DrWimp

Cures your desktop problems!

Version 3.01 (08-Jun-97)

Manual laid out & mostly proof read by Eddie Lord

Andrew Ayre
24 Skirbeck Rd
Gillshill Rd
Hull
East Yorkshire
HU8 0HR
England

Conditions of use

DrWimp is distributed on an “As Is” basis, without warranty. No liability can be accepted for any consequential loss or damage, however caused, arising from the use of this program or other utilities supplied. Boring but there you go.

The DrWimp library may be distributed for free and without the documentation, examples, utilities, etc. if it is being used in a Public Domain, Shareware, Cover Disc or Disc Magazine program. If you wish to distribute it with a commercial program then contact the author to negotiate royalties.

Public Domain libraries may make a reasonable charge for materials, handling, etc. as long as this does not exceed £2.00 (UK) net for this DrWimp package.

The DrWimp library may be reproduced in part if crunching and mangling utilities such as !BSquasher and !MakeApp are used, otherwise it must be reproduced in whole, complete with the conditions of use and the copyright banner.

If the DrWimp library is being reproduced in full then it may be added to the !RunImage (or similar) file, and does not have to be separate.

The DrWimp package (apart from the 3rd party utilities as detailed in the !!ReadMe!! file) may only be distributed as a whole. For conditions of use of the third party applications see their own !Help files.

For distribution conditions of the DrWimp package as a whole and the files included in it which are not 3rd party, see the !!ReadMe!! file.

The author retains copyright of DrWimp, documentation and examples at all times.

Documentation and utilities can be obtained from the web page below or direct from the author at:

E-mail:	A.J.Ayre@e-eng.hull.ac.uk
World Wide Web:	http://whirligig.ecs.soton.ac.uk/~aij295/groover

Help on any aspect of DrWimp (and the documentation + utils if you include a DD disc) can be obtained from the author at:

Snail mail: Andrew Ayre
24 Skirbeck Rd,
Gillshill Rd,
Hull,
East Yorkshire
HU8 0HR
England
E-mail: A.J.Ayre@e-eng.hull.ac.uk

Contents

Section 3 - Functions

Section 4 - Index

Section 1 Introduction

Prologue

Ever wanted to write high quality multitasking programs? Can’t quite get the hang of manipulating words, bytes, menu structures and message systems? Have you looked in despair at the amount of programming needed to get a single window on the desktop?

Forget all your worries and write superb applications with great ease; Doctor Wimp is here!

DrWimp - solves all your multitasking worries!

Banish that dodgy polling code to your disc box! Wipe that broken window redraw program from your hard drive! Drop that tired out menu creator into your null: device!!

The DrWimp system consists of:

- The DrWimp library.
- A template blank application.
- This manual in Impression Publisher and text format.
- Text files with the history, upgrading and security information.
- Example Template files for standard windows.
- !Fabricate for quick blank template application construction.
- Support files for the tutorials.
- !Func’n’Proc quick browser.
- !Linker BASIC library linker.
- !CodeTemps for converting windows to code.
- Various PD utilities for compression and code security.
- Example applications written using DrWimp, with fully commented code.

DrWimp - THE system to use!

Getting started

In this manual all functions (FN) and procedures (PROC) will be referred to as just functions.

The file DrWimp does all the work, and the !RunImage file is where you control things from.

DrWimp is just a collection of functions that you can call from the !RunImage file. All the functions in DrWimp are in lower case and preceded by “`wimp_`”, eg:

```
PROCwimp_dosomethingamazing(curtain$)
```

Some of these functions need help from you, so they call a function that is in the !RunImage file, where you can easily tailor it to your applications needs. All these functions are also in lower case, but they are preceded by “`user_`”, eg:

```
FNuser_givemesomehelp(tomato$)
```

For a “blank” application which does nothing all the “`user_`” functions will be empty, and return default values if they are of the “FN” type. These functions have to be there otherwise your application might complain that one of them can’t be found, regardless of whether they do anything or not.

From now on functions in the !RunImage file preceded by “`user_`” will be referred to as “user functions”, and functions in the DrWimp file preceded by “`wimp_`” will be referred to as “wimp functions”.

To save you having to type in a standard set of user functions every time you write an application, there is a template application called “!MyApp”. Inside !MyApp is a template !RunImage file. This file calls a function that starts up your application, because you can’t do anything multitasking until you call it. You will have to change the details of this call to suit your application.

The utility !Fnc’n’Prc (supplied in the “Utils” folder) is a catalogue of all the user and wimp functions. It details what parameters you have to pass and what is returned. It is also a demonstration of the DrWimp system, as it was written using it, and in particular the ability to easily add panes to windows and saving files.

The source code is included, and is fully commented to help you further understand how to get DrWimp to do what you want.

