the initialisation process of the program. A somewhat larger value will therefore be needed in practice.

A sensible way to proceed (with the temporary contents of Line 250 in place - but now inextricably buried in the crunched version!) is to run the completed application through its range of user options before quitting. This will give a larger value, to which a small margin should be added just in case. The `WimpSlot` value in the `!Run` file can then be changed to that.

Using this process a value of 56k appears to be a good `WimpSlot` choice for our final version.

Trial and error is then the only way to check whether this value is enough for all circumstances, or whether it can be trimmed further.

The chosen `WimpSlot` value is 30% of the original need (i.e. compared with `!Fuel16a`) which is a very worthwhile reduction.

Listing reference: We will save this fully processed application as `!Fuel16d` - noting that the temporary Line 250 is still in place.

However, if you have followed the above sequence precisely as described, `!Fuel16d`, when run, will show “V16c” on the icon bar, because we could not get at the original Line 120 after squashing the program.

Therefore, to add final polish:

i) replace the `!RunImage` of the above `!Fuel16d` with a copy of the `!RunImage` from `!Fuel16b`;

ii) in this `!RunImage`, change Line 120 to `"16d"` and delete (or `REM`) the temporary entry in Line 250;

iii) then repeat the use of `!StrongBS`, `!MakeApp` and `!Crunch`.

Listing reference: This new `!Fuel16d` is the fully finished version and matches the version on the supplied disc.

Summary

The following table shows the results of the various post-programming processing stages together, for easier comparison.

Naturally, the reductions achieved depend somewhat on programming style and in this tutorial exercise it is inevitable that more `REM`s were used than you might “in the privacy of your own home”. Nonetheless, it is easily seen that the post-programming facilities offered in the Dr Wimp package are both very simple to use and produce very well worthwhile results, particularly for those who wish to distribute programs.

As the table shows, `!Linker` and !`StrongBS` produce the main reductions and `!MakeApp` is best regarded as a good security step which incidentally opens the door to further disc-space saving.

Don’t forget that a great deal of protection (and size reduction) is provided merely by using `!StrongBS` on its own. It is very difficult to follow a compacted Basic listing - and often the program will not save if modifications are attempted in this form.

	'static run'	'dynamic run'	WimpSlot	Disc space
At start	56	145	201	195
!Linker	114	15	129 (64%)	121 (62%)
!StrongBS	35	13	48 (24%)	42 (22%)
!MakeApp	35	13	48 (24%)	43 (22%)
!Crunch	35	13	48 (24%)	27 (14%)

Progressive effect of post-programming size reductions using Dr Wimp utilities

Our tutorial exercise is now finished. The following chapters will cover many more features of the Dr Wimp package as separate topics although some references are still made to the tutorial program to assist understanding. As indicated earlier, `!Fuel16a` is the most useful reference version as it is the complete version just prior to post-programming processing.

17. More on menus

In our tutorial, using a simple iconbar menu/sub-menu ‘tree’, we showed how Dr Wimp firstly creates the definitions of menus and sub-menus separately - and then, subsequently, causes them to be displayed as required.

Menu manipulation is very important to Wimp programs and so Dr Wimp has a comprehensive range of facilities to offer.

Menu definition/creation

Three methods of menu/sub-menu creation are provided:

```
FNwimp_createmenu - already used in the tutorial
         application.
FNwimp_createmenuarray
FNwimp_createmessagemenu
```

One of the shortcomings of `FNwimp_createmenu` is that the items are defined by the ‘slash separated’ string, which is therefore limited to the maximum length of a Basic string e.g. 255 characters.

`FNwimp_createmenuarray` and `FNwimp_createmessagemenu` both avoid this problem.

FNwimp_createmenuarray

Firstly, a string array needs to be DIMmed: let’s call it `array$()`. The elements of this array are then used to hold the required menu items.

The string in the first element - `array$(0)` - is used for the title of the menu.

The next element - `array$(1)` - will hold the top item of the menu (item 1) and so on. **The element after the last menu item must contain the string "END"** - in capitals but without the quotes.

This means that the array is normally `DIM`med with a number one higher than the maximum number of menu items to be used. e.g. if there are to be 6 menu items, then `DIM array$(7)` would be used (8 elements, from 0-7). On initial creation, this array would have the menu title put in `array$(0)` - and `array$(7)` would be assigned the string "END". The elements 1 to 6 would be filled with the required menu item strings, in order from the top downwards.

When the array is complete, a call to

```
     FNwimp_createmenuarray(array$(),size%)
```

creates the required menu and returns its handle.

The first parameter passed is the name of the array and the second is the required maximum size of the array (which can be larger than the size of `array$()`, but would normally be one less, as indicated above).

Once the menu has been created the array is not needed any longer and so can be reused for creating/recreating other menus for instance.

FNwimp_createmessagemenu

This method offers even greater flexibility and uses the RISC OS 'Messages files' procedure *(not to be confused with the 'Wimp Messaging system'!).*

Message files are simply text files which contain lists of text items in a simple specified format. They are very commonly used in applications to hold the text of error messages etc. Chapter 26 (and the Dr Wimp Manual) contains fuller details on Messages.

The message file format requires each listed item of text to be preceded by a '`token`' (or 'tag') such as:

```
     #This is a comment line (ignored).
     ibarT:Fuel
     ibar1:Info
     ibar2:New Car
     ibar3:Quit
```

In this case the token is “ibar” and you can see that there are four items using that prefix - together with another character and separated from the item text by a colon. *(A single Message file might have many more lists, each with their own unique token.)*

For a menu, the first item is preceded by “ibarT” and this signifies that the text is the title for the menu. The ordinary menu items then follow with the prefix “ibar1”, “ibar2” etc. representing the required menu item text in order from top to bottom. It is very important that this order is strictly kept.

Once this Message file is in place the procedure is firstly to ‘initiate’ the Messages file with:

```
messagefilehandle%=FNwimp_initmessages(filepath$)
```

This also sets up some special memory blocks for later use.

The returned handle is then used to create a menu with the call:

```
menuhandle%=FNwimp_createmessagemenu
     (messagefilehandle%,"ibar",0)
```

The specific token string is needed in the second parameter to ensure that the correct list is used from the Message file - which, as had been said, may hold more than one list.

As before, the final parameter of 0 means that the maximum size of the menu will be the number of items initially used. But it can be set instead to a required higher number if the menu is to be enlarged later in the program.

Menu re-creation

Once a menu/sub-menu has been created by any one of the creation methods it can be completely re-created - within the maximum size defined at creation - using any one of the following:

```
PROCwimp_recreatemenu
PROCwimp_recreatemenuarray
PROCwimprecreatemessagemenu
```

PROCwimp_recreatemenu

This re-builds an existing menu using a ‘slash separated’ string. The call is simply:

```
PROCwimp_recreatemenu(menu%,menu$)
```

where **`menu%`** is the existing menu handle and **`menu$`** is the ‘slash separated’ string.

PROCwimp_recreatemenuarray

The call this time is:

```
PROCwimp_recreatemenuarray(menu%,array$())
```

The array is simply filled with the new title/item text exactly as if it was being used to create a new menu (see earlier).

The only important thing to remember is that “END” must be put in the array element after the last item of the intended new menu.

PROCwimp_recreatemessagemenu

The call this time is:

```
PROCwimp_recreatemessagemenu(menu%,
    messagefilehandle%,token$,title$)
```

This time we have four parameters. Again, **`menu%`** is the existing menu handle.

`messagefilehandle%` is the handle of the Message file to be used for the re-creation - and must have been previously ‘initialised’ as described in the creation process. **`token$`** is the token of the menu item text list to be used from this new Messages file.

`title$` is an added feature: if it is set to a null string then then the title defined in the new Message file will be used - but if it is not a null string then `title$` will override whatever title is in the Messages file.

One of the uses of menu creation/re-creation using Message files is to prepare menus dynamically from user data files - whose number and/or content are, of course, unknown. For instance, it is easy to read a user's directory to count the number of data files; extract their leafhames into a Messages file; from which a menu of data files can be presented. The process can then be repeated freely at any time during the application run to re-create the menu with the up-to-date number of data files.

Further menu manipulation

Once a menu/submenu has been created there are a number of manipulations that can be applied to individual items or the title text. For instance, there are wimp-functions to:

Enable/disable menu items

Add/remove an individual item

Change item or title text

Add/remove a menu tick

Add/remove a 'dotted line'

Change the colour of a menu item

Change an item to 'writable'

There are also wimp-functions to find the current state of most of the above features.

The only disadvantage is that these features have to be re-applied if any of the menu re-creation methods is used.

Menu position

If we use `FNuser_menu` to determine the display of a menu it will be displayed according to the Wimp's normal rules - which we have already seen in our tutorial.

However, Dr Wimp also offers further menu display options by the use of:

```
PROCwimp_menupopup(menu%,pos%,x%,y%)
```

where `menu%` is the handle of the menu (or, indeed, a window) to be displayed and `pos%`, `x%` and `y%` determine the displayed position as described below.

If `pos%=0` then the menu is displayed with its top left corner at **screen** OS coordinates `x%/y%`.

If `pos%=1` then the menu is displayed as if it were an iconbar menu with its left edge at `x%` i.e. its bottom edge will always be 96 OS units above the bottom of the screen. *(The `y%` value will be ignored but must be included.)*

If `pos%=2` then the menu will be centred on the screen. *(The `x%` & `y%` values will be ignored but must be included.)*

If `pos%=3` then the menu is displayed slightly to the right of and slightly above the pointer position (optimised to butt onto right edge of the 'ptr_menu' shape). *(The `x%` & `y%` values will be ignored but must be included.)*

If `pos%=4` then the menu is displayed with its left edge butting against the right edge of the icon over which the mouse is clicked. *(Designed to be used with 'pop-up menu icons'.) (The `x%` & `y%` values will be ignored but must be included.)*

Calling this at anytime (e.g. from a `<select>` or `<adjust>` mouse-click over a specific icon) will bring up ('pop up') the menu (or window) immediately. It will also act entirely like a menu - that is, the menu (or window) will close again if the mouse is clicked anywhere else on the screen.

PROCuser_overmenuarrow

We have mentioned this user-function but not yet introduced it. It provides yet another means of changing menus dynamically.

The empty (default) function definition is:

```
DEF PROCuser_overmenuarrow(RETURN
          nextsubmenu%,parentmenuitem%,x%,y%)
ENDPROC
```

This user-function is automatically called whenever the user 'moves over' a menu arrow leading to a sub-menu.

When called (by the `DrWimp` library, as usual) `nextsubmenu%` holds the handle of the sub-menu about to be displayed and `parentmenuitem%` holds the menu item number to which the submenu is 'attached'. `x%` and `y%` are the (screen OS) pointer position coordinates when over the arrowhead.

Note the `RETURN` with the first parameter: this implies that the value of `nextsubmenu%` can be changed by the programmer within the coding of this user-function definition.

Font menus

From applications such as !Draw you will be familiar with the use of a menu to show the list of outline fonts currently available to you on your particular machine.

In fact, as you will know, a font menu is invariably a small menu ‘tree’ with the sub-menus offering a choice of variations within each main font family, e.g. “Trinity” on the main menu, with its variations “Bold”, “Bold.Italic”, “Medium” and “Medium.Italic” on a sub-menu.

Dr Wimp allows you to provide the same facility in your own applications - either as a separate menu or with the font menu starting as a sub-menu to any other menu/sub-menu.

Creation of a font menu is effected by:

```
FNwimp_createfontmenu (no parameters)
```

and this returns a handle which you can use as normal to display the menu or attach it as a sub-menu to another menu/sub-menu item.

Note that you do not have to worry about specifying a maximum size for a font menu.

If you want to recreate an existing font menu you need to use:

```
FNwimp_recreatefontmenu(fontmenu%)
```

and in practice it would be normal to use this prior to each opening of the font menu in case the user has altered his/her fonts since the menu was last opened (quite a normal occurrence to be catered for).

The only restrictions on using font menus is that you cannot manipulate them with, for example, the wimp-functions for enabling/disabling, adding/removing items, etc. - but Dr Wimp will give you a ‘non-fatal’ warning you if you attempt these.

Selections from a font menu

As with all menus, when we make a selection from a font menu the following user-function comes into play:

```
PROCuser_menuselection(menu%,item%,font$)
```

The only difference is that the second parameter, `item%`, is now not used (and is automatically set to 0 by Dr Wimp) and the third parameter (instead of being set to a null string) now gives the selected font as a full 'period separated' font string e.g.:

```
"Trinity.Medium.Italic"
```

It is worth noting that although there is, at any one time, really only one font menu it is sometimes necessary to create more than one within the same application i.e. each with different menu handles. A reason for this would be to ensure that selections from different font menus lead to different actions in the application.

Finally, the wimp-function:

```
PROCwimp_puticonfont(window%,icon%,fonthandle%)
```

is provided specifically to allow the font of text within an icon to be changed - as long as its icon definition specifies that an outline font is used. *(This will automatically mean that the icon will also be 'indirected'.)*

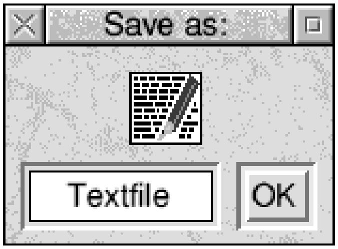
Listings of typical font menu routines are contained in the Dr Wimp Manual.

18. Saving and loading data

Saving data

In the tutorial application we saved the car fuel data directly to a predetermined directory with an automatically constructed filename.

However, many applications would normally use the standard Save window - shown below in a text file form.



This window has three icons, the draggable file icon, the writable icon and the OK button. (These are given the handles **`drag%`**, **`write%`** and **`ok%`** respectively in the standard Dr Wimp programming - see below.)

The writable icon needs to be indirected and it is best if the draggable icon is also indirected.

Dr Wimp provides a group of three functions to enable this standard Save window to be harnessed painlessly (and without any worry about which icon button types to use). They are:

```
FNuser_savefiletype(window%)
PROCuser_saveicon(window%,RETURN drag%,RETURN
         write%, RETURN ok%)
FNuser_savedata(path$,window%)
```

Your program must first have the above Save window loaded - and we will assume that its handle is `Save%`. **(A copy is included in the Dr Wimp package in `Template4` of the Tutorial directory. Its icons are already of the required button types.)**

In addition you will need to cause the window to be opened when needed, from a mouse click and/or a menu selection, using the normal Dr Wimp methods covered already. (Don't forget that a window can be attached as a 'sub-menu' to a menu item in exactly the same way as a real sub-menu. Just use a window handle instead of a menu handle in `PROCwimp_attachsubmenu`.)

Then, the first programming step is to decide the filetype of the data to be saved. Let's assume that it is a textfile and hence the filetype `&FFF` is appropriate. We simply make changes to `DEF FNuser_savefiletype` on the lines of the following:

```
DEF FNuser_savefiletype(window%)
Type$=""
CASE window% OF
      WHEN Save%
      Type$="FFF"
ENDCASE
=Type$
```

Once this has been done, the filetyping of the save action (see below) will be taken care of by Dr Wimp automatically.

The particular routine above allows for easy addition of further save windows, but an `IF ... THEN` statement could have been used equally well.

The draggable file icon needs to match this chosen filetype, so `PROCwimp_puticontext` can be used to change the sprite displayed in the icon if need be.

Next, it is necessary to check the icon numbers used for the three Save window icons. Dr Wimp will assume that the drag/writable/OK icons are numbered 0, 1 and 2 respectively unless you use `PROCuser_saveicon` to tell it otherwise.

If, in your case, the corresponding icon numbers are 6, 17 and 5, then a typical usage might be:

```
DEF PROCuser_saveicon(window%,RETURN drag%,
         RETURN write%,RETURN ok%)
IF window%=Save% THEN
      drag%=6
      write%=17
      ok%=5
ENDIF
ENDPROC
```

All is now ready to take the actual saving action, which is put inside `FNuser_savedata(path$,window%)`.

This user-function is called from `DrWimp` when the draggable save icon is dragged to a directory window or the OK button is pressed (if a valid path has been put into the writable icon). A typical sequence - for a text file - might be:

```
DEF FNuser_savedata(path$,window%)
LOCAL ERROR
ON ERROR LOCAL =2

IF window%=Save% THEN
      file!=OPENOUT(path$)
      BPUT#file%,"This is the Heading text"
      BPUT#.file%,"First text line."
      etc.
      etc.
      CLOSE#file%
      PROCwimp_menuclose
ENDIF

=1
```

There is a very important point in the above routine which is easy to miss - and that is the return value of 1. (Note that the default return is 0 i.e. in the ‘empty’ state.)

The return value of 1 tells Dr Wimp that you have actually taken some saving action and therefore want the automatic filetyping etc. to take place.

The above sequence will cause **FNuser_savefiletype** to return a 1, or 2 if an error occurs.

There is also a use for returning 0 - which, in general, is when no save action is taken. For instance, if the file we wished to save already exists we might wish to give the user the option to overwrite it or cancel the save action. If cancel was chosen then we would not start the saving action and would return 0 instead.

So a revised sequence might then be:

```
DEF FNuser_savedata(path$,window%)
LOCAL ERROR
ON ERROR LOCAL =2

Used%=0
IF window%=Save% THEN
      Choice%=TRUE
      file%=OPENIN(path$)
      IF file%>0 THEN
         CLOSE#file%
         Choice%=FNwimp_errorchoice(app$,"This
            file already exists. Press 'OK' to
            overwrite it.",2)
      ENDIF

      IF Choice%=TRUE THEN
         file%=OPENOUT(path$)
         BPUT#file%,"This is the Heading text"
         BPUT#.file%,"First text line."
         etc.
         etc.
         CLOSE#file%
         Used%=1
      ENDIF
ENDIF
=Used%
```

If required, a default file path could also be placed in the writable icon of the Save window using `PROCwimp_puticontext`.

It is therefore very easy to handle Save windows and there is no reason why an application cannot have several of them and/or several different saved file types and contents. In the Dr Wimp `Examples` directory, the application `!Saver` shows how easy it is to change the filetype and the sprite in the draggable icon - to achieve, for instance, multiple save options.

Loading data

Complementary to the above, Dr Wimp provides facilities to help with loading data from existing files into your application - for example, so that data can be extracted from a file for manipulation by the application.

Only one user-function is needed for this:

```
FNuserloaddata(path$,window%,icon%,ftype$,
     workx%,worky%)
```

and this is called from `DrWimp` **whenever** a file (or directory or application) is dropped onto a window of your application - or even when it is just ‘double-clicked’.

The full path of the file, its filetype, the window/icon and position it was dropped into are passed to you by `DrWimp` in the 6 parameters.

You simply have to fill the `DEF FN` with the necessary loading instructions, using any of the passed parameters you wish.

If we follow on from the preceding saving example, a typical loading sequence for a text file might be:

```
DEF FNuser_loaddata(path$,window%,
         icon%,ftype$,workx%,worky%)
Used%=0
IF ftype$="FFF" THEN
      file%=OPENIN(path$)
      N%=0
      REPEAT
          N%+=1
          Array$(N%)=GET$#file%
      UNTIL EOF#file%
      CLOSE#file%
      Used%=1
ENDIF

=Used%
```

Clearly, the array to hold the file data would need to be set up previously, preferably in `PROCuser_initialise`.

When a file is 'double-clicked', the `window%` and `icon%` parameters are set by Dr Wimp to 0 and -1 respectively - and the final two parameters are both set to -1. Thus, in this case, the contents of `path$` and `ftype$` will be of main interest.

Dr Wimp also provides wimp-functions for loading image files (sprites, drawfiles and JPEG files) into memory, usually prior to their display by an application. This is covered in the next chapter.

Specific examples

The `Examples` folder in the Dr Wimp package contains the applications `!Blocks`, `!Dynamic` and `!Saver` which show in detail how points in this chapter are implemented in practice.

19. Handling image files

There are many circumstances where you want your application to display and/or print an image from a sprite-file, a drawfile or a JPEG file - and Dr Wimp has three broadly similar suites of wimp-functions to help with this.

With all three types of image a common feature of the RISCOS process is that the the file contents need to be copied into RAM before display and/or printing can take place. *(This is not strictly true for JPEGs, where an option exists to display - but not print - a JPEG directly from file. Dr Wimp also offers this option.)*

> *We have already met this need to load the image into RAM in a slightly different way in our tutorial. You'll recall that it was necessary to load the* `!Sprites`/`!Sprites22` *files into a special part of RAM called the 'Wimp sprite pool' before, say, the application icon would appear correctly in a Filer window. This was done there with the Star Command* `*IconSprites` *in the application's* `!Boot` *and* `!Run` *files. Generally speaking you should only use the Wimp sprite pool for sprites which are intended for use by any application - particularly by the Filer.)*

Before proceeding with the detail it is also worth remembering that a single sprite-file can hold more than one sprite image, each of which can be extracted independently - whereas drawfiles and JPEG files hold only one image per file.

Common steps

Although the process is simple, it is absolutely essential that the following sequence is used with any image file:

> **1) Measure the size of the file(s) using a specific wimp-function;**
>
> **2) Set aside a memory block of the corresponding size;**
>
> **3) Load the image file(s) into that block.**

It is important that only `FNwimp_measurefile` is used for Step l, because it not only finds the size of a file but it also sets up a few extra bytes used behind-the-scenes by Dr Wimp. These extra bytes are vital to the correct working of the associated wimp-functions used for displaying/ printing the images.

For Step 2, the choice is between using `DIM` to set aside a block of memory for holding the image(s), or to set up a Dynamic Area instead. Chapter 26 (or the Dr Wimp Manual) covers the latter in more detail, but we will be using only `DIM` in our sample listings below.

For Step 3, the appropriate wimp-functions are:

```
FNwimp_loadsprites(file$,address%)
FNwimp_loaddfile(file$,address%)
FNwimp_loadjpegfile(file$,address%)
```

All three operate in the same, unusual, way in that they are designed to facilitate more than one file to be loaded into a memory block contiguously. So they load the file `file$` into a block of memory starting at `address%` - but they return the address of the start of the next block of memory.

An example will make this clear. Let’s assume you have three sprite-files to load - with leafnames `SpritesA`, `SpritesB` and `SpritesC`. A typical sequence would be:

Step 1

```
Size%=FNwimp_measurefile("<App$Dir>. SpritesA")
Size%+=FNwimp_measurefile("<App$Dir>.SpritesB")
Size%+=FNwimp_measurefile("<App$Dir>.SpritesC")
```

Note the += actions in the second and third lines: this ensures that `Size%` ends up being large enough to cater for all three files.

Step 2

```
DIM SpriteArea% Size%
```

Step 3

```
A%=SpriteArea%
B%=FNwimp_loadsprites("<App$Dir>.SpritesA",A%)
C%=FNwimp_loadsprites("<App$Dir>.SpritesB",B%)
X%=FNwimp_loadsprites("<App$Dir>.SpritesC",C%)
```

The three sprite-files are loaded one after the other into the memory block at `SpriteArea%` with their respective 'sub-block' areas starting at `A%`, `B%` and `C%` - which are now the respective handles of each sprite-file. (`A%` is the same as `SpriteArea%` and is only used for consistency.)

Note that we have no use for the return from the third loading action and so `X%` is a dummy value - and is actually the address of the first byte after the end of the block which has been set aside.

If we had only one sprite-file to load then Step 1 would require just one sizing action, and Step 3 just one loading action.

The procedure is exactly the same for drawfiles and JPEG files - but using the corresponding loading wimp-function in Step 3.

With the files loaded into memory we can now simply refer to their handles in the subsequent display/printing actions.

In all cases, display of an image on the screen means using PROCuser_redraw, so don't forget that the 'auto redraw' flag in the window definition must be unset for the 'redraw' process to work. For printing, either 'redraw' or 'user' printing can be used - as defined in Chapter 14.

There are slight differences in the way the three types of image files are handled for display/printing so we now need to look at each individually in turn.

Displaying/printing sprites

For a sprite, the memory area into which we have loaded the sprite-file is usually called a 'user sprite area' and so we will follow that nomenclature here.

It is also worth remembering that a sprite is a bit-image format and not tied to any 'page' format.

Dr Wimp offers two main wimp-functions for displaying sprites on the screen, which also work for printing them. In both cases, the sprite to be used must already be loaded into a user sprite area as described above.

As we have said, the unique point about a sprite-file is that it can hold more than one sprite image. Consequently, in order to display a particular sprite image we have to identify the user sprite area and the sprite name within it.

You will see this reflected into the two wimp-functions:

```
PROCwimp_rendersprite(sprite$,sprite%,bx%,by%,
          minx%,miny%,maxx%,maxy%,
          xscale,yscale)
PROCwimp_renderwindowsprite(window%,sprite$,
          sprite%,bx%,by%,minx%,miny%,maxx%,
          maxy%,xscale,yscale)
```

For display ('rendering') both these wimp-functions are intended to be used within `PROCuser_redraw`. For printing, either 'redraw' printing via `PROCuser_redraw` or 'user printing' via `PROCuser_print` can be used.

The first wimp-function draws a sprite directly onto the screen and the second draws it in a window. The only difference between the two parameter lists is the addition of `window%` in the second case, so we need only describe the list once.

The name of the particular sprite is entered as a string into `sprite$` and the handle of its user sprite area is placed in `sprite%`.

`bx%` and `by%` are the OS coordinates of the bottom left corner of the required sprite position. In the first case they need to be in screen coordinates, whilst in the second case they need to be in work area coordinates. So the *y*-values will need to be negative in the second case.

`minx%`, `miny%`, `maxx%` and `maxy%` are the OS coordinates (in screen coordinates in **both** cases) of the 'clipping' rectangle. The sprite will only be drawn (and then in full) if **any** part of it lies within this clipping rectangle. As the wimp-functions will be used within `PROCuser_redraw` or `PROCuser_print` - both of which will hold the clipping rectangle coordinates in their own parameters - these coordinates can simply be transferred directly to our wimp-functions without worrying about them.

`xscale` and `yscale` are largely self-explanatory. Note that their values are generally non-integer - values less than 1 reduce the displayed size and values more than 1 increase it.

Printing sprites

Printing sprites is straightforward, using either the 'redraw' or 'user' method. It is simply a matter of calling

`PROCwimp_rendersprite` or `PROCwimp_renderwindowsprite` with the `bx%` and `by%` values converted to 'paper coordinates' as described in Chapter 14.

Sprites in the 'Wimp pool'

Dr Wimp recognises that there will be occasions when you want to display/print a sprite which is already known to be in the Wimp sprite pool.

Accordingly, there is a pair of special wimp-functions for this purpose. They are:

```
PROCwimp_renderpoolsprite(sprite$,bx%,by%,minx%,
          miny%,maxx%,maxy%,xscale,yscale)
PROCwimp_renderwindowspoolsprite(window%,sprite$,
          bx%,by%,minx%,miny%,maxx%,maxy%,
          xscale,yscale)
```

As you can see, they are identical to their 'user sprite area' counterparts except for the absence of the sprite area handle. So no further explanation is needed - apart from the good news that, for obvious reasons, you can use these calls without the need to go through the common measuring and loading sequence.

Displaying/printing drawfiles

Although drawfiles contain only one image, they have the added complication that they often contain text using outline fonts and they can also include sprite and/or JPEG images. Further, a drawfile exists in relation to the 'page' on which it is drawn.

Accordingly the Dr Wimp display/print options for drawfiles need to take these into account and the wimp-functions hide an unusually large amount of complication behind the scenes.

However, as usual, all this is hidden from the Dr Wimp user who only has to remember to take one other very simple step during the common preparatory file measuring and loading sequence and that is to make a single call to `PROCwimp_initdfiles`. This sets up some special memory blocks/arrays to handle most of the complications automatically. *(You only have to call it once irrespective of how many drawfiles are involved. And it doesn't matter whether the call is made before or after the measuring/loading sequence - just as long as it is before attempting to display/print the drawfile.)*

So, along with measuring the drawfile(s), setting up the memory block and loading the file(s) into the block - using **`FNwimp_loaddfile`** this time (which gives us a drawfile handle) - the single call:

```
PROCwimp_initdfiles
```

is made.

Following this, displaying drawfiles takes place using either of the following wimp-functions, both of which use the drawfile handle (here called **`dfile%`**) from the common loading sequence:

```
PROCwimp_render(dfile%,bx%,by%,minx%,miny%,
          maxx%,maxy%,scalex,scaley,origin%)
PROCwimp_renderwindow(window%,dfile%,bx%,by%,
          minx%,miny%,maxx%,maxy%,
          scalex,scaley,origin%)
```

and you can see that, apart from the additional final parameter, they are essentially the same as those for displaying/printing sprites.

The extra parameter **`origin%`** is set to either 0 or 1. If set to 0 the drawfile is displayed with the bottom left corner of its ‘page’ at **`bx%/by%`** - and if set to 1 it is displayed with the bottom left corner of the bounding box surrounding all its ‘objects’ at **`bx%/by%`**. *(In effect, you have the choice of eliminating any white space below and to the left of the drawfile image on its ‘page’. The Dr Wimp manual contains a diagram which explains this in detail.)*

Apart from this point the use of the wimp_functions follows that of the sprite case.

Printing drawfiles

For printing drawfiles there is one extra aspect to take care of: the need to 'declare' any outline fonts that may be included in the drawfile, in case a PostScript printer is being used.

This was mentioned initially in Chapter 14 where the role of `PROCuser_declarefonts` in both 'redraw' and 'user' printing was introduced.

When drawfiles are being printed we simply use:

```
     PROCwimp_declaredfilefonts(dfile%)
```

within `PROCuser_declarefonts` - using the drawfile handle of the drawfile involved.

It is always safest to use this font declaration process when printing drawfiles - even if you are sure that the particular drawfile doesn't contain text. It will do no harm.

Text areas in drawfiles

Although ordinary text within a drawfile is rendered by the above wimp-functions, Dr Wimp will not cope with any specialised 'text areas' i.e. multi-line blocks of text in a drawfile - which are usually made by dropping a text file into a frame within the drawfile page.

Dr Wimp will not throw an error in these cases - it will simply not render such areas. Fortunately they are not a common feature.

However, Dr Wimp has very good facilities for plotting text directly to the screen, as is covered in the next chapter……

Displaying/printing JPEGs

If you are using RISC OS Version 3.60 or higher then you will be able to use Dr Wimp to display/print JPEG images.

The JPEG format is very popular, particularly for digital camera photo images. It is a bit-image format and (unlike sprites) there is only one image per file.

As with sprite-files and drawfiles you must carry out the standard preparatory sequence to measure the file(s), set up a memory block and load the JPEG files into this block in the previously described manner - but this time using `FNwimp_loadjpegfile` for the loading action.

No additional actions are needed and displaying JPEGs (or printing them) can then be carried out with:

```
PROCwimp_renderjpeg(jpeghandle%,bx%,by%,minx%,
            miny%,maxx%,maxy%,scalex,scaley)
PROCwimp_renderwindowjpeg(window%,jpeghandle%,
            bx%,by%,minx%,miny%,maxx%,maxy%,
            scalex,scaley)
```

These are entirely similar to the sprite case - but without the need to specify a particular image name, as there is only one image per file. Accordingly, no further explanation is needed.

Displaying JPEGs without loading into memory

Exceptionally, RISC OS allows JPEG images to be displayed on screen (**but not printed**) directly from their file i.e. without the need to first carry out the common measuring and loading sequence.

Dr Wimp supports this option and the wimp-functions are:

```
PROCwimp_renderjpegfile(jpegfilepath$,bx%,by%,
            minx%,miny%,maxx%,maxy%,
            scalex,scaley)

PROCwimp_renderwindowjpegfile(window%,
            jpegfilepath$,bx%,by%,minx%,miny%,
            maxx%,maxy%,scalex,scaley)
```

As you can see, the JPEG handle is simply replaced by the full path name of the JPEG file and hence no further explanation is needed.

Complementary wimp-functions

Complementary to the above suites of functions for displaying/printing images are several wimp-functions offering further help with image file manipulation.

Sprites and sprite-files

Because a sprite-file can hold more than one sprite image, there is a wimp-function to count the number of sprites in a sprite-file which has already been loaded into memory. It is:

```
FNwimp_countsprites(sprite%)
```

which will return the number of sprites in the user sprite area starting at `sprite%` - remembering that if you have loaded more than one sprite-file, then you need to specify the handle which refers only to the sprite-file you want. For example, in our three sprite-file example earlier, the three handles were `A%`, `B%` and `C%` - and they were loaded contiguously into one memory block starting at `SpriteArea%`.

So you would need to use either `A%` or `B%` or `C%` for `sprite%` when using `FNwimp_countsprites(sprite%)`.

Associated with the above is:

```
FNwimp_getspritename(sprite%,position%)
```

which will return the name of the loaded sprite at position `position%` in the user sprite area defined by `sprite%`. Position 1 is the first sprite etc. - so it is usually important to check how many exist before seeking a name.

More directly linked with the drawing/printing actions is:

```
FNwimp_getspritesize(sprite$,sprite%,side%)
```

which returns the width/height of a loaded sprite (in OS units), `sprite$` is the sprite name and `sprite%` is its user sprite area handle. `side%` is 0 for the width and 1 for the height.

This, for example, can be used to check the physical size of a sprite before displaying it - to set the position accurately or resize the window etc.

Drawfiles

In this case, as there is only one image per drawfile, the only extra wimp-function is:

```
FNwimp_getdfilesize(dfile%,side%)
```

which returns the width/height (in OS units) of the overall bounding box surrounding all the ‘objects’ in the drawfile loaded into the memory block starting at `dfile%`.

JPEGs

Again, the only extra facility is to find the size of a JPEG image - but this can be done either from the loaded image file or directly from the file itself. The two wimp-functions are:

```
FNwimp_getjpegsize(jpeghandle%,side%)
FNwimp_getjpegsizefile(jpegfilepath$,side%)
```

No further explanation should now be necessary.

Specific examples

The Examples folder in the Dr Wimp package contains the applications `!ScaleDraw` and `!SprAreas` which show in detail how points in this chapter are implemented in practice.

Changing the pointer sprite

This facility does not sit entirely comfortably in this section but it does concern sprites which can be in a user sprite area. So it is as well to cover it now.

Using the validation string of an icon (see Appendix 5) it is easy to change the pointer to another shape as it enters and leaves an icon. However, Dr Wimp goes one better and provides a wimp-function to enable the mouse pointer sprite to be changed more generally from its default shape (and back!).

The call is:

```
PROCwimp_pointer(pointer%,area%,pointer$)
```

where **`pointer%`** can be 0 (the default pointer) or 1 (a user-defined pointer)

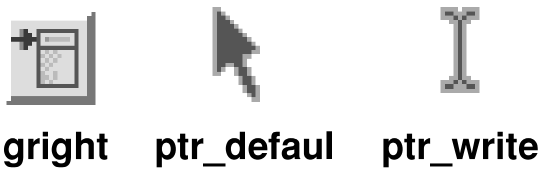
`area%` is a handle to a user sprite area (as defined earlier in this chapter) or 0 for the wimp sprite pool.

`pointer$` is the name of the required sprite, which can be a null string if the default pointer is to be used (i.e. if **`pointer%=0`** and **`area%=0`**).

This is a fairly flexible function in that the values of the first two parameters can be set independently and it will work - provided you have the required sprites stored in the right places. The only (sensible) rigidity is that if **`pointer%`** is set to 0 (i.e. calling for the default pointer) then the pointer name is automatically set to **`"ptr_default"`** (the name of the default pointer supplied by RISCOS) irrespective of the contents of the third parameter.

A typical use for this facility is to change the pointer as it enters and leaves a window. The Dr Wimp Manual shows how easy it is to do this, using the pair of user-functions **`PROCuser_enteringwindow`** and **`PROCuser_leavingwindow`**.

A further thing to note is that sprites used for pointers are normally defined in screen mode 8 and 19 (both 4-colour modes) and attempts to use other sprites may not work properly. Below are screenshots of three different 4-colour pointers held in the Wimp pool for use by any application - together with their names.



20. Handling text

Dr Wimp offers a very wide range of wimp-functions for putting text (including outline fonts and colour) into windows or directly onto the screen. *(Remember that putting text into icons is a separate - and easy - issue, partially controlled by the icon validation string and some basic wimp-functions covered early in the tutorial chapters.)*

When outline fonts are used, the wimp-functions allow the font choice to be entered as a normal font string or as a font handle.

Control codes can be inserted within a text string to change font effects letter by letter if need be.

Further, the physical size of the displayed/printed text can be found, thus allowing accurate positioning for various purposes e.g. tabulation.

It is easiest to introduce the various features by starting with a common example.

```
PROCwimp_plottext(t$,f$,s%,x%,y%,fr%,fg%,fb%,
            br%,bg%,bb%)
```

The string to plot is **`t$`** and the font to use is placed in **`f$`** in the common ‘period separated’ string form e.g. **`"Homerton.Bold.Oblique"`**.

`s%` is the required font size in points and **`x%`** and **`y%`** are the screen coordinates (in OS units) where the text is to be plotted (the bottom left corner of the ‘bounding box’ surrounding the text, to be more accurate).

The final six parameters are the foreground and background colours of the text in the common red/green/blue amounts i.e. each can take a value from 0-255 with 0,0,0 being Black and 255,255,255 being White. (The background colour is for anti-aliasing rather than any overt visible presence.)

The above wimp-function is directly comparable in its action with its image counterparts e.g. `PROCwimp_render` etc. - except that no ‘clipping rectangle’ is involved.

Similarly, for text plotting in a window there is:

```
PROCwimp_plotwindowtext(window%,t$,f$,s%,x%,y%,
          fr%,fg%,fb%,br%,bg%,bb%,
          minx%,miny%,maxx%,maxy%)
```

which does include a ‘clipping rectangle’ and equates exactly with its image counterparts e.g. `PROCwimp_renderwindow` etc. - the `x%` and `y%` values now needing to be in work area coordinates and the extra four parameters being the clipping rectangle again.

Font handles

As we have already seen, fonts can also be defined with handles, and this is a more sensible way to do things because the Wimp itself always uses font handles anyway and more efficient coding results behind the scenes.

For example, the call:

```
FontHandle%=FNwimp_getfont("Trinity.Medium",12)
```

will return a handle for the `"Trinity.Medium"` font at 12-point size i.e. one handle applies to a font/size pairing.

Note that the font specified must be available in your central font resources - otherwise a 0 will be returned. This can be used to give a warning to a user and perhaps substitute another one which is available.

It would be normal to get these font handles within `PROCuserinitialise`.

With the handles available, the corresponding set of wimp-functions for text plotting are:

```
PROCwimp_plottexth(t$,font%,x%,y%,fr%,fg%,fb%,
          br%,bg%,bb%)
PROCwimp_plotwindowtexth(window%,t$,font%,x%,y%,
          fr%,fg%,fb%,br%,bg%,bb%,
          minx%,miny%,maxx%,maxy%)
```

which merely add an "h" to the name and substitute a font handle `font%` for the previous font string and point-size parameters.

A point to note when using font handles is that they are very much like file handles. Thus getting a font handle is more akin to 'opening' a channel to the central font resource files (and don't forget that you are very likely to have several font handles 'open' at the same time). Accordingly, font handles ought to be 'closed' as soon as you have finished with them within the application and Dr Wimp provides a simple call for this:

```
PROCwimp_losefont(FontHandle%)
```

will 'lose' (close) the font handle specified by `FontHandle%`.

Dr Wimp does, however, provide a 'fail safe' facility in that, on quitting the application properly, it will automatically close any fonts that have been opened using `FNwimp_getfont` *or contained in any window template loaded by the application.*

Font changing within a string

So far, we have simply said that `t$` is the string to plot - but Dr Wimp offers further functions to modify this string if we wish. For instance:

```
FNwimp_underline(on%)
FNwimpfontchangeh(font%)
FNwimp_fontcolour(fr%,fg%,fb%)
```

are all available for modifying things within a string.

An example of using all these calls might be:

```
FontHandleA%=FNwimp_getfont("Trinity.Medium",16)
FontHandleB%=FNwimp_getfont("Homerton.Bold", 24)
String$="This is a "+FNwimp_underline(1)+"test"+
            FNwimp_underline(0)+" string to
            demonstrate changing the
            "+FNwimp_fontcolour(255,0,0)+
            "colour"+FNwimp_fontcolour(0,0,0)+"
            and also the
            "+FNwimp_fontchangeh(FontHandleB%)+"
            font"
PROCwimp_plotwindowtexth(window%,String$,
            FontHandleA%,50,150,0,0,0,255,255,
            255,minx%,miny%,maxx%,maxy%)
PROCwimp_losefont(FontHandleA%)
PROCwimp_losefont(FontHandleB%)
```

which will produce the following displayed text, which has one word underlined; the word "colour" in red (on the screen, anyway!) and a font change for the last word:

This is a test string to demonstrate changing the colour and also the **font**

These effects apply only to the one string, so the next string plotted will start in the default state.

Text positioning

With these possibilities it could obviously be a problem placing/aligning text accurately. So Dr Wimp provides the following wimp-functions to tell you the text length and height prior to display:

```
FNwimp_gettextsize(text$,font$,size%,side%)
FNwimp_gettextsizeh(text$,font%,side%)
```

The first is for use with a font string/point-size combination and the second with a font handle. In both cases, if `side%=0` the text width (physical length) in OS units is returned and if `side%=1` the height is returned.

You should note that if you want the width (length) of a string which has a font change within it - as in the above diagram - these two functions do still return the correct width. However they do not always return the correct height in such cases. Fortunately, it is the text length which is usually the one needed.

'Desktop font'

With modern RISCOS computers the user can choose which font to use for his/her normal desktop display - which, of course, will be the font appearing by default in icons also.

It is therefore sometimes helpful to be able to ensure that directly plotted text appears in exactly the same font as the desktop - and two wimp-functions are provided for this. They are:

```
PROCwimp_deskplottext(t$,c%,x%,y%,fr%,fg%,fb%,
            br%,bg%,bb%)
PROCwimp_deskplotwindowtext(window%,t$,c%,x%,y%,
            fr%,fg%,fb%,br%,bg%,bb%,
            minx%,miny%,maxx%,maxy%)
```

With one exception, the parameters are as for their previous counterparts. The exception is the parameter `c%`, which determines the justification. If it is 0 then the text is left-justified at `x%` - and if it is 1 the text **is horizontally centred around `x%`**.

If your computer can only use 'system font' - or you have chosen to configure it to use that - the above two wimp-functions will correctly produce their output in 'system font' also.

Don't forget that if you are printing text which is plotted with the wimp-functions in this chapter then - to cater for PostScript printers - you need to put the appropriate 'font declarations' inside **`PROCuser_declarefonts`** - see Chapter 14.

Specific example

The **`Examples`** folder in the Dr Wimp package contains the application **`!PrintTest`** which shows in detail how points in this chapter can be implemented in practice for a large text document.

21. More on ‘user’ printing

In our tutorial application a typical printing task was tackled using the ‘redraw printing’ method and this chapter will complete the picture by having a look at the ‘user printing’ method.

You will recall that ‘redraw printing’ essentially provides a means of printing what is displayed on the screen (‘Wysiwyg’). In contrast, ‘user printing“ allows you to print something independent of the screen display.

In some ways ‘user printing’ is a little easier to implement simply because - as there is no display to copy - there is no conversion between display coordinates and paper coordinates. We can work solely in paper coordinates directly i.e. OS units referred to the bottom left corner of the paper.

Having said this, the same preliminaries need to be taken care of. That is:

Check that a printer driver is installed;

Check paper size and borders;

Declare all fonts to be used, including any in rendered drawfiles (for Postscript printer drivers).

As before, printing is initiated by calling:

```
PROCwimp_print(user%,window%,fpage%,lpage%,perpa-
          ge%,copies%,orient%)
```

but this time `user%` is set to 1 and `window%` can be any value because it will be ignored. (But don’t set it to an undefined variable!)

This (after a great deal of detailed preparation hidden behind the scenes) will, in turn, call:

```
    PROCuser_print(minx%,miny%,maxx%,maxy%,page%)
```

(Not to be confused with `FNuser_printing`*, which again is available to check on the printing progress.)*

The first four coordinates are the now-familiar clipping rectangle, this time supplied in paper coordinates. The clipping rectangle is generally of less interest for 'user printing' but is supplied anyway.

`page%` is passed through from the Wimp and is the current page being printed, which would typically be used by the programmer to determine exactly what is to be printed on which page.

The only minor complication is having to specify printing positions with respect to the bottom left corner. It does make sense - but it needs concentration when, say, thinking of lines of text from the top of the page.

The printing borders obviously need to be accounted for - particularly with text. It is easy to lose a line of text at the top or bottom of a printed page.

Also don't forget that the Dr Wimp text plotting commands use the bottom left corner of the text string bounding box for positioning - so the top text line on a page needs to be positioned below the top margin by at least the text height.

Printing sprites, drawfiles and JPEGs

Printing images is straightforward once the images have been loaded into memory - as was described in detail in Chapter 19.

After that, the use of `PROCwimp_rendersprite` or `PROCwimp_render` or `PROCwimp_renderjpeg` as appropriate - using paper coordinates (same as screen coordinates) - will print the image with its bottom left corner at the `x%`, `y%` position specified.

For obvious reasons, it does not make a great deal of sense to use the 'window' versions of these two functions when 'user printing'.

Finally, in case a PostScript printer is used, don't forget to 'declare' any drawfile fonts as described in Chapters 14 and 19.

Length to OS conversion

We have already met **`FNwimp_lengthtoOS`** in the tutorial application and in 'user printing' it is frequently used to position text/sprites/drawfiles accurately on the paper - either in metric or imperial measurements.

For example, within **`PROCuser_print`**, the sequence:

```
OneInch%=FNwimp_lengthtoOS(1,100,1)
RECTANGLE FILL OneInch%,OneInch%,OneInch%
```

will produce a filled square of 1 inch side, located 1 inch from the left and bottom edges of the paper.

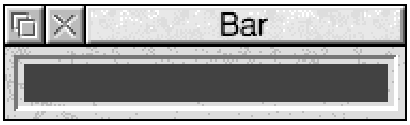
22. Sliders and bars

Many applications use a sliding bar to indicate the progress of a task e.g. when formatting a disc - and applications sometimes require the user to drag a sliding bar in order to set values e.g. the colour picker. Dr Wimp calls the first usage a ‘bar’ and the second usage a ‘slider’. That is, a bar is essentially an output device and a slider is both an input and output device.

Dr Wimp offers a set of wimp- and user-functions to make the management of bars/sliders very easy.

Bars

These are straightforward. All that is needed is a small window with two icons in it: one icon being filled with a colour to make it become the coloured bar itself and the other being a cosmetic frame around it, like this:



What Dr Wimp does is to alter (in one direction, decided by the programmer) the size of the coloured icon within defined limits, to produce the sliding bar effect.

The coloured icon needs to be of the ‘Click/Drag’ button type. The length and depth of the bar icon can be anything you wish, but you do need to know its icon number and required maximum length of it i.e. the ‘size’ of the bar icon. This can be found and adjusted via **`!TemplEd`**, see Appendix 7.

The size of the bar icon can then be changed with:

```
PROCwimp_bar(window%,icon%,length%,dir%)
```

where **`window%`** and **`icon%`** are the window/icon pair forming the bar, **`length%`** is the bar length required at the time and **`dir%`** determines whether the bar is to ‘move’ horizontally (as here) or vertically. If horizontally then **`dir%=0`**. If **`dir%=1`**, the bar height would change, keeping the width constant. This allows you to use vertical as well as horizontal bars.

Behind the scenes, Dr Wimp deletes the bar icon and re-creates a replacement one of the required new size - updating the display straightaway.

So, you simply call **`PROCwimp_bar`** with whatever bar length you wish - but it is down to you to ensure that it does not exceed its designed maximum. (Nothing drastic will happen if you don’t - the bar will simply overlap its frame at the right end and extend, maybe invisibly, outside its window and thus spoiling the effect.)

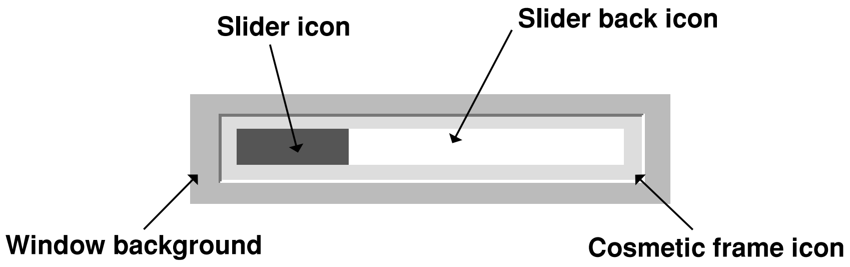
A good way to minimise the chances of this happening is to set the bar length as a percentage of its intended maximum value and ensure that this never exceeds 100%.

As a bar is often used to show progress of a task which is taking time, it is often useful to trigger a new **`PROCwimp_bar`** call periodically via an ‘internal multitasking’ process and the Dr Wimp Manual shows how to do this.

Sliders

Dr Wimp implements sliders in almost as simple a way. The vital point is that - in addition to the surrounding frame icon - the slider itself must now be made up from two icons - of contrasting colours: the ‘slider’ icon itself and the ‘slider back icon’ (both of the ‘Click/ Drag* button type).

The ‘slider back icon’ remains a constant size and forms the background to the slider and defines the area over which the ‘slider icon’ can be dragged. The following figure shows the relationship for a horizontal slider.



In this case, the width of the ‘slider back icon’ defines the maximum width that the ‘slider icon’ can be dragged over, and the height of the slider and back icons are made to be exactly the same. Vertical sliders are constructed in a corresponding way - and note that if the icons are made higher than their width Dr Wimp will automatically assume that it is a vertical slider. Luxury!

The sole purpose of the back icon is to register the presence of the pointer in the draggable area. So it is essential that Dr Wimp knows that the two icons are a pair - and the first action needed is to tell Dr Wimp precisely that, using **`FNuser_slider`** and **`FNuser_sliderback`** as follows:

```
DEF FNuser_sliderback(window%,icon%)
Return%= -1
CASE window% OF
       WHEN SliderWindow%
       IF icon%=SliderIcon% THEN
              Return%=SliderBackIcon%
ENDCASE
=Return%

DEF FNuser_slider(window%,icon%)
Return%= -1
```

```
CASE window% OF
       WHEN SliderWindow%
       IF icon%=SliderBackIcon% THEN
              Return%=SliderIcon%
ENDCASE
=Return%
```

Note the symmetry of these two sequences. The two user-functions are called from the `DrWimp` library every time a mouse-click (or drag) occurs.

If the user-functions return an icon handle/number - rather than -1 (their default return) - then `DrWimp` treats the returned icons as a slider pair and takes action accordingly. (A slider window - called `SliderWindow%` here - has to be loaded and opened, of course.)

Merely by entering a sequence like the above, the slider will correctly operate: either by dragging or by clicking in the slider area. You are saved all the hassle.

There are then three other complementary functions to enable you to harness the slider usefully. The first is:

```
PROCuser_slidervalue(window%,slider%,pcent%)
```

which continuously reports the slider percentage value (of the slider icon `slider%`) whilst dragging is happening. (This facilitates ‘live’ updating of a number read-out, for instance.)

The second is:

```
FNwimp_getsliderpcent(window%,slider%)
```

which returns the current slider percentage value - in the range 0-100 and which can be a real number.

And the third is:

```
FNwimp_putsliderpcent(window%,slider%,pcent)
```

which allows a slider value to be changed to the specified value - which again can be any real number in the range 0-100.

The 0-100 percentage values can easily be scaled for any range you wish - and don't forget that you will need to use `STR$` to convert the returned value into a string for displaying it in an associated icon, for instance.

Specific examples

The `Examples` folder in the Dr Wimp package contains the applications `!Bar` and `!Slider` which show in detail how bars and sliders are implemented in practice.

23. Panes

Panes are windows which are linked with another window so that they seem to be permanently attached together and move as one when dragged.

Typical examples of panes are:

the toolbox window attached to a !Draw window.

toolbars within a word-processor window.

In fact, panes are separate windows, designed and carefully placed with respect to their parent window and having their opening and closing actions carefully linked.

As usual Dr Wimp makes their management very easy - with the help of `FNuser_pane` which is used in concert with `PROCuser_openwindow` and `PROCuser_closewindow`.

These latter two user-functions are called by the `DrWimp` library every time a window has just been opened or closed (respectively) and they pass to the programmer the window handle involved. (For `PROCuser_openwindow`, the window position and stack position are also passed.)

It is therefore straightforward to trigger the opening/closing of another window - e.g. the pane - within these user-functions, and if the opening is carried out by using `PROCwimp_openwindowat` the positioning of the pane can be accurate.

The role of **`FNuser_pane`** is to keep the **`DrWimp`** library aware of the window/pane relationships so that the pane always appears to 'sit on top of its parent window *(that is, in relation to their displayed stack positions: their x/y positions are independent and controlled by you, the programmer)*.

This is done with a simple routine such as:

```
DEF FNuserpane
Return%=-1
IF window%=Main% THEN
Return%=Pane%
ENDIF
=Return%
```

This tells the DrWimp library that the parent window, with the handle **`Main%`**, has a pane whose handle is **`Pane%`**.

The only thing that needs to be watched is when a parent window has more than one pane. In this case you need only to decide their required 'stacking order' and in **no circumstances should you try to attach a pane to a pane**.

The Dr Wimp Manual covers this whole subject very comprehensively.

Specific examples

The **`Examples`** folder in the Dr Wimp package contains the applications **`!MultiPane`**, **`!PanePain`** and **`!Toolbar`** which show in detail how panes are implemented in practice.

24. The ‘NULL%’ global variable and ‘internal multi-tasking’

Very early on in this book the Dr Wimp global variable **`NULL%`** was mentioned, at that time only to register that it was one of few global variables used by Dr Wimp which does not follow the normal pattern of having a name starting with a lower-case “w”.

`NULL%` is always initially set to **`FALSE`** by the **`DrWimp`** library in the Wimp initialisation action.

It is then checked by the **`DrWimp`** library prior to each call made to **`SYS "Wimp_Poll"`** within **`PROCwimp_poll`**. If it is still **`FALSE`** at this point then there is no reason to use Reason Code 0 and it is duly ‘masked out’ of the poll.

If **`NULL%`** is **`TRUE`** at this point then Reason Code 0 is not masked out and every time it occurs (which is very often indeed) another internal function is brought into play - and this, in turn, calls **`PROCuser_null`**.

Thus, if **`NULL%`** remains false nothing much happens, but if at any time the programmer has set **`NULL%`** to true then **`PROCuser_null`** is called each time Reason Code 0 is received - until **`NULL%`** is again set to false.

The programmer can therefore then use **`PROCuser_null`** to carry out very quick background tasks while nothing else is going on i.e. ‘internal multi-tasking’.

Bearing in mind that Reason Code 0 occurs very many times, `PROCuser_null` needs to be used with care if the application is not to be slowed down considerably. Nonetheless, the facility is there if you need it.

However, Dr Wimp also offers some useful alternatives with:

```
PROCwimp_singlepoll
PROCwimp_pollidle
PROCwimp_singlepollidle
```

PROCwimp_singlepoll

As its name indicates, this wimp-function makes a call to the Wimp Poll just once - at whatever place you choose to put it in your program (which, of course, will almost certainly already be running inside the main Wimp Poll loop).

The effect is to provide multi-tasking within the application.

Try it by putting the following routine temporarily into (a copy of) the `!Fuel16a !RunImage`.

```
2742 WHEN 1
2744 FOR N%=1 TO 300
2746 PROCwimp_singlepoll
2748 a$=STR$(N%)
2752 PROCwimp_puticontext(Info%,4,a$)
2754 NEXT
```

Now save it and run the application. Press `<menu>` to display the iconbar menu, then trigger the action by pressing `<select>` on the first menu item - ‘Info’. Then bring up the menu again and slide right across ‘Info’ to show the `Info` window. You will see the text in the top icon changing as it counts up to 300. It does not stop other actions being taken.

PROCwimp_pollidle(duration,seconds%)

This is probably of more specialised use because it is used instead of the usual `PROCwimp_poll` and probably not in a loop.

What it does is to call the Wimp Poll once every interval set by the value of `duration`. (If `seconds%` is set to 1 then the value of duration will be interpreted as seconds. If `seconds%` is 0 then duration will be interpreted as centi-seconds.)

So, for instance, by also setting `NULL%=TRUE` a call to `PROCuser_null` would then only occur once every interval of duration.

PROCwimp_singlepollidle(duration,seconds%)

This acts similarly to `PROCwimp_pollidle` except that it calls the Wimp Poll once only - after a delay of the value of `duration`.

These wimp-functions are undoubtedly a bit specialised and need to be used with care.

The Dr Wimp `Examples` directory contains four example applications using them. They are: `!Animate`, `!Clocker`, `!FastSlow` and `!SlowFast`.

25. Colour picker

If you have a RISCOS Version from 3.50 onwards you will be familiar with the Colour Picker window, from applications such as !Draw and !Paint.

The Colour Picker is a special window which enables the application user to choose, visually, what colour is required - from the complete range of colours available. A colour can be specified by clicking on the colour actually displayed in the window, or by specifying the proportions of the constituent colour components.

(These colour components can be in RGB, CMYK or HSV terms: these are known as the 'colour models' and the Dr Wimp manual covers them comprehensively).

With Dr Wimp you can get the Colour Picker window either by opening it exactly like any other window, or by opening it as a 'submenu' from a menu item.

'Dialogue type'

If you open the colour picker window as a 'sub-menu' it will, as you would expect, close automatically like any other sub-menu if you move the pointer back over the menu item from which it came - and if you click elsewhere on the screen. This is known as the 'sub-menu dialogue type'.

If you open the colour picker window as an ordinary window, Dr Wimp also allows you to have the choice of closing it in the normal way (e.g. by clicking on its Close icon) or closing like a menu i.e. closing when a click is made elsewhere.

In all cases, the window will close automatically when you use `<select>` over the colour picker window's 'OK' or 'None' buttons (but using `<adjust>` will keep it open as usual).

The Colour picker as a normal window

This is the easiest method and there are two wimp-functions available - but only one will be introduced here. It is:

```
     PROCwimp_opencolourpickerrgb(dialoguetype%,
                 red%,green%,blue%,none%,x%,y%)
```

This can be regarded as the basic operation: it opens the colour- picker window in the RGB model and the initial colour is specified in **`red%`**, **`green%`** and **`blue%`** using colour component values in the familiar range 0-255.

If **`dialoguetype%`** is set to 0 the window will stay open until the user presses its Close icon or makes a colour selection by pressing 'OK' or 'None'. If it is set to 1 the window will act like a menu and close if the mouse is clicked outside the window (as well as pressing 'OK' or 'None' of course).

`none%` chooses whether the 'None' button is available or not and whether it is selected when the colourpicker window opens. A value of 0 means the 'None' button will not be available for use; 1 means it will be available and initially de-selected; and 2 means it will available and initially selected.

`x%/y%` are simply the required screen OS coordinates for the top left corner of the opening colour picker window.

This wimp-function is called in similar circumstances to using **`PROCwimp_openwindow`** and is perfectly straightforward in practice.

The Colour picker as a ‘sub-menu’

Opening the colour picker as a ‘sub-menu’ involves two steps.

A special global variable called `wSUBMENUCOLOURPICKER%` *(a mouthful, but you won’t easily forget it!)* has been created in the `DrWimp` library to act, in this case, as the colour picker window handle - and is set to 1 as default.

If you want the colour picker window to appear as a sub-menu to an (already defined) parent menu item, then the first step is to ‘attach’ the above global variable to that item as its sub-menu/window handle, e.g.:

```
PROCwimp_attachsubmenu(parentmenu%,3,
              wSUBMENUCOLOURPICKER%)
```

This action ensures that the required item (here, 3) on the parent menu (whose handle here is `parentmenu%`) correctly displays a submenu ‘arrowhead’ - and that is all it does at the moment.

The second step is to decide which of two wimp-functions you wish to use to actually cause the window to open (when you move across the menu arrowhead) and put it into `PROCuser_overmenuarrow`.

As before, we will show only the simplest of the two wimp-functions here. It is:

```
PROCwimp_opensubmenucolourpickerrgb(red%,green%,
             blue%,none%,x%,y%)
```

As you can see, this matches the previously-described wimp-function for opening the colour picker window normally - except that there is no `dialoguetype%` parameter because, in this case, the colour picker window will always act exactly like a sub-menu i.e. it will use the ‘sub-menu dialogue’.

A typical coding might therefore be:

```
DEF PROCuser_overmenuarrow(nextsubmenu%,
            parentmenuitem%,x%,y%)
CASE nextsubmenu% OF
      WHEN wSUBMENUCOLOURPICKER%
        PROCwimp_opensubmenucolour-pickerrgb
           (255,187,0,0,x%,y%)
ENDCASE
ENDPROC
```

This would cause the colour picker window to open when you move across its parent menu's arrowhead - and, here, it would open with the RGB model showing a pale straw yellow and without a 'None' button. It will open in the same place as a conventional sub-menu, because the `x%`/`y%` values have been passed directly from the user-function in which it sits.

Making the colour choice

Once the user has decided which colour is wanted from the displayed colour picker window, he/she presses the window's 'OK' button (or 'None' button). Therefore you, the programmer, need a means of extracting the user's choice from the colour picker window and into your program for further use.

Dr Wimp provides two user-functions for this purpose. One provides the chosen colour output in 0-255 'rgb' values and the other in the values of the 'model' actually selected in the colour picker window at the time the selection was made.

> *Note that outputs are always made automatically by Dr Wimp into* ***both*** *of these user-functions when the user makes the colour choice - irrespective of the actual configuration of the colour picker window at that time.*

To match our previous descriptions we will look at only the simplest of the two user-functions here. It is:

```
PROCuser_colourpickerrgb(red%,green%,blue%,none%)
```

As already said, this is called automatically every time the user makes a colour selection in the colour picker window i.e. when the user presses ‘OK’ or ‘None’.

If ‘OK’ was pressed, then the user-function will pass the values (0-255) of the chosen colour in `red%`, `blue%` and `green%` - and `none%` will be set to 0.

If ‘None’ was pressed, then the currently displayed colour values in the colour picker window will still be passed, but `none%` will be set to 1. (So it is up to you to decide how to react to `none%` having a value of 1. For example, you might want to ignore the colour data in this case, or you might want to store the values in order to open the window with these values next time.)

Irrespective of the method you chose to open the colour picker window and irrespective of the actual ‘model’ currently selected in the window at the time you made your colour choice, `PROCusercolourpickerrgb` will always give the values `red%`, `green%` and `blue%` in the range 0-255, which is often used in other wimp-functions and can therefore be conveniently passed on directly.

The Dr Wimp Manual gives a full explanation of all the colour picker options - in particular, a description of the ‘colour models’ and the wimp-functions and user-function provided for using them all i.e. the functions not covered in this chapter.

Specific example

In the Examples folder of the Dr Wimp package there is an application called `!ColPick` which gives a practical demonstration of how to use the above-described facilities.

26. Important common facilities

Dr Wimp includes a number of facilities which are likely to appear in any full application and some of which are designed to help you comply with the RISCOS Style Guide painlessly.

Writable icons and the keyboard

Because of the particular nature of keyboard operations the Wimp has to be helped to know whether a particular keypress is going to be used by your application or is intended for another. For this reason, all Wimp applications must have a 'keyboard handler' in place. Dr Wimp provides this via `FNuser_keypress`.

Further, for writable icons, the effect of pressing one of the keyboard keys when an icon has the caret is controlled in RISCOS by the contents of the icon's validation string (see Appendix 5) - in particular, the A-command and the K-command.

The 'keyboard handler' and the validation string work in harmony: the A-command controls which keyboard characters are allowed (or not allowed) to be typed into a writable icon - and the K-command determines which keypresses are passed to Dr Wimp for use in `FNuser_keypress` and also - in later RISCOS versions - how the caret is moved among different writable icons in the same window.

In our tutorial we only needed the application to take special action if the `<return>` key was pressed when the caret was in certain icons. (You will recall that this keypress initiated some validation tests.) However, many applications will want to do more than that.

The definition of this user-function in its default (‘empty’) state is:

```
DEF FNuser_keypress(window%,icon%,key%)
=0
```

and - as said above - it is called automatically by the `DrWimp` library if one of your windows has the input focus and the keypress has been passed to your application by virtue of its validation string K-command setting.

If you do take action on a specific keypress in `FNuser_keypress` then you should change the return from the function to 1 (as we did in the tutorial) when you have done so, instead of returning its default value of 0. A return of 1 tells the Wimp that you have dealt with the particular keypress and it will then not try to pass it on to other applications.

Further aspects on this topic are best left to Appendix 5.

Errors

Dr Wimp provides two standard error facilities plus the means for making ‘custom-built’ versions.

The first is:

```
PROCwimp_error(title$,error$,button%,prefix%)
```

which brings up the standard error box, with the first two parameters specifying your own choice of window title and error message.

There is the choice of having either an ‘OK’ button or a ‘Cancel’ button, by making the third parameter 1 or 2 respectively. The title can also be prefixed with `'Error from'` or `'Message from'` by setting the final parameter to 1 or 2 respectively.

Clicking on the displayed button simply closes the window and it is left to the programmer to arrange for any necessary consequential action to take place.

This wimp-function is probably the most commonly-used Dr Wimp error facility. As the above implies, it can be used for helpful warning messages as well as for real errors - and the skeleton Dr Wimp `!RunImage` itself contains a typical global real error call near its very start. (This appears as Line 160 in the tutorial application.)

By and large, warnings to the user are not ‘fatal’ i.e. the program does not quit after reporting the problem - whereas genuine programming errors are normally ‘fatal’.

The second standard option provided is:

```
FNwimp_errorchoice(title$,error$,prefix%)
```

which is a **FN** rather than a **PROC** and always displays a standard error box with both ‘OK’ and ‘Cancel’ buttons.

But this time the wimp-function returns **TRUE** if ‘OK’ is pressed by the user and **FALSE** if ‘Cancel’ is pressed.

This wimp_function is often used to get the user to confirm a certain step before taking it, with the **TRUE**/**FALSE** return deciding the subsequent action path.

Custom error boxes

To allow you to design error boxes of your own, Dr Wimp uses the already introduced user-functions:

```
PROCuser_openwindow
PROCuser_closewindow
```

together with two new wimp-functions:

```
PROCwimp_bindpointer(window%)
PROCwimp_releasepointer
```

The procedure is that you design your error window as you wish - with as many action buttons as you like - and then load it as usual and cause it to open when needed. Let's assume the window has a handle called `ErrorWindow%`.

Then, within `PROCuser_openwindow`, a sequence such as:

```
CASE window% OF
       WHEN ErrorWindow%
       PROCwimp_bindpointer(window%)
ENDCASE
```

is added - and, within PROCuser_closewindow, the complementary:

```
CASE window% OF
       WHEN ErrorWindow%
       PROCwimp_releasepointer
ENDCASE
```

completes the picture.

This will prevent the pointer going outside your custom error box until the user has taken one of the choices offered in it.

It is therefore very important that - whatever else you cause to happen - a `PROCwimp closewindow` call is made when the user makes a choice. Otherwise there is no way out!

The Dr Wimp Manual has an example of this facility within its own tutorial.

Messages

(Not to be confused with the Wimp Messaging system introduced in Appendix 10!)

Many applications nowadays use Messages files to hold the various items of text that are used in the application e.g. the different error messages, or for preparing menus (as we have already seen in Chapter 17). It makes the editing of such text much more convenient, particularly where the same message is used by different parts of an application. It also means that such messages can be translated into other languages by someone who does not need to know anything about programming.

RISCOS (from version 3.00 onwards) provides `SYS` calls to handle Messages files - including the ability to insert strings into messages when they are called (rather like passing parameters in star commands).

As usual, Dr Wimp makes things a lot easier by using what is now a fairly familiar method. The messages are prepared and stored in a textfile in a special (but simple) format. This file is identified to Dr Wimp which, in turn, sets up memory blocks ready to read the file. Wimp-functions are provided to manage the process.

The first step is to prepare the Messages textfile. This is straightforward: each text message is entered as required but is preceded with a ‘token’ which is separated from the message text by a colon. A typical example of part of a Messages file might be:

```
#Error messages
ErrorA:The date needs to be in dd/mm/yyyy format
ErrorB:The mileage entry must be higher than last
            time. Please correct.

#Miscellaneous
Ver:version 2.02 (16th May 1998)
```

In this simple example, `"ErrorA"`, `"ErrorB"` and `"Ver"` are tokens, which essentially means that we can retrieve the message for use simply by referring to its token.

It is important to ensure that <Return> is pressed at the end of the last entry of the Messages file - otherwise the last entry will not be recognised.

Once the Messages file exists, a call to:

```
FNwimp_initmessages(path$)
```

does all the necessary setting up, where `path$` is the full file path of the file. The return from this `FN` is the handle of the Messages file, which is used in other wimp-functions to read the file etc.

Whenever a message text is to be retrieved from the Messages file, a call to one of three similar wimp-functions then does the trick. These are:

```
FNwimp_messlook0(messagefile%,token$)
FNwimp_messlook1(messagefile%,token$,a$)
FNwimp_messlook2(messagefile%,toke$,a$,b$)
```

`FNwimp_messlook0(messagefile%,token$)` returns the corresponding message text untouched. Thus:

```
PROCwimp_error(YourAppName$,
          FNwimp_messlook0(messagefile%,
          "ErrorA"),1,2)
```

would bring up an error box with `"The date needs to be in dd/ mm/yyyy format"`, if the earlier example textfile was used.

The two other wimp-functions return the message text complete with one or two parameter text substitutions respectively - provided the message in the textfile has been set up accordingly.

For instance, if the previous textfile entry for `"ErrorB"` was changed to:

```
ErrorB:The mileage entry must be %0 last time,
          Please correct.
```

the call:

```
FNwimp_messlook1(messagefile%,"ErrorB",
          "lower than")
```

would produce the message with the text `"lower than"` substituted for `%0`.

Similarly, if the message textfile entry was changed to:

```
ErrorB:The %1 entry must be %0 last time. Please
          correct.
```

the call:

```
FNwimp_messlook2(messagefile%,"ErrorB","lower
          than","temperature")
```

would produce the message with the text `"lower than"` substituted for `%0` and `"temperature"` substituted for `%1` and would appear as:

```
"The temperature entry must be lower than last
          time. Please correct."
```

It's as simple as that, but in the last case note that the string parameters placed in `a$` and `b$` are substituted for `%0` and `%1` respectively - irrespective of the order they appear in the message (and even if they appear more than once).

Other points to note are:

If `FNwimp_messlook0` is used with a message containing `%0` or `%1` then no substitutions will occur and the returned text will merely print a null string where the `%0` (or `%1`) occurs.

If **`FNwimp_messlook1`** or **`FNwimp_messlook2`** is used with a message which does not contain any **`%0`** or **`%1`** then no error will occur. The message will be returned exactly as if **`FNwimp_messlook0`** had been issued.

Similarly, if **`FNwimp_messlook2`** is used with a message which contains only **`%0`** (or **`%1`**) then no error will occur. The message will be returned with just the possible substitution(s) in place.

Interestingly, in many cases it takes less memory to use messages instead of putting the text in the program lines. The message procedure takes extra space for the tokens, but each message string needs only to be stored once even though it can be used several times in the program.

You can also find the number of messages with a given token in an initiated Messages file - using:

```
FNwimp_getnumberofmessages(messagefile%,token$)
```

Re-initiating Messages files

There are several circumstances where it is helpful to be able to change a Messages file after it has been initiated and yet keep the same Messages file handle. For example, you may want to alter the file contents during the program run, or even substitute a completely different Messages file.

Dr Wimp allows you to do this very simply. All it takes is a call to:

```
FNwimp_reinitmessages(messagefile%,path$)
```

where **`messagefile%`** is the handle of the existing Messages file and path$ is the full path name of the new Messages file.

The return is the Messages file handle - which may well be different from what it was. So if you want to keep the same file handle variable you simply need to call the wimp-function like this:

```
messagefile%=FNwimp_reinitmessages(messagefile%,
          path$)
```

That is, re-assign the new handle to the old handle’s variable.

Interactive help

Dr Wimp supports RISCOS's interactive help system, activated via **`!Help`** or similar applications. It is extremely easy to implement using the two user-functions provided:

```
DEF FNuser_help(window%,icon%)
DEF FNuser_menuhelp(menu%,item%)
```

The first acts for windows and icons only and the second for menus. Their usage is very similar.

If RISCOS's **`!Help`** (or a similar application) is running, the DrWimp library calls these user-functions whenever the pointer is over a window/icon or menu belonging to your application, passing the window/icon or menu/item and icon handles.

All you have to do is arrange for the user-function to return the required text instead of the default null string.

A typical construction might be:

```
DEF FNuser_help(window%,icon%)
Help$=""
CASE window% OF
       WHEN NewCar%
          CASE icon% OF
             WHEN 0
             Help$="Enter a car registration here
              - of at least 6 characters"
       ENDCASE
ENDCASE
=Help$
```

You will then see your chosen text appear in the help display when the pointer is over icon 0 in the **`NewCar%`** window. *(In this example, the window and icon handles are those from the tutorial application in earlier chapters. The icon is the one in which the user has to enter the car registration when setting up a new vehicle file.)*

An entirely similar process is used with **`FNuser_menuhelp`**, using the menu handle and menu item number instead of the window/icon combination.

As you may well have realised, a convenient means of holding the help text is in a messages textfile as described above.

Dynamic areas

Dynamic areas provide a means, from within a program, of creating a block of memory which can be expanded or shrunk at will during the program run - limited only by the computer memory available. This is particularly useful, for instance, where the sizes of data files which might need to be loaded by the application are not known.

The Wimp only allows dynamic areas with RISCOS version 3.01 and higher - and it also uses a different method to implement them from version 3.50 onwards. Dr Wimp provides facilities to cover both situations with automatic detection of the RISCOS version.

The difference between the two methods is that, for RISCOS version 3.50 and above, the Wimp allows fully flexible dynamic areas to be created solely for an application - from the ‘Free’ application memory pool. When the application quits the dynamic area is returned to this pool. Dr Wimp calls thes types Application Dynamic Areas (ADAs).

However, for RISCOS versions from 3.01 onwards (but less than 3.50) a more restricted method is used, whereby an application is able to create a dynamic area in the Relocatable Module Area (the RMA) instead. This has the disadvantage that the Module area is a common resource for all applications, which can lead to wasted memory space caused by ‘fragmentation’ (a similar process to what happens if Basic string variables are not ‘sized’ before use).

Dr Wimp’s facilities handle either ADA or RMA transparently.

To create a dynamic area:

```
    FNwimp_createdynamic(size%,maxsize%,type%,drag%,n-
        ame$)
```

is used and returns the handle (start address) of the dynamic area.

`size%` is the required initial size of the memory block in bytes, It can be set to 0, but the Wimp may set a minimum size of 4kbytes anyway.

`maxsize%` is the maximum size to which the dynamic area can be enlarged to subsequently. It is important to try to set this to a realistic limit - and often `size%` and `maxsize%` can be made identical i.e. no increase needed. If you really do not want to limit the value then you can enter -1 here, which has the special meaning of ‘no limit’ - **but the use of this value is not recommended unless absolutely necessary** as it can lead to problems in dynamic area management.

`type%` is 0 for ADA or 1 for RMA, but note that if the RISCOS version is less than 3.50 then RMA will be used automatically, whatever the value of `type%`. *(By setting* `type%` *to 1 with version 3.50 or greater, the dynamic area will be forced into RMA. This allows you to check out your application for version 3.1 usage if you have a lot of dynamic area manipulation intended.)*

`drag%` can be 0 or 1, but is ignored unless `type%` is 0, in which case it allows you to alter the dynamic area size by dragging the corresponding bar in the Task Display (in the Dynamic areas section).

`name$` is the name that will appear in the Task Display if `type%` is 0. (Don’t confuse this with the dynamic area handle returned by the function and used to access/change the area.)

Having created the dynamic area it can be used to store whatever you wish - in exactly the same way as a block of bytes which has been created with `DIM`. But don’t forget to ensure its size is large enough for your specific purpose - preferably before each usage. For example, to save a sprite or drawfile to it, you would measure the sprite/drawfile first (see earlier chapters) and then change the dynamic area accordingly, before taking the save action.

Once created as above, a dynamic area can be changed in size by:

```
FNwimp_changedynamic(darea%,absolute%,size%)
```

where **darea%** is the handle of the dynamic area to be changed. *(As with re-initiating Messages files seen earlier, it would be normal to assign the return from the above to* **darea%** *again.)*

If **absolute%** is set to 1 then the value in **size%** will be interpreted as the required new absolute size of the dynamic area. If **absolute%** is set to 0 then the value in **size%** will be interpreted as relative i.e. the amount of memory to be added/subtracted to/from the current dynamic area size. (In this case, a minus sign is placed in front of the **size%** value if it is a reduction.)

The current size of a dynamic area created in this way can be measured with:

```
Size%=FNwimp_measuredynamic(darea%)
```

where **darea%** is the dynamic area handle.

Finally, as soon the dynamic area is no longer required it should be deleted, using:

```
PROCwimpdeletedynamic(Handle%)
```

As a 'fail safe' facility, on normal quitting (and on quitting due to most fatal' errors) Dr Wimp will automatically delete any dynamic areas created in the above way which have not already been deleted.

Quitting

Dr Wimp provides two functions to help us to quit applications in a controlled and orderly fashion. They are:

```
FNuser_quit
```

and:

```
PROCwimp_quit
```

(Don't mix them up, mentally!)

It needs to be remembered that an application may be required to quit as a result of:

- 'Quit' being chosen as a user option from within the application
- 'Quit' being chosen against the application in the Task Display
- Shutdown being chosen from the Task Manager menu

When any of these occurs, the DrWimp library causes:

```
DEF FNuser_quit(type%)
```

to be called, passing 0 in `type%` if it is a Quit action or 1 if it is a Task Manager Shutdown.

The default return from this user-function is 1, which simply means "continue with the quit/shutdown action".

However, if 0 is returned the quit/shutdown action is stopped. This paves the way to allow us to, say, save unsaved data before quitting. All that is needed is a flag of some type to be activated when unsaved data exists and for that flag to be tested within `DEF FNuser_quit`, such as:

```
DEF FNuser_quit
Quit%=1
IF UnSaved%=TRUE THEN Quit%=0
=Quit%
```

This example is, of course, very basic and only stops the quit/shutdown. A more useful sequence might therefore be to bring up the appropriate save box to be actioned and then allow the quit/ shutdown action to continue.

The Dr Wimp Manual gives a more detailed example using an error box to warn that unsaved data exists and showing how to implement the usual asterisk in a window title when unsaved data exists.

The second function is `PROCwimp_quit` and its main purpose is to allow the user to initiate quitting from the application.

This is done by calling:

```
PROCwimp_quit(type%)
```

where `type%` is set to 0 to quit the application or 1 to invoke the Wimp Shutdown procedure *(which latter will quit all running applications, of course)*.

Typically, as indeed was the case in our tutorial application (Line 2730), the call is made with `type%` set to 0 and normally follows from the user selecting ‘Quit’ from the iconbar menu.

The main result of making this call is, as implied above, to cause `FNuser_quit` to be called by the `DrWimp` library.

The Dr Wimp manual contains a whole chapter on managing quitting and shutdown.

27. Other facilities

The preceding chapters have attempted to introduce all the main features of Dr Wimp, but the coverage is by no means total and the package's own on-disc Manual needs to be examined to complete the picture - particularly its Section 3 which lists all the available user-functions and wimp-functions. (**The Manual is always updated with each release of a new Dr Wimp Version and therefore the list of available user-functions and wimp-functions applies only to that Version.**)

The following list shows some of the topics covered by Dr Wimp and not mentioned elsewhere in this book:

Finding out contents of a directory
Reading 'system variable' contents
Hourglass
Changing 'WimpSlot'
Introducing a pause
Issuing a 'star command'
Plotting some standard geometric shapes
Alerting a screen Mode change
Intercepting 'wimp messages'
Implementing the 'iconiser' protocol
Reading and setting window scroll
Displaying a 'banner'

Appendix 1. SYS calls (SWIs)

Normal Wimp programming depends so much on using SYS calls that a review of how to use them is worthwhile - even though using Dr Wimp makes this invisible to us.

SYS calls are the means provided by RISCOS to give the programmer easy access to a multitude of useful Software Interrupt (SWI) routines. The PRM index lists over 18 pages of SYS calls so there is not much you can't do!

In the earlier versions of BBC Basic the star command *FX was provided for this purpose (and is still available today, for backward compatibility). This allowed very many OS features to be selected/deselected e.g. enable/ disable the arrow keys; set flash rate of flashing colours; etc. It worked very well with 'set/reset' types of actions but was limited and somewhat awkward to handle when input parameters needed to be passed to, or output results were wanted from, the OS routines.

In BBC Basic V (and VI) the keyword SYS provides a much enhanced and much easier-to-use facility.

Keyword SYS

The general format of SYS is best described with a real example:

```
SYS "OS_ReadModeVariable",-1,1 TO ,, ScreenWidth%
```

This particular SYS call, as the words imply, reads information about the screen mode - but let's concentrate on the general structure of the SYS statement rather than the particular example.

In its simplest form, a `SYS` statement breaks down into three parts:

name

input parameters (if any)

output variables (if any)

and the example above contains all three.

Name

The first item after the keyword `SYS` is `"OS_ReadModeVariable"` and this one-word string (i.e. no spaces) is the name of the particular routine.

Note that the name is very similar in construction to a variable name and the use of capital letters at the start of each 'sub-word' is the same policy used in this book for naming variables.

As with variables, the match must be exact i.e. the string name is case sensitive - so `"OS_Readmodevariable"` will not do.

The technical name for the routines called by `SYS` is SWIs ("Software Interrupts") and you may often see the two words used synonymously.

With such a large number of SWIs available there really is no option other than to use the PRM if you need to know fully what is available. Fortunately, we need to know only a few SWIs for the purposes of this book and they are introduced as needed.

Each SYS/SWI has a number as well as a name. Either can be used. Thus, in our example above:

```
SYS &35 , -1 , 1 TO ,, ScreenWidth%
```

would do the same thing. (It is usual to use hex numbers for SWIs.)

The string version makes it far easier to read what a listing intends, so you are recommended to keep to this - although the numbers are processed faster. (As you might expect, there are SWIs to convert SWI names to numbers and vice versa. They are:

```
    "OS_SWINumberToString" (&38)
    "OS_SWINumberFromString" (&39)
```

Variables can also be substituted for the direct SWI name/number if you wish. Thus:

```
    ModeInfo% = &35
    SYS ModeInfo% , -1 , 1 TO ,, ScreenWidth%
```

or:

```
    ModeInfo$ "OS_ReadModeVariable"
    SYS ModeInfo$ , -1 , 1 TO ,, ScreenWidth%
```

would each work equally well.

Input parameters

After the SWI name/number, but before the `TO`, appears some numbers separated by commas - including the comma after the SWI name/number.

These are input parameter values being passed by the programmer to the routine. The parameter values need to be in the order and of the type specified (by the PRM) for the particular SWI, which are usually an integer number but sometimes a string - real numbers are not used here. In particular, memory locations (integers) are very commonly required as input parameters.

These parameters are passed to special storage locations called "registers", of which up to eight are available to Basic, named `R0` to `R7`.

Not all SWIs use all eight registers and in our example only two are used and both need integer numbers (i.e. here `R0` is -1 and `R1` is 1). Again, the PRM is needed to find the details of how many parameters/registers are used, what each of them means and what type and range of values each can take.

In our example, `R0` is used to pass the screen mode number we are interested in, or -1 is used to denote "the current mode in use". So, we could have put, say, 12 here if we had wanted to know about `MODE 12`. The next parameter/register (`R1`) is used to choose which item of mode information we want to know about - and the PRM lists thirteen items to choose from. Our example has used 1, which means "How many text characters per line in this mode?" (Other items include No. of text lines, maximum no. of colours, pixel size, etc.)

When a string is passed as a parameter, it is actually the memory address of the string that is passed to the register i.e. as with indirected icon text - and this is done automatically by Basic, so the programmer need only enter the string in the right place in the `SYS` statement (directly in quotes, or by reference to a string variable name).

Entering parameter values

If we are providing input parameter values to be placed in the registers, it is pretty obvious that we have to ensure that we list them in a manner which causes no doubt as to which value is to go to which register.

Generally, this is simply achieved by listing the input values in register number order (`R0` first, then `R1` and so on). However, it is quite common for an SWI to use only a few registers, not necessarily starting with `R0` and/or not necessarily consecutive.

To cope with all variations unambiguously, commas are used to "acknowledge the presence" of any register not used but preceding a used register.

This is easier to show than to explain in words. Imagine an SWI requiring input parameter values to be placed only in registers `R2` and `R4` - with a string and a number respectively. To be unambiguous, our `SYS` statement would therefore need to show the input parameter values as follows:

```
SYS "SWI_NameString" , , , "R2InputString" , ,
     R4InputInteger%
```

Effectively, what this shows is that each input parameter value needs to be regarded as being preceded by a comma i.e. `<, Value>` and if the value is not given then the comma must still be used, to show that a value is missing. In our mythical example above, the first two commas show that values for `R0` and `R1` are missing; the third comma is the one which precedes `"R2InputString"`; the fourth comma shows that an `R3` value is missing, and the fifth comma is the one preceding `R4InputInteger%`.

If the SWI is being used only to make an input (i.e. no output results to be returned) then the `SYS` statement simply ends after the sequence of input parameters.

Output variables (with the keyword TO)

If a SWI provides any output results, then the same set of registers (`R0`-`R7`) are used for the result values - which again may be integer numbers or strings. (Thus, any input values placed in the registers may well be overwritten by output values from the routine.)

The `SYS` statement helpfully allows these output results to be assigned directly to Basic variables. This is done by using the keyword `TO` after the input parameter values (if any) and then listing the required ‘destination variable’ names of the right type, separated by commas, in the order corresponding to the registers. As before, commas are used if necessary to cope with registers whose value is not required for output.

Thus, in our earlier real example:

```
    SYS "OS_ReadModeVariable",-1,1 TO ,, ScreenWidth%
```

we have used `TO` followed by `,, ScreenWidth%` - because the PRM tells us that the output we want is an integer number and it is going to be placed in register `R2` in this case. Further, the PRM tells us that registers `R0` and `R1` will be left unaltered from their input values.

Hence, we are not interested in the output values in `R0` and `R1` but we have to ‘acknowledge their presence’ and this we do by putting a comma to represent each of them i.e. two commas in this case. We can then follow these commas with the variable name

`Screenwidth%` and there will now be no doubt that this is our required destination for the `R2` output result.

Note that there is an important difference in the use of commas between the input parameters and the output results. Assuming that both are present in the `SYS` statement, then there is always a comma in front of the input `R0` value (i.e. after the SWI name/ number) but not in front of the `R0` output result variable (i.e. after the `TO`). If `TO` happens to be followed immediately by a comma it means that we have decided not to assign the `R0` output result to a variable.

To demonstrate the point we can expand our previous mythical example to:

```
SYS "SWI_NameString" , , , "R2InputString" , ,
            R4InputInteger% TO R0OutputVariable%,
               , R2OutputVariable%
```

when output results from R0 and R2 only are required (and assuming they are both integer numbers).

Or:

```
SYS "SWI_NameString" , , , "R2InputString" , ,
           R4InputInteger% TO , ,
           R2OutputVariable%
```

if only the result from R2 is wanted.

Parameter blocks

Frequently in Wimp programs, we need to send/receive rather a lot of data to/from a SWI in a `SYS` call - far more than can be done with eight registers. For instance, when using the SWI for defining a window on a Wimp screen we need to pass the size, colours, heading text etc. - maybe 20-30 items. In such cases, great use is made of “parameter blocks”.

A parameter block is simply a block of memory set aside by the programmer for the purpose of holding the required data; maybe a few bytes or maybe several ‘pages’ of memory.

The starting address of the data block is then passed as a parameter to the SWI - and the SWI automatically reads the data from the block (previously placed there by the programmer) and the SWI often puts some or all of the output result data into the same block (this time for the programmer to extract and use).

It is the PRM, of course, which details which parameters need to be of this type and the minimum data block necessary and where each item of data needs to be placed within it.

To set aside a block of memory for this purpose we usually use the keyword `DIM`, For example:

```
DIM BlockName% SizeOfBlock%
```

remembering that there must be a space between BlockName% and SizeOfBlock%.

Final points

By and large, with 18 pages of available SWIs, there are very few aspects of the OS that you cannot directly interface with the `SYS` keyword. It is probably worth noting that SWI names are very helpfully structured - as our earlier example demonstrated:

```
SYS "OS_ReadModeVariable"
```

The first part of the name - before the underscore character - gives the heading of the group of SWIs that this one belongs to - the ‘OS’ group in this case. There are ‘headings’ for printer drivers, sound, Wimp, fonts, colour etc. so it is easier to find what you are looking for.

Dr Wimp

If you use Dr Wimp all of the `SYS` calls are hidden from view, of course. Indeed the main purpose of Dr Wimp is to do just that, relieving you of the considerable intricacies of registers and parameter blocks.

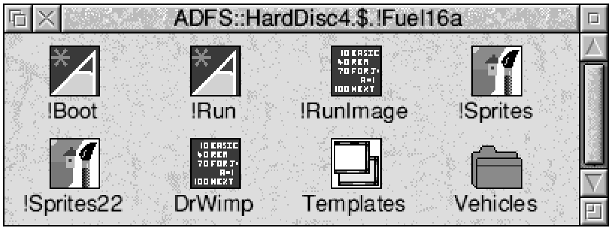
Appendix 2. Reason Codes

0*	**Null code ("Nothing has happened requiring application action")**
1	**Redraw window request**
2	**Open window request**
3	**Close window request**
4*	**Pointer is leaving window (or that window area is now covered)**
5*	**Pointer is entering window (or that window area is now revealed)**
6	**A mouse-click has occurred**
7	**A user drag action has just been completed (i.e. at 'drop' position)**
8	**A keyboard key has been pressed in a window with input focus**
9	**A selection has been made from a menu**
10	**Window scroll request**
11*	**The caret has gone to another window (maybe in another task)**
12*	**The caret has arrived at a window (maybe from another task)**
13	**("Poll word non-zero") Used to force task to carry out a priority job temporarily**
14	**(Reserved)**
15	**(Reserved)**
16	**(Reserved)**
17*	**(Used to manage messages of the Wimp Messaging system.)**
18*	**(Used to manage messages of the Wimp Messaging system.)**
19*	**(Used to manage messages of the Wimp Messaging system.)**

* These Reason Codes can be masked out during task initialisation.

Appendix 3. Application resources

Open any application directory and you will probably find a fairly similar-looking set of files/folders. The following screens hot is of our tutorial application, for instance.



These files/directories hold various ‘resources’ needed to allow the application to operate as intended. Typically, at least some of the following files/directories will be present:

```
!Run
!Boot
!RunImage
!Sprites
Sprites22
Templates
Messages
!Help
Modules
Fonts
```

The reason why most applications use the same structure for these resources is that the RISCOS has in-built management arrangements which offer considerable advantages to the programmer if the resources conform to recommended practices - as described in the RISCOS Style Guide.

To give a practical example, if you double-click on an application icon the RISC OS will automatically look inside that application's directory for a file called `!Run`, and run it. So, the standard way to make an application start via the usual Wimp method is to include a `!Run` file and put the necessary instructions in it.

The other items offer their own advantages, as described below:

!Boot (not to be confused with your computer's main start-up !Boot application). When you open a directory window, the RISCOS looks to see if there are any applications inside it (i.e. directories with names starting with "!") and, if so, looks inside their directories for a `!Boot` file, which it then runs.

Typical of the contents of a !Boot file are the lines:

```
Set TestApp$Dir <Obey$Dir>
IconSprites <TestApp$Dir>.!Sprites
```

The first line creates a 'system variable' called `TestApp$Dir` and stores in it the application's current full path (i.e. the path of the directory which was double-clicked - not the full path of the `!Boot` file).

The second line tells the Wimp where any sprites special to that application are being held - the application sprite itself, for instance - and effectively adds these sprites to the common pool that can thereafter also be accessed by any application, until the computer is switched off.

If the sprite file contains a sprite whose name is exactly the same (but all in lower case) as the application's name, then the Wimp will display that as the application's identifying sprite in the opened directory window.

Another typical item in this file is an instruction to tell the Wimp that certain file types 'belong' to this application - so that, thereafter, double-clicking on such a file will cause this application to be run and for the file to be loaded into it.

By running an application's `!Boot` file, the Wimp is said to be 'seeing' the application - because it now knows where the application resides (and its sprite, usually) and can find it again if need be.

A `!Boot` file is not mandatory. If it does not exist then the application will still have been ‘seen’ if its parent directory window is opened, but may appear in the window with the default RISCOS-supplied application sprite instead of its unique one.

!Run As already stated, this file is run when you double-click on the application’s icon. Typical contents are:

```
Set TestApp$Dir <Obey$Dir>
IconSprites <TestApp$Dir>.!Sprites
WimpSlot -min 128K -max 128K
Run <Obey$Dir>.!RunImage
```

and there can be many more lines of commands.

Note first of all that the two lines in the `!Boot` file are usually repeated - in case the application is started before it has been ‘seen’ e.g. from the Command Line.

The third line tells the Wimp how much memory to allocate for the application and this value will be reflected in the application’s entry in the Task Display. If this line is not included the Wimp will allocate a default amount of memory shown by the ‘Next’ item in the Task Display. (The ‘Next’ value can be changed by dragging its bar in this display or by using a third parameter with a `WimpSlot` command - usually in the start-up `!Boot` application.)

The last line tells it to run the `!RunImage` file i.e. run the program.

It is essential to have a `!Run` file if you want an application to start in the usual way i.e. by double-clicking on its application icon.

`!Boot` *and* `!Run` *files are both of the ‘Obey’ file-type and each line of action in them is a ‘star command’.*

!Sprites and !Sprites22 RISCOS provides a large number of standard sprites in a central resource which any application is free to use. But if you want **the Filer** to use some other sprites to display your application and any of its special files, then these sprite-files are the place to keep them. Ideally, application and file sprites should be provided for ‘small icons’ and pinboard use, as well as standard size versions. The RISCOS Style Guide gives the rules for sizing/naming them.

The contents of the two files can look identical at first sight, but `!Sprites22` is used to hold high-definition sprites which are brought into play automatically if a high-definition screen mode is used. `!Sprites23` is also seen sometimes and is for high definition black-and-white screen modes.

It is important to appreciate that - to follow the Style Guide - these sprite-files are meant exclusively for use by the Filer and as such are effectively added to the central pool of sprites by the use of the Star Command `*IconSprites` in an application’s `!Boot` and/or `!Run` files (see above).

Any sprites intended for use solely by and within an application should strictly be held in a separate sprite-file - often just called `Sprites` - and handled by creating a ‘user sprite area’. By doing this, the memory taken by the user sprites can be released when the user quits the application. Dr Wimp has special facilities for handling this type of sprite - see Chapter 19.

Templates This resource is covered in more detail in Appendix 7 and elsewhere in this book. It is a more convenient way to design and store window/icon definitions.

Modules and Fonts Just as with sprites, RISCOS provides facilities to store and manage special ‘re-locatable’ programming routines (called ‘modules’) and fonts.

If an application has a need for such resources then they are usually supplied in these application folders. However both of these types of resource can only properly be managed by the Wimp if they are added to its central pools.

Therefore, the usual procedure is for the `!Run` file to include a check for their presence in the central pool - and any that are not present are copied and initialised accordingly, before the `!RunImage` is run. Alternatively, sometimes this check is done as part of an installation procedure the first time the application is used.

Once they are added to the central pools it is usual for them to stay there 'permanently' - because they tend to be resources which are useful for more than one application. *(However, they can be removed if need be e.g. very occasionally a module might be incompatible with another application and need to be removed or disabled before it can operate.)*

Messages *(Not to be confused with the Wimp Messaging system!)* The Wimp has convenient facilities for handling the various message strings which appear in error/warning boxes etc. - by holding them in a separate textfile in a special way. Chapter 26 introduces this in detail.

The `!Run` file might typically be used to identify the location of the `Messages` file to the application.

!Help If an application has a `!Help` file, the Wimp will automatically detect this when the application is 'seen' and the 'Help' item on the Filer menu will be enabled when `<menu>` is pressed over the application icon - and selecting this item will display the file.

(This is independent of the interactive Help facilities offered by RISCOS and described in Chapter 26.)

Appendix 4. Window/Icon button types

The standard RISCOS mouse has three buttons which are called Select, Menu and Adjust. For right-handed people, these are usually configured in the order left/middle/right respectively. (In this book, clicking with these buttons is shown as `<select>`, `<menu>` and `<adjust>` respectively.)

The middle button (Menu) is special in that pressing it over a window or icon always causes the Wimp to report that the action has occurred.

However, for the Select and Adjust buttons, the characteristics of windows and icons can be defined so as to respond in different ways to different types of button clicks e.g. single clicks, double-clicks, drags, button release etc. can all be distinguished specifically, if required by the programmer.

The characteristic of window/icon design which determines this is called the ‘button type’ and the Wimp currently offers button types 0-15 (of which, button types 12 & 13 are not used).

The only sensible way to describe each button type is in a table and this is shown below.

Once a button type has been defined, the Wimp automatically manages things so that responses (Reason Code 6 - see Chapter 1) are only generated when the appropriate mouse-click actions occur.

In addition, for a single mouse-click, the Wimp identifies which mouse button has been pressed by using a ‘button number’, as follows:

	Button number
`<select>`	4 (binary 100)
`<menu>`	2 (binary 010)
`<adjust>`	1 (binary 001)

Note the binary values: the position of the 1 in the 3-bit sequence is different for each mouse button. If more than one button is pressed at the same time the returned number is simply the appropriate addition of these values e.g. `<select>+<adjust>` pressed together would be represented by 4+1=5 (binary 101).

Do not confuse 'button type' and 'button number'!

Where a button type permits more than one click action (e.g. Type 10 'Double-click/click/drag') the Wimp distinguishes between these by simply multiplying the above button number by a given factor e.g. times 16, for drags.

If double-click is one of the allowable actions i.e. Types 5, 8 or 10, the Wimp will also report the initial click first.

Dr Wimp handles the Wimp responses from all the allowable button types, for both windows and icons, via `PROCuser_mouseclick` and `PROCuser_menu` - see Chapter 8.

Thus, Dr Wimp has no restrictions when using window templates to design windows/icons. Neither is there any restriction on icons if they are created within a program via `FNwimp_createicon`.

Button type		***Description***
Number	**Name**	
0	*Never*	*Mouse-clicks are ignored (except <menu>)*
1	*Always*	*Wimp continuously reports pointer presence over window/icon*
2	*Click (auto-repeat)*	*Wimp reports click (auto repeat)*
3	*Click(single)*	*Wimp reports click*
4	*Release*	*Wimp reports button release (If icon, click selects, moving pointer away deselects)*
5	*Double-click*	*Wimp reports double-click (If icon, click selects)*
6	*Click/drag*	*as for 3, but Wimp also reports drag*
7	*Release/drag*	*as for 4, but Wimp also reports drag (and moving pointer away from icon does not deselect)*
8	*Double-click/drag*	*as for 5, but Wimp also reports drag*
9	*Menu*	*as for 3 (but for icons, pointer over icon selects and moving away deselects)*
10	*Double-click/click/drag*	*Wimp reports double-click and single click and drag*
11	*Radio*	*as for 6 (If icon, click selects)*
12	*(reserved)*	
13	*(reserved)*	
14	*Writable/click/drag*	*(Applies to icons only) As for 6 but also click gains caret and parent window gains input focus*
15	*Writable*	*As for 14, but without drag*

The Button type numbers are those used for the parameter `button%` in Dr Wimp's wimp-functions `FNwimp_createicon` and `FNwimp_createwindow` - see Chapter 13.

Appendix 5. Icon validation string

A ‘validation string’ is simply one of the characteristics of an icon. It is a particularly powerful feature because it allows a very large range of icon features to be set.

As with the other icon characteristics (e.g. size, position, button type etc.) the validation string must be defined at the time the icon is created - whether via a window template or by direct creation within a program (using `SYS` calls or Dr Wimp’s wimp-functions). If you do not wish to use a validation string then it needs to be set to a null string, which is done automatically by template editors.

The one thing that must be done if a validation string is to be used is that the icon must be defined as having, at least, indirected text.

The validation string, which needs to be in the correct format, comprises one or more ‘Commands’. If more than one command is present they must be separated by a semi-colon.

Commands

These commands all take the form of a single letter (best as a capital, for good visibility) followed by a series of letter/number/text codes. There are eight official single-letter command types A, D, F, K, L, P, R and S - but the popular window template editor `!TemplEd` (and possibly others) also uses N.

A typical, multi-command validation string looks like this:

```
    Ktar;Pptr_write;F17
```

which is a validation string using just three of the available commands - K, P and F.

The only sensible way to introduce the various commands is one by one:

A-command : (A)llow

This controls the user-input to writable icons. Essentially, it determines which keyboard characters will be accepted/rejected for the text in the icon. It therefore provides a very powerful first line of input validation.

The rationale of the structure of this command is that it states which typed characters (in the ASCII range 32-255) are allowed to be entered into the writable icon, assuming it has the input focus.

Thus the command:

```
A0-9a-z
```

would mean “allow the icon to display any of the digits in the range 0-9 inclusive and any of the lower case letters from a-z inclusive”. Thus, if the user types:

```
Joe90
```

when the icon has the caret, the icon would display only:

```
oe90
```

because the upper-case “J” is not allowed.

Any key presses which are not allowed are either dealt with by the Wimp automatically or passed on via the conventional Reason Code 8 - see Chapter 7 and Appendix 2.

The symbol ~ (the tilde, ASCII character 126) is used to specify exceptions or “not the following”. For example, the string:

```
A0-9a-z~tbg
```

means “allow any of the digits from 0-9 inclusive and any of the lower case letters from a-z inclusive, except the characters t, b and g”

By default, all the normal keyboard characters are allowed. It is therefore common to see the A-command immediately followed by ~ and then some exceptions. For example:

```
A~0-9
```

would mean “allow all characters except digits in the range 0-9”.

Because the four characters ;~\- (ASCII characters 59, 126, 92 and 45) are also normal keyboard characters but have special meanings within the validation string, if you want to specify them for inclusion/exclusion you need to precede them with the \ character. Thus:

```
A~\~
```

means “allow all normal keyboard characters except ~”.

Finally, the space (ASCII character 32) is a valid keyboard character, so using a space within an A-command will actually mean something!

Don’t forget that the A-command only determines the characters that will (or will not) be allowed to appear in a writable icon as a result of a keyboard keypress. It does not control whether the keypress is reported to the application and passed on to FNuser keypress. See the K-command below.

D-command: (D)isplay

This is typically used for icons which are going to have a password typed into them. The command:

```
D*
```

would mean that any (of the allowed) characters typed in would appear in the icon as *.

So, if you wanted the password to look like a row of dashes, you would need to use:

```
D\-
```

i.e. use a preceding \ as described in the A-command, because - has a special meaning in validation strings.

F-command: (F)ont colour

This specifies the background and foreground colour for anti-aliased fonts - note the order. For example:

```
Fc7
```

would set the background colour to Wimp colour &c (decimal 12) and the foreground colour to Wimp colour 7.

By default, `"F07"` is used i.e. Black on White normally.

K-command: (K)eys

This command is probably the most complicated and it determines how the caret moves between icons and also allows certain keys to be given specific functions. **It is particularly relevant to which keys are passed to the `key%` parameter in Dr Wimp's user-function `FNuserkeypress`.**

> *The scope of the K-command has changed with different RISCOS Versions. The following is believed to be relevant to Versions from 3.60 onwards.*

The command is followed by one (or more) of the letters R, A, T, D or N, which are shown in upper-case here but are best used in lowercase in the Command - for clarity (see later):

R -pressing **`<return>`** will move the caret to the next writable icon (next in icon number sequence) within the same ESG - if there is a next icon. If it is the last (or only) writable icon, then ASCII code 13 will be passed to the application, (i.e. in Dr Wimp, **`key%`** will be 13 in the third parameter of **`FNuser_keypress.`**)

A -Pressing the **`<up>`**/**`<down>`** arrow keys will move the caret to the next writable icon (if there is one) in the same ESG. If it is the first/last writable icon then the caret will be moved to the last/ first writable icon. That is, the caret will be cycled round a group of writable icons. The arrow keypresses will not be passed to the application.

T -Pressing **`<Tab>`** or **`<Shift-Tab>`** will have the same effect as for the **`<down>`**/**`<up>`** arrow keys as described above.

D -Pressing any of the keys **`<Copy>`**, **`<Delete>`**, **`<Shift-Copy>`**, or **`<Ctrl-U>`** or **`<Ctrl-Copy>`** will cause the appropriate key code to be passed to the application in addition to the usual editing action occurring.

N -All keypress codes will be notified to the application, even if they are automatically handled by the Wimp. *(This is the K-command needed to get ordinary keyboard letter presses in a writable icon passed to* **`FNuser_keypress`**.*)*

K-commands can (and often do) have more than one of these options selected. For instance:

```
Kar
```

will activate both the `<return>` and arrow key functionalities as described.

(This example also demonstrates why it is clearer to use the subsequent letters in lower-case.)

L-command: (L)ine spacing

This allows long text to be formatted over more than one line in an icon. It is particularly useful for Error boxes.

Note that:

The text must be centred both horizontally and vertically;

The icon must not be 'writable' (i.e. not user writable);

The font cannot be anti-aliased.

P-command: (P)ointer

Allows the pointer shape to be changed whilst it is over the icon. Thus:

```
Pptr_write
```

will cause the pointer to change to the common blue vertical text cursor when it is over the icon.

The text after the command (here, `"ptr_write"`) must be the name of a sprite in the Wimp sprite pool and it must be a 4-colour sprite.

(Using `*IconSprites` *allows your own sprite designs to be loaded into the Wimp sprite area - see Appendix 3.)*

R-command: bo(R)der

This command sets the particular type of border for the icon - assuming that the border option has also been selected - and will override the default border.

R is followed by one of the following numbers which specifies the border type:

```
0 - normal border
1 - 'slab out'
2 - 'slab in'
3 - 'ridge'
4 - 'channel'
5 - action button (highlights when selected)
6 - default action button (highlights when
             selected)
7 - 'editable field'
8 or more - as for 0
```

If 5 or 6 is used, a second number can specify the particular highlight colour (which is Wimp colour 14 by default)

S-command : (S)prite

Used to specify the name of the sprite in a text-plus-sprite icon.

Two sprite names can be used, comma separated, to specify a second sprite to be used when the icon is highlighted - otherwise the default ‘EORed’ colours are used. If two sprites are used they must be the same size. *(An example of using two icon names can be seen with the ‘nudger’ icons in, say, the* `!Slider` *application in the* `Examples` *folder.)*

N-command : (N)ame (!TemplEd additional feature)

If you use `!TemplEd` - included in the Dr Wimp package and described in Appendix 7 - it allows the use of a further validation string command called the N-command.

This permits each icon to be given a name, which can then be used in the application programming instead of icon numbers. A utility is included to extract the icon names from a template file and to present them for direct inclusion into a Basic or C program listing.

Appendix 6. !Fabricate

This is a very simple but highly effective utility application included with the Dr Wimp package - in the `Utils` directory.

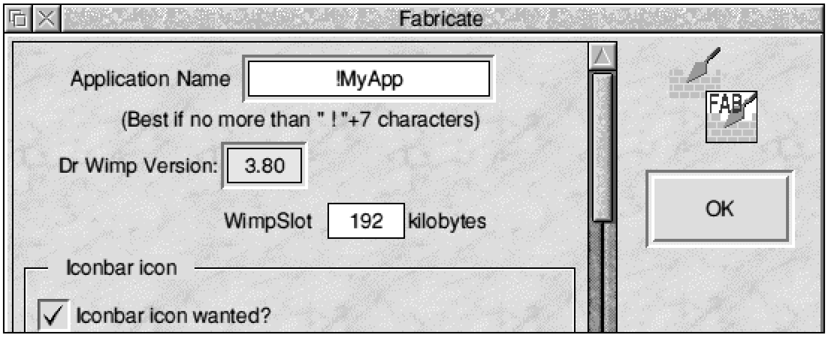
Its purpose is to produce, within a few seconds, a fully-working, **customized** initial application at the start of a new programming task - ready for your more detailed program development.

`!Fabricate` has been developed considerably in recent years and can now provide the programmer with a customised starting point well beyond the supplied seminal `!MyApp` application.

To use it, you do have to be familiar with at least Chapters 4 and 5 of this book (or the first few pages of the Dr Wimp Manual's own tutorial) - but that is all.

!Fabricate comes with its own comprehensive `!Help` file, so we will give only an outline description here.

It is easiest to explain how to use `!Fabricate` by showing the utility's main window. So load `!Fabricate` onto the iconbar in the usual way and click `<select>` over it to get:


Fabricate

Application Name: !MyApp

(Best if no more than " ! "+7 characters)

Dr Wimp Version: 3.80

WimpSlot 192 kilobytes

Iconbar icon

✓ Iconbar icon wanted?

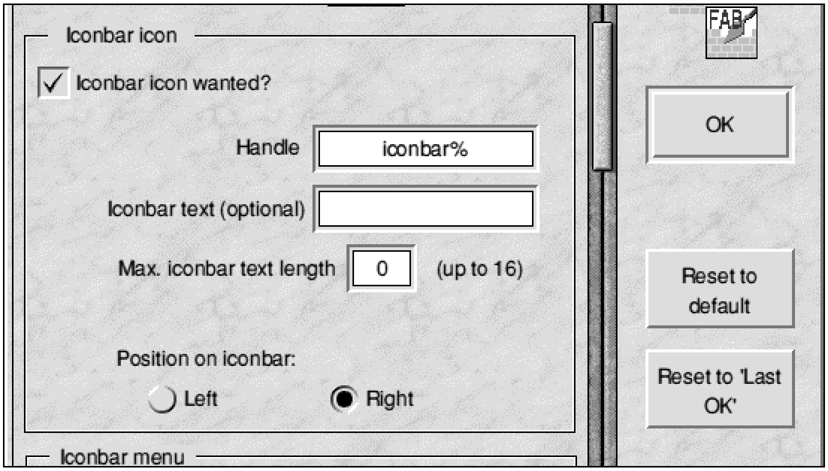
Note that the window has a pane within it which has a scroll bar. The above screen shot has the uppermost part of the pane visible and the process simply requires us to work down this pane from top to bottom.

So, the first place to visit is the writable icon with the label 'Application Name'. Delete what is there already and enter the name you want for your new application - don't forget to start it with a "!".

The yellow icon shows which Dr Wimp version your new application will be using. You cannot alter this - and your version of **`!Fabricate`** may show a different Dr Wimp Version here. But, whatever it is, the corresponding **`DrWimp`** library will automatically be placed in your new application directory when created.

The next icon shows the WimpSlot size that will appear in your new application's **`!Run`** file. You may well have to increase this (by editing the **`!Run`** file directly) as your application develops. The shown value should be enough to get you started.

Now scroll the pane down to get:


Iconbar icon

✓ Iconbar icon wanted?

Handle: iconbar%

Iconbar text (optional):

Max. iconbar text length: 0 (up to 16)

Position on iconbar: Left [✓ Right]

Iconbar menu

This part concerns the iconbar icon. So, decide whether or not you want your application to start with an iconbar icon (most do). If so, check that 'Iconbar icon' is ticked - which will ensure that the rest of the options in this section are enabled for your detailed choices. You can accept the default entries or make your own.

Type into 'Handle' the handle/variable name you want to use for the iconbar icon handle in your application's `!RunImage`.

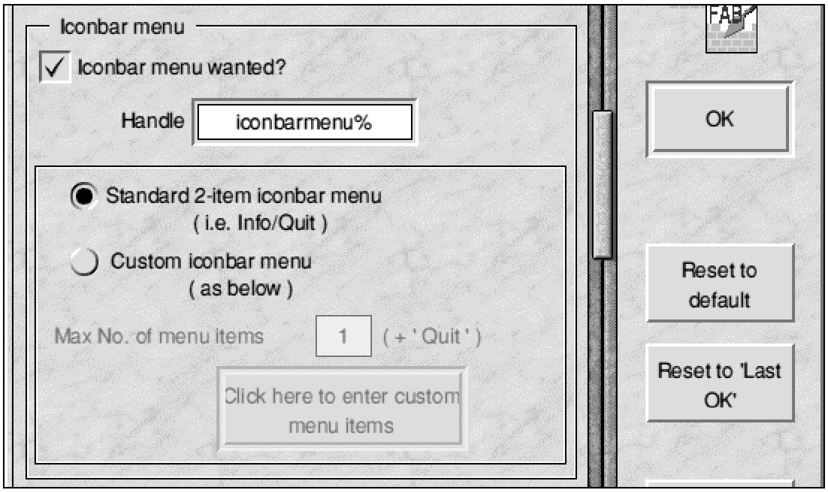
If you want the iconbar icon to have text beneath it from the start then this is entered at 'Iconbar text'.

If you want to alter this text to a larger string during the program run then you need to specify the maximum size in the box below.

(If you don't want an initial text but want to add some during program run you can do that by entering a null string in 'Iconbar text' but setting the required later maximum size below.)

The final choice in this section is the radio icons choosing which side of the iconbar you want your iconbar icon to appear.

Now scroll the pane down to get:



Iconbar menu

✓ Iconbar menu wanted?

Handle: iconbarmenu%

● Standard 2-item iconbar menu (i.e. Info/Quit)

○ Custom iconbar menu (as below)

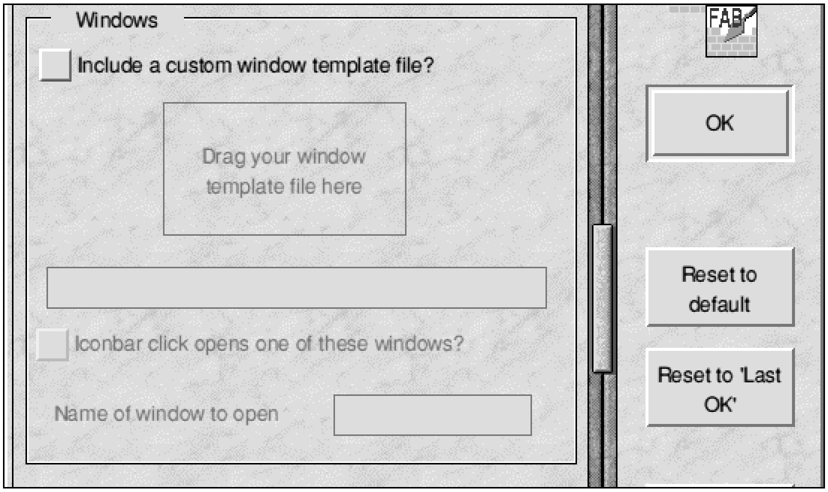
Max No. of menu items: 1 (+ 'Quit')

This section concerns the iconbar menu.

If you want your starter application to come ready with a standard 2-item iconbar menu i.e. with just the usual ‘Info’ and ‘Quit’ items, then tick the box accordingly and decide upon the menu handle name.

Alternatively, you can start off with your own design of iconbar menu. Just select the ‘Custom iconbar menu’ radio button and enter your required maximum menu size. Clicking the lower button will then open another window into which you can enter your choice of menu item text.

Now scroll the pane down to get:

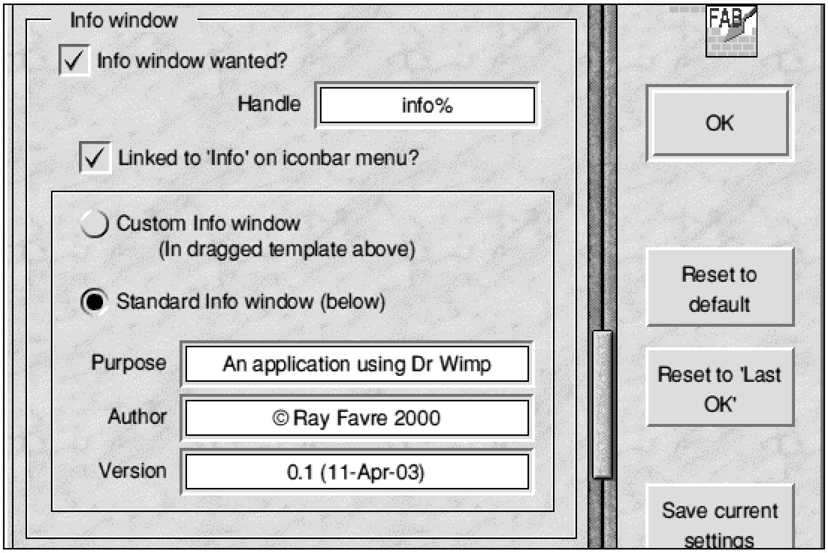

Windows
Include a custom window template file?
Drag your window template file here
Iconbar click opens one of these windows?
Name of window to open
FAB
OK
Reset to default
Reset to 'Last OK'

Via this section you can get your starter application to contain your own window template file **and** for the `!RunImage` to load all of its windows **and** for one of these windows to be opened when the user presses `<select>` over the iconbar icon. This carries out quite a lot of routine coding work for you and is well worthwhile.

Clicking the first option icon enables the drag box for you to drag in your own template file.

Clicking the lower option icon arranges for you to get a menu of your window names and to choose which one will be opened from the iconbar click.

Again, scroll the pane down to get:



Info window

[✓] Info window wanted?

Handle: info%

[✓] Linked to 'Info' on iconbar menu?

○ Custom Info window (In dragged template above)

● Standard Info window (below)

Purpose: An application using Dr Wimp

Author: © Ray Favre 2000

Version: 0.1 (11-Apr-03)

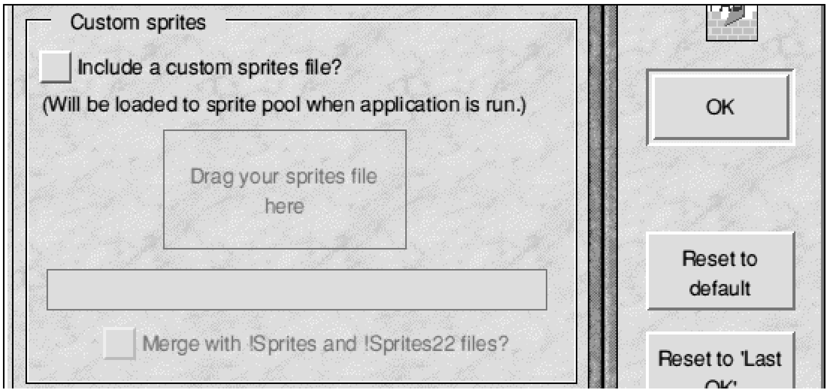
As you can see, we have now moved on to the 'Info' window. If you want your application to have one (and most do) first tick the 'Info window' item. Then, as for the iconbar icon, decide what handle name you want for your 'Info' window and enter this.

Then decide if you want this window to be attached as a 'sub-menu' to the "Info" item on your iconbar menu (the usual way).

You then have a choice of using a standard Info window (supplied by Dr Wimp) or one of your own. If the latter it must be included in the window templates file dragged into `!Fabricate` in the previous step.

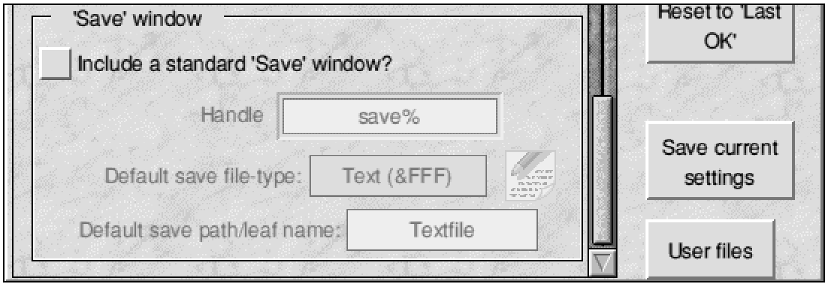
If you want the standard window then select the appropriate radio icon and enter your required text into the ‘Purpose’, ‘Author’ and ‘Version’ sections.

Now scroll the pane down to get:


Custom sprites
Include a custom sprites file?
(Will be loaded to sprite pool when application is run.)
Drag your sprites file here
Merge with !Sprites and !Sprites22 files?
OK
Reset to default
Reset to 'Last OK'

This section allows you to include your own sprite file with the starter application and arrange for it to be merged with the application’s `!Sprite`/`!Sprite22` files (provided automatically) - and hence loaded into the Wimp sprite pool via the standard `!Boot` and `!Run` files.

Now scroll the pane down to get the final section:


'Save' window

Include a standard 'Save' window?

Handle: save%

Default save file-type: Text (&FFF)

Default save path/leaf name: Textfile

This allows you to include a standard ‘Save’ window (as per tutorial) with your starter application and to choose which type of file it will save - and what path/leaf name appears in its writable icon.

Creating the starter application

Having made all your choices we come to the final moment for creating your new starter application.

Click on ‘OK’ (near the top of the non-scrolling right side of the window) and a standard Save box will appear, showing an application icon.

Drag the application sprite to the Filer window of your choice and in a few seconds your new starter application will appear.

Open its application folder and you will see a complete typical set of application resource files (similar to that in Appendix 2) - plus a copy of the `DrWimp` library. The detailed contents will, of course, depend on the choices you have made in `!Fabricate`’s window.

You should examine the contents of each file in appropriate editors, to reassure and familiarise yourself with what, precisely, !Fabricate provides. It seems like magic the first time you use it!

In particular, the `!RunImage` will contain all the necessary code to reflect the choices you made in the `!Fabricate` window and, most importantly, it will include all the user-functions for the particular version of the DrWimp library included - some with code in them reflecting your choices in `!Fabricate`’s window.

Further, the `!RunImage` coding will include a fair sprinkling of comments so that you can see what code has been added and why.

As promised, everything is therefore ready for you to start developing your new application further from a starter application which is already a fair way downstream.

Don’t forget that **everything** provided is a starter position: you can change and/or add to it as you wish.

There are a few further things worth noting about `!Fabricate`:

Having gone through a specific selection of choices to create a new application you might want to save those choices for use again. You can do this from the ‘Save current settings’ button on the right of the window. This brings up a Save box to save a file in the “FabFile” format. These can be loaded back into `!Fabricate` simply by dragging them in - or by selection from the menu that appears when you press ‘User files’.

Each time you press ‘OK’ to create a new application the settings are automatically saved into a ‘FabFile’ called ‘LastOK’ which you can restore by pressing ‘Reset to Last OK’ - but don’t forget that this file will be overwritten the next time you press ‘OK.

You can also reset to the default settings at any time by pressing ‘Reset to default’.

You will get a warning if you attempt to create a new application in a directory which already contains an application with the same name. **Pay heed to that warning** as it will completely wipe out the whole of the existing application if you continue.

`!Fabricate` is always upgraded with a new Dr Wimp version. So, be careful to use the latest version.

!Fabricate really is a very worthwhile time-saver at the start of any new application development. Get to know it!

Appendix 7. The !TemplEd application

For all Wimp applications, the user action revolves around windows and therefore window/icon design and layout is of paramount importance. Indeed it is often the first thing done when starting to program a new application *(and this would fit in well with the use of !Fabricate)*.

As this book has shown, you can design your windows/icons the hard way or use a graphics editing utility to produce a window template file i.e. using a 'template editor'. Even where there is good reason to avoid using templates overtly (perhaps to keep them safe from 'meddling') it is always a lot easier to use a template editor for the design and convert its output subsequently - see Chapter 13 and Dr Wimp's **`!CodeTemps`** utility.

Because of their importance to both 'ordinary' Wimp programming and Dr Wimp, it seems worthwhile to describe how template editors are used and a particular one therefore needs to be chosen for this.

There are several template editors around, but one of the most popular - and the one supplied with the Dr Wimp package - is **`!TemplEd`** a PD Freeware product by Dick Alstein. This appendix looks at this utility (Version 1.34) in some detail - but it is not intended to replace the documentation supplied within the utility itself.

!TemplEd

Introduction

`!TemplEd` is a Wimp-compliant graphical template editor i.e. it allows you to design and/or edit window templates in a conventional Wimp manner, mainly using 'drag and drop' actions.

In its supplied state, it can cope with up to 100 windows in each template file and up to 200 icons per window. There is also a limit of 8000 bytes of indirected buffer space. These values are large enough to cope with most needs but can be altered if need be in the application's **`!Run`** file.

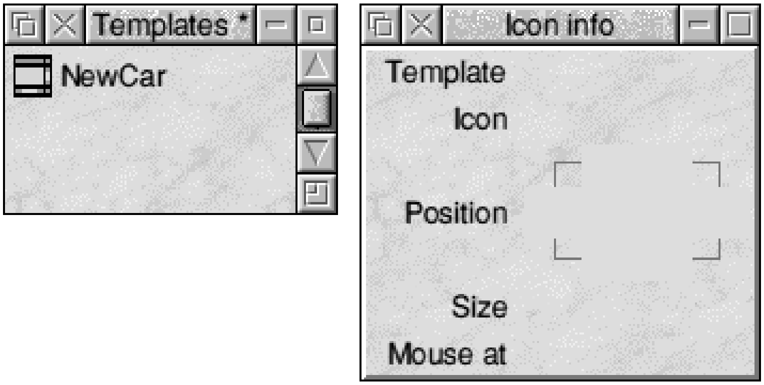
Only one template file at a time can be handled by !TemplEd, so if you need to work with more than one file at a time e.g. to copy windows from one template file to another, multiple copies of !TemplEd need to be running, each with its own iconbar icon. This is usually no problem (apart from the common difficulty of remembering which iconbar icon represents which template file!).

Creating a new template file from scratch is covered in detail below but, provided !TemplEd has been 'seen', double-clicking on an existing template file will start up !TemplEd and load the file ready for viewing/ editing. *(This is often useful to find out how others have designed their windows/ icons.)*

Creating a new template file

To make the description more meaningful, the following goes through the sequence of designing the 'NewCar' window (in a templates file called `"Templates"`) from the tutorial application in Chapters 4 to 16.

To start a new window templates file, `!TemplEd` is loaded onto the iconbar in the usual way. Pressing `<select>` over it brings up the following two small windows, one in each top corner of the screen. (Note that the left one will be empty at first and not exactly as shown.)



The right-hand one (called "Icon info") is not of initial use, but note that by default it appears at a reduced size with its size icon in the 'small' state accordingly. It is worth bringing it up to full size - as shown here - to see what else it contains and then reducing it again until needed. If you don't do this there is a distinct risk that you will not realise that some of the window is hidden and might be confused a little later!

The left-hand window (called `"Templates"` by default) is the one mostly used at first. Pressing `<menu>` over it brings up the `TemplEd` menu which, at the start, has only the first of its several items available for selection, called `'Create'`. Sliding over this item brings up a writable box into which the name of your new window needs to be entered. So type `"NewCar"` here and press `<return>`. *(The window name can always be changed later, so choose anything if you are not sure.)*

The results will be that a small blank default window opens on the screen (with `NewCar` also as its title) and a small representative icon appears in the `Templates` window, with `NewCar` as its label (exactly as in the previous screenshot). This icon will have a yellow centre (to indicate that its window is open) and the `Templates` window will have an asterisk added to its title (to show that unsaved data now exists).

To create other windows in the same `Templates` file you simply repeat the process of pressing `<menu>` and typing a different name in the 'Create' writable box. Each time you do this a new icon will be added to the `Templates` file and a new window will appear - all of the same default size at this stage.

You can save the `Templates` file from the 'Save' item on the `'TemplEd'` menu, which will now be enabled. As usual, you will have to drag the file icon to a directory of your choice the first time you save it.

Check that a yellow icon changes to white when you close its corresponding window - and double-click on the icon to re-open its window.

Many of the other items on the `'TemplEd'` menu are similar to the standard Filer menu and work similarly. A few others are different and we will come to them later.

Assuming that icons are going to be added, it is usually best to leave the default window unaltered at this stage and we will come back to the window characteristics later.

But it is good practice to save the new template file now - and frequently during the creation sequences…

Saving

Saving a complete template file is straightforward and the normal arrangements are available from the last item on the `'TemplEd'` menu - obtained by pressing `<menu>` over the background of the templates file window. There is no keyboard shortcut e.g. the common use of `<ctrl-F3>` is not available.

You can also save an individual window template by pressing `<menu>` over one of the template file window icons and following the 'Template xxxx' menu item to its sub-menu. Again, the last item on the sub-menu is 'Save'. This feature is also particularly useful for copying individual (or groups) of window templates from one template file to another - simply drag the save icon to the required template file window.

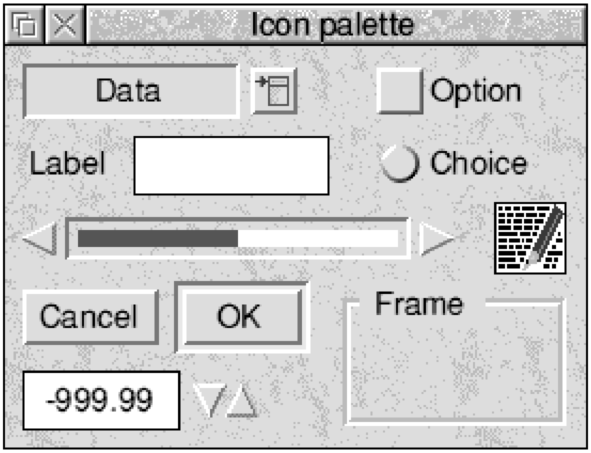
Adding icons

The description which follows creates several different types of icons and uses this as a vehicle to introduce the practical operation of `!TemplEd`. It covers all the more common icon-editing needs but there are many further options available whose details can be found in the `!TemplEd` on-disc manual. The important aspect of the following is that it uses a certain sequence for creating icons. Clearly, it is not the only possibility but it is offered as a sensible one until greater familiarity is gained.

Icon palette

Bring the `NewCar` window up to its full size and press `<menu>` over it. Above the dotted line, this 'Window' menu deals with icons only and at this stage only 'Create icon …' will be available. Select this and yet another window will open, called 'Icon palette'.

As its name says, this window contains a selection of all the basic icons you are likely to need. If you need something different it is best to choose the nearest one from the palette and 'tweak' its precise characteristics to your liking.



All the icons on the palette can be dragged (like files) so select the white, writable icon. Then press and hold `<select>` until the pointer changes to a hand and then drag the icon to the NewCar window and drop it. A copy of the icon now appears in the window (and the asterisk duly appears in the title of the `Templates` window once again).

Icon info window

Now bring the `Icon info` window up to full size and move the pointer over the `NewCar` window and then over its newly-acquired icon. The working of the `Icon info` window should now be obvious, but note that the icon number is 0. Icons in a window are numbered in the order they are created, starting from 0. Their numbers can be changed, as will be shown later. (Reduce the `Icon info` window again to avoid distraction.)

Icon editing window

We need to edit the new icon for the particular application. Let's assume that it is going to be the writable icon into which a new car's registration number is to be entered. Therefore, for instance, we want it to have a red outline and to display its text/numbers in outline font.

So, either double-click on the new icon or choose 'Icon#0.Edit...' from the 'Window' menu (after pressing `<menu>` when over the icon) - or select the icon and press `<ctrl-E>`. All will bring up the `Edit icon` window.

This window, which is quite 'busy' for the newcomer, is split into four main areas plus some action buttons. The first thing to note is that **changes to the icon are not implemented until you press 'Update' (or 'Update & Exit')**. This might take a little getting used to. For instance, pressing `<return>` will not implement the changes.

Button type

A good place to visit first is the 'Button type' section, about two- thirds of the way down. Button types are described in detail in Appendix 4 and as we selected the writable icon from the icon palette, it should be no surprise to see that 'Writable' already appears in the box.

However, in other cases you might not always want the exact settings from the icon palette. If you move the pointer over the box with 'Writable' in it, the pointer will change to the menu symbol and `<select>` will bring up a list of all 16 button type options, for your consideration. Selecting any item on the list will change the box entry accordingly. *(If you want this change to be put into effect, don't forget to press 'Update'!)*

Although you can visit this area at any time, it seems sensible to get the button type right from the start.

ESG (Exclusive Selection Group)

If you choose icons with the 'Menu' or 'Radio' button type *(or if you have several writable icons among which you want the caret to 'cycle' - see Appendix 5)* the ESG value will also need attention.

The ESG is a number in the range 0-31 inclusive, with the value 0 meaning that no ESG applies - and 0 is the default value and used by most icons. But if any other value is used the Wimp ensures that only one icon at a time can be selected (high-lighted) among those icons with the same ESG.

The most familiar application of ESG is with icons of the 'Radio' button type. Typically, there may well be a group of, say, 4 icons offering one choice from 4 mutually-exclusive features e.g. Up/Down/Left/Right. If you click over the 'Left' icon it is selected **and Up/ Down/Right are all de-selected**. You can only ever have one choice selected from among these 4. That is, they are all in the same ESG. This is arranged simply by setting the ESG value of all four to the same non-zero number (in the range 1-31) and making sure that any other non-zero ESG values used by other icons in the same window use different numbers from this group.

'Menu' icons also need to have a non-zero ESG - otherwise they blink rapidly whilst the pointer is over them. In this case, each 'Menu' icon needs to have its own exclusive ESG value. An example of this is given a little later.

Text/Sprites/Indirection

The three options at the top left of the `Edit icon` window are probably the next place to visit. This is the area where we decide whether or not an icon is going to contain text and/or a sprite, and how the text and/or sprite name is to be handled.

Let's look first at the cases where only text or a sprite is to be put into an icon (i.e. not both).

If the text length is no more than 11 characters (a sprite name always will be) **and the text or sprite will not change at all whilst the application is running**, then a 'direct' text/sprite name can be used. This means that the text/sprite name concerned will be an inherent and unchanging part of the icon design, stored as part of the icon definition

However, if the text is longer or the text/sprite needs to be changed whilst the application is running, then the icon must be defined as ‘indirected’. All this means is that the place where the icon definition would store the direct text/sprite name is taken instead by a pointer/buffer (a memory location) at which the text is held - and it can now be up to 255 characters long in any indirected text icon, although we need to define the maximum number of characters we wish to use in each case.

Now, whenever the Wimp displays/redraws the icon, it looks at the pointer location and extracts what it finds there. So, if you have changed the contents at the pointer location since the last time, the icon will show different text (or a different sprite).

If an icon is to show both text and a sprite, indirection is necessary and the sprite name is included in the ‘validation string1 - see Appendix 5.

The first icon we chose above was a writable one, so the text is certainly intended to be changed and needs to be indirected. No sprite is required. So, at the top of the `Edit icon` window, the first and third options need to be ticked but not the second one - and this should be the default state for this icon when the window appears.

A suitable maximum text length value also needs to be entered into the ‘Max size’ box, to the right of this section. The value needs to be one more than the maximum number of visible characters needed, to cater for a string terminator. *(If you ever try to exceed the defined maximum length of an indirected text icon, one of the first things that usually happens is that the text of your whole desktop display starts to revert to System font, although other effects can occur.)*

Border/fill/justification

Next, have a look at second section of the `Edit icon` window. This is pretty self-explanatory and does not need many words here. It offers several choices for how the icon and its text will look. The default options are acceptable this time i.e. a border, filled colour, text centred both ways.

(Later on, we give a hint that it helps to keep a border around an icon until the last possible moment in the window design process.)

Fonts

The next item to tackle is the font (and its colour) to be used for the text in the icon. If we want to use the System Font (or Desktop Font, with later machines) then the ‘Outline font’ option in the bottom section of the `Edit icon` window needs to remain unticked. In this case, choosing the colours of the text and its background is simply carried out via the right-hand side of the third section of the window. You can either use the ‘nudgers’ or move the pointer over the colour box and press any mouse button when the pointer changes to a menu symbol. This brings up the complete palette and you click on the one you want. You do this twice: once for foreground and once for background.

Note that the foreground colour is also used for the border of the icon (assuming that the border option is ticked) and the background colour will only be seen if the filled option is ticked.

Things get a little more complicated if we want to use an outline font. Choosing the font itself is easy: just tick the ‘Outline font’ option and choose your required font and size in the bottom section of the `Edit icon` window. In our case, we will use “Homerton.Bold” at 12 point. (Note that, by default, the text width and height are the same - but you can make them different values if you wish. The way to do this is self-evident in the `Edit icon` window.)

However, if we want colour as well as the outline font, then the ‘validation string’ needs to be used - see Appendix 5 to confirm the details used below.

Validation string

This subject is important enough to warrant its own appendix, so see Appendix 5 for details.

The validation string is simply entered into the writable icon at the top right of the `Edit icon` window. You will see that this window already contains the string:

```
    Ktar;Pptr_write
```

which is the default validation string chosen by `!TemplEd` for a writable icon. As it happens (see Chapter 7) we want to avoid any K-command in our example, so our first action is to delete this command.

However, we do want the pointer to change to the 'writable' symbol when it is over this icon, so we can leave the P-command as it is. (Try it out by moving the pointer over our new Icon 0 in the `NewCar` window, to see that it is working OK.)

To make the chosen outline font show text in red on a white background, we need to add the F-command:

```
    F0b
```

(The colours will default to Black on White - see Appendix 5.)

Further, also from Chapter 7, we want to restrict entries in this icon to the upper case letters, the space character and the digits 0-9 (i.e. for car registrations). Hence we need to add the A-command:

```
    AA-Z 0-9
```

(The space character has been put in the middle to make it more obvious.)

So, our complete validation string for Icon 0 is now:

```
    AA-Z 0-9;F0b;Pptr_write
```

Press 'Update' to effect this. You will see the colour change immediately but the entry restrictions will only take effect in the actual program window - have faith!

Initial text

At this stage, whether we have used an outline font or not, it is as well to have a look at the writable box to the right of the text option (to the left of and a little above the validation string writable icon). This allows us to enter any text that we want to appear in our icon as default. Obviously, as we are going to use indirected text in this particular case, we could also arrange any initial text from within the program. As it happens we do not want any default text this time so, when we are finished, this box will be left blank - as it is by default.

However, there are a couple of very useful facilities which we can use temporarily at this point. If we type into that writable icon the maximum size of text we are going to allow (and with wide characters) we will be able to check how this looks in the icon before we finalise things - and further, when we press ‘Update’, the text ‘Max size’ value will be adjusted correspondingly, if it is currently too low (a very helpful feature).

In the tutorial application we decided that a maximum of 10 characters would be allowed for the car registration, so type:

```
8888888888
```

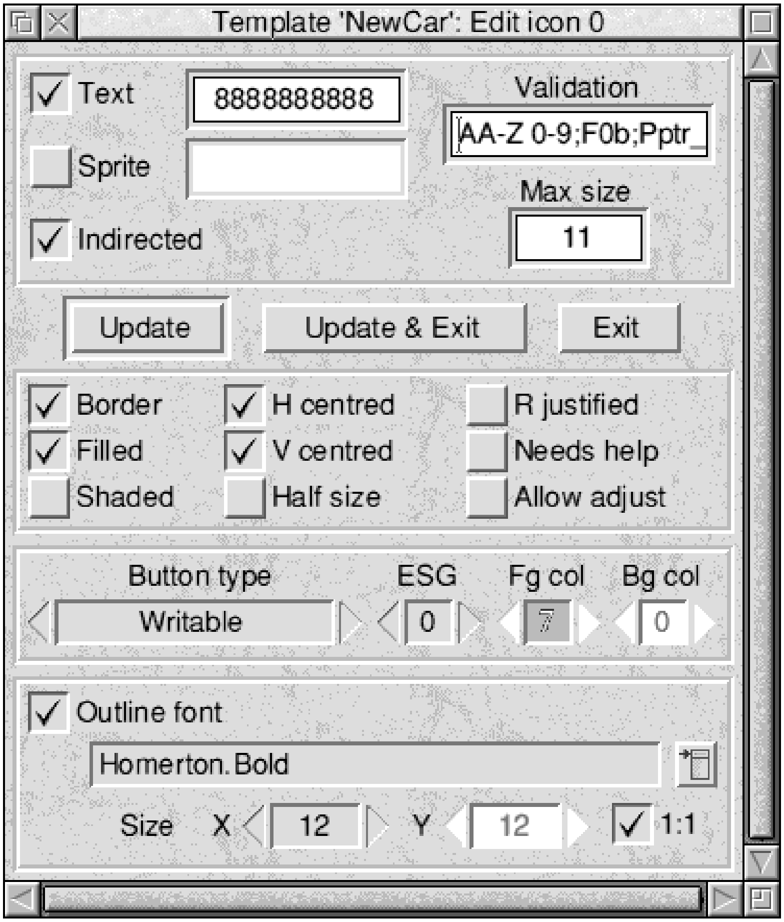
into the edit window top box and press ‘Update’. The digit 8 is fairly wide and so now a string of ten of them will now appear in our new icon and we can see straightaway that the icon is physically not large enough.

There are several ways to change the icon size. The quickest is to `<adjust-drag>` any of the sides of the icon (press and hold <adjust> until the pointer shape changes). You will soon get used to this: the particular side is the one nearest the pointer when you press and hold - it can be a little tricky with small icons.

Another way is to choose ‘Icon #0.Min. size’ from the ‘Window’ menu and choose one of the options ‘Both/X/Y’. The icon size will be adjusted to fit round whatever is currently the text in the icon - in either the x-direction or y-direction or ‘Both’ according to the choice you made. (With outline fonts, you may find that the result is still not quite right. In this case, you simply have to adjust the size a little by the dragging method above.)

(If you are using a modern computer but are releasing an application for others to use, then you should check that all your icons are large enough to display their text correctly if System Font is used. It only takes a few seconds to do this using the Configuration window.)

The following screenshot shows the `Edit icon` window at this stage:



Template 'NewCar': Edit icon 0

✓ Text 8888888888

Sprite

✓ Indirected

Validation: AA-Z 0-9;F0b;Pptr_

Max size: 11

Update | Update & Exit | Exit

✓ Border ✓ H centred R justified

✓ Filled ✓ V centred Needs help

Shaded Half size Allow adjust

Button type: Writable ESG: 0 Fg col: 7 Bg col: 0

✓ Outline font

Homerton.Bold

Size X 12 Y 12 ✓ 1:1

With Icon 0 now correctly sized, we can clear the `"8888888888"` from the icon and press ‘Update’. The icon definition is now complete.

Copying icons

If you look at the picture of the finished `NewCar` window (page 84) you will see that we need to end up with three similar writable icons (numbers 0, 1 and 2). We could achieve this by repeating the drag and drop from the Icon palette and going through the same procedure, but there is a better way to reduce workload.

Press `<menu>` over Icon 0 and ‘slide right’ over the ‘Icon #0’ and ‘Copy’ items. Then select one of the options ‘Up/Down/Left/Right’. This will create a new icon (with the next available icon number) which will be an exact copy of Icon 0 and will be placed above/below/ to the left/to the right of Icon 0, corresponding to your choice.

Note that this new icon is selected after creation, so if you had used the `<adjust>` button to keep the menus visible when making your copy position selection, you could simply have pressed adjust twice to get the required two copies of Icon 0. You would end up with a group of the three icons touching and they will be numbered 0, l, and 2 if you have followed the same sequence.

Icon position/alignment

We can now adjust the position of these icons. It can be done roughly by dragging the icons. It can also be done accurately by selecting all three icons (by the usual methods of `<select>` then `<adjust>`, or by dragging a box around them) and then following the ‘SelectionAlign’ and/or ‘Selection.Space out’ menu chains to get the `Align icons` (or `Space out`) window.

These are fairly self-explanatory and best explored by suck-it-and-see.

There are certain sensible rules which are followed, such as, with alignment for example, the icon whose top left corner is nearest the top and/or the left of the window is taken as the key icon. All other icons will be moved to meet the selected alignment.

Other rules soon become clear with practice and are detailed in the `!TemplEd` on-disc documentation.

Identical copies

Having got our three writable icons where we want them, we just need to remember that although they are physically identical they do need to have different A-commands in their respective validation strings - because Icon 1 is for dates and Icon 2 for a mileage number. You can peek at the `Templates` file in the tutorial application to see what changes are needed.

Just as important is the point that if you have used the `!TemplEd` icon-naming facility in the validation string (the N-command, see Appendix 5) any icon copying will have faithfully reproduced the string and the result will be more than one icon with the same name! Be warned!

The other text icons

To the left of each of the writable icons in the `NewCar` window is a text label and, in fact, these are also icons. They were started by dragging the 'Label' icon from the icon palette then manipulated in the `Edit icon` window to achieve the desired effect. In this case, no border and not filled gives the effect that the text is written directly onto the window. Note that the button type is 'Never' which means that it will not respond to clicks or drags in the program window: it is, as it says, merely a label.

The text used in each label is not going to change and is entered into the top box in the `Edit icon` window as default text. It needs to be indirected because it is too long (>11 characters) to avoid that. The label icon does, in fact, default to 'indirected' but, if it did not, it would automatically switch to 'indirected' if default text of 12 or more characters was typed in.

It is suggested that, as before, one of these label icons is created and then copied twice. The text can then be altered in the copies.

The long icon to the bottom left of the `NewCar` window is also a label, but it has been given a border and a fill colour to associate it visually with the two icons to its right - the next two we need to deal with.

The ‘Clear all’ and ‘Create file’ icons are ‘Menu’ type buttons. That is, they select/deselect themselves as the pointer moves over them.

There isn’t one of this button type in the `!TemplEd` icon palette, so you need to create it by dragging one of the other types (of a similar look) and then change the button type (as explained earlier) in the `Edit icon` window. You can do it either by ‘nudging“ the current entry or (more useful) by selecting from the menu which appears when you press `<select>` over the icon labelled ‘Button type’.

Note that in this application these two menu type icons each need to be in their own ESG for their menu select/deselect properties to work properly. ESGs 1 and 2 have been used here.

Don’t forget that any icon created from the icon palette automatically defaults to ESG 0.

The sprite icon

The final icon to look at is the decorative one which shows the petrol pump sprite in the top left corner of the NewCar window. This is achieved following the same procedure as for other icons, except that the ‘Sprite’ option is ticked and the sprite name (not sprite file path!) is entered into the writable box to the right of the tick. (You would also need to tick ‘Indirected’ if you wanted to change the sprite during the program run.)

The only point to watch - and it is very easy to forget - is that the sprite you want to display must have been ‘seen’ by the Wimp - typically by ensuring it is in a sprite file which is included in an `*IconSprites` call in your application’s `!Boot` and `!Run` files (and your application has been ‘seen’!). Don’t forget that you will almost certainly need at least a `!Sprites` and a `!Sprites22` version for the display to occur in all screen modes.

When the sprite is (finally?!) displayed you can size the icon to fit it properly and that is all there is to it.

Icon numbering

If you have used the sequence described the icons will probably be numbered as follows:

0-2	- the three writable icons, from top to bottom
3-5	- their corresponding labels
6	- the long label icon at the bottom left
7	- the 'Clear all' menu button
8	- the 'Create file' menu button
9	- the sprite icon

Check this with the `Icon info` window, as described earlier.

In many programs, the programming is very much easier if groups of icons have sequential numbering - for using `FOR ... NEXT` or `REPEAT ... UNTIL` loops for instance, or for getting the caret cycling logical. So an ability to renumber icons is very important in a template editor and `!TemplEd` has very helpful facilities to do this.

Altering the number of a single icon is achieved by following the 'Icon #n.Renumber' menu trail and putting the required new number into the writable box and pressing `<return>`.

To alter the numbers of several icons together (to number them in sequence), the same steps are followed - this time 'Selection.Renumber', of course - and then enter the new **starting** number required.

However, this bald description is not sufficient, because you need to be aware of certain rules which come into play with renumbering.

Firstly, every time you create an icon it is given the lowest available icon number. This does not always mean the next number in the sequence - because you may have created an icon earlier and then deleted it. In this case, a new icon is given the lowest available number of a deleted icon.

(`!TemplEd` never leaves any numbering gaps at the top of the numbering sequence. So if you delete the icon with the highest number it will not be registered as a deleted icon. Further, if there was a numbering gap immediately below it, then this too will be eliminated at the same time.)

This means that it is possible (probable, perhaps) that you will complete your icon designing and end up with gaps in the numbering. Unless you realise what has happened, this can play havoc with the nerves when renumbering!

To guard against this problem, follow the ‘Misc’ menu item from the ‘Window’ menu. This brings up a menu with four items on it and it is ‘Deleted icons’ (the last one) we are interested in, If this is ‘greyed out’ then you have no deleted icons registered. If it is not greyed out then you have deleted some icons - and following the item across will give you another menu with two options to undelete them and one option to ‘Purge’ them i.e. get rid of them for good. If you opt for this latter choice, the remaining icons will be renumbered from 0, eliminating any gaps.

It is always prudent to make this check before ‘signing off on icon design and before any renumbering.

Once you have eliminated the gaps (or, at least, are aware of them) you can renumber with confidence.

Note that when you choose a group of icons to renumber in sequence, they are renumbered in an order **which is determined by their position in the window**. The order is akin to calculations on a spreadsheet: the ‘topmost and leftmost’ setting the final numbering order. It is easy once you get the hang of it and for windows with many icons you will probably find yourself renumbering successive small groups of icons rather than the whole lot at once.

`!TemplEd` is very helpful when renumbering groups of icons successively, because it sets the default next starting number to be the one after the renumbering sequence you have just completed - which is usually what you want.

The final rule about renumbering is that you can’t get something for nothing. If there are 10 icons numbered 0-9 and you renumber one of them, you cannot renumber it to a higher number than 9. Further, if you renumber, say. Icon 7 as Icon 2, then the previous Icon 2 will be renumbered as Icon 7 i.e. numbers are swapped.

Selecting groups of icons to edit

To carry out the icon-editing actions on groups of icons e.g. renumbering, it is necessary to select all the appropriate icons first. This can often be done by the usual method of using `<adjust>` after the initial `<select>`. However, this will not work if you have set up an ESG group - as soon as you select another icon the already- selected one is de-selected!

The way round this is to <select drag> a box around the icons concerned. On release, all the enclosed icons will be selected.

Leave the icon border on!

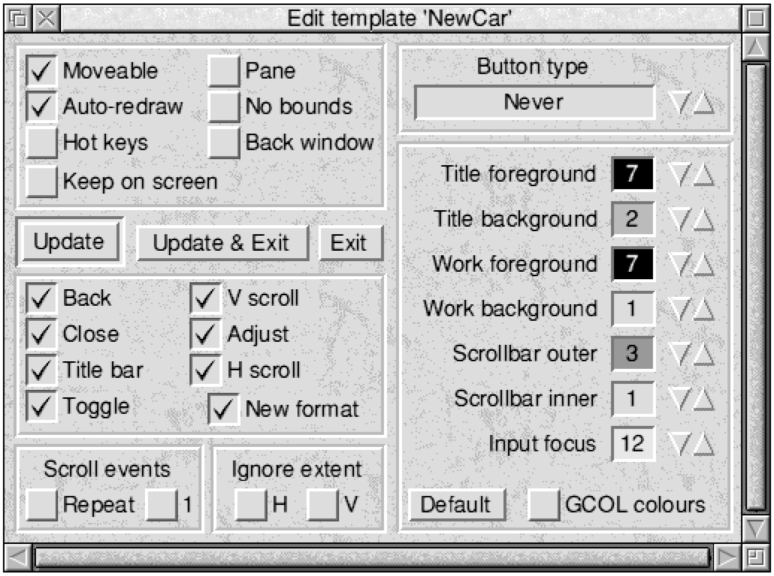
One final overall tip when designing icons: if you intend an icon to have no border - and even more so if it is also going to be unfilled - put a border around it until you have got everything else to your liking, then finally un-tick the border option. Without a border, it is very easy to make an icon invisible on the screen, which can be a nuisance and can even lead to it being forgotten (and later mystifying you when the icon numbering goes awry - see below).

Window editing

Now that all the icons have been created, we can return to the window itself.

At this stage, we can now sensibly look at the window size and what window controls we want around its edges e.g. scroll bars, close icon etc. *(These features are sometimes referred to as the window 'furniture'.)*

All these items are set from the Edit template window (see below) or via the lower portion of the 'Window' menu - and they are mostly self-explanatory.



Edit template 'NewCar'

✓ Moveable
Pane
✓ Auto-redraw
No bounds
Hot keys
Back window
Keep on screen

Update | Update & Exit | Exit

✓ Back
✓ V scroll
✓ Close
✓ Adjust
✓ Title bar
✓ H scroll
✓ Toggle
✓ New format

Scroll events: Repeat, 1
Ignore extent: H, V

Button type: Never

Title foreground	7
Title background	2
Work foreground	7
Work background	1
Scrollbar outer	3
Scrollbar inner	1
Input focus	12

Default | GCOL colours

Note the difference between the phrases "Window name" and "Window title". The former is the name of the window template which appears in the template file window (usually at the top left of screen) and is referred to in the SYS call (or the Dr Wimp wimp-function) when loading the template into a program. The name is editable via the `'TemplEd'` menu.

The latter is merely the text that appears in the title bar of a particular window. This is editable via the 'Edit title' item of the 'Window' menu - and note that it can be made indirected if you wish (essential if you are going to use the usual asterisk to indicate unsaved data - in which case, don't forget to allow for the space-plus-asterisk in the maximum number of text characters).

If you untick 'Auto-redraw' in the `Edit template` window, as you must do if you intend to use the redraw method for graphics etc. (see Chapters 11 and 12) then, on pressing 'Update' you will see that the template window is cross-hatched, to alert you to this. It does not appear in the program window, of course.

Note also that you can give a window (i.e. its background area) a button type. This is essential if you want the window itself to respond to mouse clicks, file drags, etc.

You may decide not to have a Close icon in a window. In which case, when you want to close that window in the template editor, use the 'Close window' item from the 'Window' menu. (In the program, of course, you will then have to take steps to open/close such windows in response to other user actions.)

Templates file window appearance

When you alter some of the window 'furniture' characteristics e.g. untick one of the scroll bars, you will notice that the small icon representing that window in the templates file window (normally at the top left of the screen) will change correspondingly. This gives a quick visual indication of the style of each window defined.

Default window/icon states

Finally, it is worth noting that when you load and display your window template in your application, it will appear in exactly the state that it was when you last saved it in the template editor. So, for instance, any icons which were ticked or otherwise selected when you saved it in the editor will appear as the default settings.

You may find this helpful, but it is probably better programming practice to ensure that all icons etc. are unselected in the template editor before saving them and then set deliberate default states in the program.

Final comment

As was said at the start of this Appendix, the above descriptions are not a comprehensive guide to all the **`!TemplEd`** features: rather, it is a good practical introduction to get you started with the most frequent actions. The **`!TemplEd`** on-disc manual gives a much deeper insight - and Chapter 16 covers one particular other feature - the ‘Statistics’ option.

You will also probably find that other window template editors operate in very similar ways for their basic actions.

Appendix 8. User-functions

Whenever you use Dr Wimp, whatever the version, your `!RunImage` will always (need to) contain all the user-functions relevant to that version - even if you are not using them i.e. you have left them ‘empty’.

The vital point to note about all user-functions is that you, the programmer, never call them. They are called automatically when needed by the DrWimp library - usually passing you important information in their parameters. Your role is to fill the user-function `DEF`s with the code necessary to meet your needs - using the passed information as needed and mainly by calling the wimp-functions.

Because the concept of user-functions is what makes Dr Wimp so powerful, it is worthwhile running fairly quickly through several of them to ensure that their purpose and working is understood. *(More detailed information often occurs in other chapters under specific topic headings.)*

But don’t forget that user-functions can change from one Dr Wimp version to another and you should always refer to the Dr Wimp Manual for the details of the version you are actually using.

The following comments apply to the user-functions as they existed in Dr Wimp Version 3.80 (31st March 2003).

```
DEF PROCuser_initialise
ENDPROC
```

Called by `FNwimp_initialise`. Should be used in all applications. Intended to help keep applications in a well-structured format. It should contain anything that is necessary before wimp-polling starts. Generally, that includes all window definitions/loading, initial menu definitions, global variables. Dimensioning of arrays, parameter blocks, etc. should usually be in here.

```
DEF PROCuser_error
ENDPROC
```

Only called in standard global error line of the skeleton `!RunImage`. It is used to implement 'good housekeeping' actions which may be needed when a 'fatal' error occurs e.g. closing files.

```
DEF FNuser_quit(type%)
=1
```

Called by various Wimp poll routines in the `DrWimp` library. It should be used in all applications. It ensures an orderly exit from the application, whether initiated by user choosing 'Quit' from the application or by a desktop shutdown.

`type%` is 0 if the quit action was to exit the application, or 1 if a shutdown was started.

Return is left as (the default) 1 if quitting/shutdown is to continue, or changed to 0 if quitting/shutdown is to be halted e.g. to allow saving of unsaved data.

```
DEF PROCuser_redraw(window%,minx%,miny%,
                   maxx%,maxy%,printing%,page%)
ENDPROC
```

Called by `DrWimp` when the Wimp calls for a redraw. It is used in all applications that require the program (rather than the Wimp) to generate text/graphics and for wysiwyg printing i.e. generally for any items that are not contained in icons. The required plotting action is placed in this user-function. (See the tutorial application.)

The window definition must have its 'auto-redraw' option deselected for this function to work.

The parameters actually define the precise rectangle which the Wimp is asking to be redrawn, thus paving the way for more efficient redraws. The last two parameters indicate whether printing is under way and, if it is, which page is currently being printed.

```
DEF PROCuser_mouseclick(window%,icon%,button%,
                    workx%,worky%)
ENDPROC
```

Called by `DrWimp` every time the `<select>` or `<adjust>` button is clicked in a window of the application. The parameters give the window/icon over which the click occurred, which button and the pointer coordinates. Responses to mouse clicks are placed in this user-function. It is used in probably every application - including the tutorial application.

```
DEF FNuser_menu(window%,icon%)
= 0
```

Called by `DrWimp` whenever the `<menu>` button is pressed over a window/icon of the application, which latter are identified by the parameters. The program simply needs to return the handle of the menu then required to be opened. (Used in tutorial application.)

```
DEF FNuser_keypress(window%,icon%,key%)
=0
```

Called by **DrWimp** each time a keyboard key is pressed (if a window in your application has the input focus). The parameters give the window/icon having the caret and (according to use of the K-command in the icon's validation string - see Appendix 5) can also give the ASCII code of the pressed key. (Used in tutorial application.)

All Wimp applications need to handle keypresses one way or another and the **DrWimp** library does the necessary behind the scenes. If you want to use the keypress to trigger some program action then that action is placed in this user-function and the return value must then be changed to 1. Unused keypresses should return 0.

```
DEF PROCuser_menuselection(menu%,item%,font$)
ENDPROC
```

Called by **DrWimp** each time an item is selected from a menu, with the parameters giving the menu handle and item chosen - plus the font if the selection was from a font menu. (See Chapter 17) Responses to the selections are placed in this user-function.

It is used by nearly all applications including the tutorial application.

```
DEF FNuser_savefiletype(window%)
= ""
```

Used in conjunction with the two user-functions following below and called by **DrWimp** in several circumstances which might involve a save action. It identifies which filetype is to be used with a standard Save window. When the parameter is a save window, the return is simply set to the filetype required; see Chapter 18.

```
DEF PROCuser_saveicon(window%,RETURN drag%,RETURN
            write%,RETURN ok%)
ENDPROC
```

In effect, this is a small utility routine to allow the programmer to specify the icon numbers of the three icons in a standard Save window - the ‘drag’ icon, the writable icon and the ‘OK’ button. **`DrWimp`** will assume that these are icons 0, 1 and 2 respectively unless this user-function says differently; see Chapter 18.

```
DEF FNuser_savedata(path$,window%)
LOCAL ERROR
ON ERROR LOCAL =2
=1
```

Called by **`DrWimp`** when saving action needs to be taken e.g. when the draggable icon from a Save window has been dragged to a directory. The Save window handle is passed as a parameter, as well as the full path of the destination to which it was dragged. (See Chapter 18.) The precise saving action is put into this user-function and a 1 is returned.

The unusual default state of this user-function is necessary to allow **`DrWimp`** to invoke the correct error messages when a drag to, say, a write-protected floppy disc is made.

```
DEF FNuser_loaddata(path$,window%,icon%,ftype$,
            workx%,worky%)
=0
```

This complements **`FNuser_savedata`** shown above. It is called by **`DrWimp`** whenever a file is dragged onto a window of your application - giving the window and icon handles dragged to, the path and filetype of the dragged file and the position (in work area OS coordinates) where the file was dropped. If a file is double- clicked instead of being dragged, the final two parameters will both be -1. If you decide to load the file, the loading action is put into this user-function and a 1 needs to be returned.

```
DEF PROCuser_null
ENDPROC
```

Called by `DrWimp` if the global variable `NULL%` is set to `TRUE` and Reason Code 0 is received. It is one of the means by which timed operations and 'multi-tasking' are facilitated. (See Chapter 24.)

```
DEF PROCuser_print(minx%,miny%,maxx%,maxy%,page%)
ENDPROC
```

Called by `DrWimp` whenever a page needs to be printed, if 'user printing' method is active; see Chapter 21. It gives rectangle on page to be printed (in paper coordinates) and page number.

```
DEF FNuser_printing(copy%,page%,
            totpages%,pagepos%)
=0
```

Called repeatedly by `DrWimp` during printing, so as to keep the user informed of current progress - passing current copy/page etc. It is typically used to update visual progress indicator. If a 1 is returned the printing is cancelled.

```
DEF PROCuser_printerchange
ENDPROC
```

Called by `DrWimp` whenever a printer driver is changed/modified, in order to allow user to update page measurements, printer name etc. (Typical use demonstrated in tutorial application.)

Remember!

There are many more user-functions and they are subject to change with new releases of Dr Wimp.

All user-functions relevant to a specific Dr Wimp version must always be present in the !RunImage file.

Appendix 9. Wimp-functions

The wimp-functions are subject to much more change than user-functions and therefore any descriptive detail in this section would soon become out of date.

Full details of all available wimp-functions are always listed in Section 3 of the Dr Wimp Manual corresponding to the Dr Wimp Version involved.

Appendix 10. Wimp Messaging system

(Not to be confused with application 'Messages', as covered in Chapter 26!)

It is not necessary to give details of the Wimp Messaging system in this introductory book, but it is useful to provide a brief non-technical description.

The purpose of the Wimp Messaging system is to allow the Wimp to pass useful management messages to/from a desktop application. Some are 'broadcast' to all applications and some are for individual applications.

For example, when an application wishes to print some output on a printer, the application would normally wish to devolve the actual printing task to a printer driver loaded by the `!Printers` application.

To do this, RISCOS has devised a Messaging system to allow, for instance, an application to advise that it wants to send data to be printed; for the `!Printers` application to acknowledge this and concur; and for the data transfer to take place.

Naturally, the messages and the protocols are strictly defined and will only work if all applications keep to the same rules.

The notifications to applications that these messages have arrived etc. are carried out via the Wimp Poll process, specifically by Reason Codes 17, 18 and 19.

Perhaps the most important practical point to note is that it is essential for each application to enable the Wimp Messaging system in its Wimp initiahsation process. If this is not done the Messages will not be passed to the application. *(This may not prevent the application from running, but it will not be able to use Reason Codes 17, 18 or 19.)*

Some of the topics which are carried via the Messaging system are:

Quit e.g. as a result of a Shutdown action.
Saving a Desktop boot file.
Opening/closing file directories
Screen Mode change
Desktop saving and loading
Printing via !Printers
Interactive Help
Colour picker
Sub-menu opening

Index