It may seem that most of the functions have far more parameters than is needed. This is true in that most of the time the parameters will be largely the same, but having more parameters gives you, the programmer, as much flexibility as possible.

The main difference that multi-tasking programs have over the others, is that they are not linear. The program doesn't start at the top and work its way to the bottom. Instead, multi-tasking programs jump about depending on whether an icon has been clicked on, or a window dragged, or a menu item chosen, etc. All your application has to do is respond to these "events" when they happen.

Handles

DrWimp expects windows to be loaded in from a templates file. Menus are created from a shorthand in the !RunImage file. Once you have loaded in a window or created a menu you need some way of getting at the information so you can put the menu on the screen, or change some of the window attributes, etc. The way this is done is by handles, and they form a very important part in programming the wimp.

When you load in a window, DrWimp asks the machine to reserve some memory for it to put all the information about the window into. It does this using the DIM command, eg:

```
DIM block% 256
```

This will reserve a chunk of memory 256 bytes long, and the first memory address in that chunk is in block%. This is so you can find it and change the contents of the chunk of memory.

So if you load in a window called "save", then you might reserve a chunk of memory called save%, so you can easily see that the memory address save% is the start of the information about the save window, you loaded in when the application first started.

The handle to the save window is therefore save%.

The same is for menus. When you ask DrWimp to create a menu from the shorthand that you supply, it gets a chunk of memory and fills it with the information needed for the wimp to create the menu. eg: if you were creating a menu for the iconbar icon of your application, then you might call the handle something like barmenu%.

In general it is very wise to call the handle something that reminds you of what it is a handle for. If you load in five windows, then it would be a bit stupid to call the handles a%, b%, c%, d%, e%!

You will find that most wimp and user functions require a handle to a window or a menu to be passed to them, so you will need to use them a lot.

Look at the following piece of code (the functions will be described in more detail later on):

```
    find% = FNwimp_loadwindow(“Templates”,“find”,0)
    PROCwimp_openwindow(find%,1,-1)
```

The first line calls a wimp function that loads in a window from a templates file called “find”, so we can assume that it is a window for entering something to find. The function returns a handle to the window, which has been called find%. From now on, whenever we want to do something with the find window, we use the variable find%. Look at the second line. Here a wimp function is being called that opens a window. Passed to it is the window handle, telling it which window to open (and some numbers regarding the positions to open it at).

The handle could have been called anything, eg:

```
    coffee% = FNwimp_loadwindow(“Templates”,“find”,0)
```

Or changed later on, eg:

```
    find% = FNwimp_loadwindow(“Templates”,“find”,0)
    coffee% = find%
```

Variables & Strings

One of the first lines of the !RunImage file should be something like:

```
    LIBRARY “<MyApp$Dir>.DrWimp”
```

This tells the BASIC interpreter that you want to use a library and where it is. After this call, the two files (DrWimp and !RunImage) may be separate but they work as if all the code is in one file. Eg: you can call functions in DrWimp from !RunImage and call functions in !RunImage from DrWimp.

More importantly, any variables or strings that are created in one of the files and isn’t localised, will be available to both files, so either one can alter the contents. This can cause problems, because you could be using a variable in the !RunImage file, and then call a wimp function that also uses it, so when the function is exited, the variable will be different from what your code expects.
This problem has been largely overcome by localising most of the variables used in DrWimp. However, there are a handful of variables that cannot be localised for one reason or another.

All but two of the variable names start with a lower case “w”, so you can easily choose variable names that don’t clash: simply don’t start them with a “w”.

The variables whose names start with “w” should not be changed. The other variables finished% and NULL are supposed to be changed:

By setting finished% to TRUE, the application will quit as soon as possible. This is how the Quit menu option on the iconbar menu works.

By setting NULL to TRUE, the user function `PROCuser_null` will be called every time the wimp passes control to the application and nothing is happening with it, eg: clicking on icons, dragging windows, receiving messages, etc. This is useful for timed or delayed things. You keep track of the time and after a certain delay you change a windows contents (say for a clock) or send a message to another application.

If you wanted something to happen continuously like animation then that it how you would do it.

System variables

It is important that you make sure that your application can run from anywhere, eg: floppy disc, hard drive, CD, etc, but you have to access files in the application directory, eg: templates, sprites, message files, etc.

System variables differ from BASIC variables in that System variables are set from the command line using a star command. Unfortunately it isn’t very easy to read the contents of a system variable from BASIC programs. Because of this, DrWimp provides a wimp function (`FNwimp_sysvariable`) to do this for you, and it will be described in more detail later.

When the !Run file of your application is run it sets up a system variable called “Obey$Dir” which contains the full pathname to your application directory. So if you have a file called “Templates” inside your application directory, then its full pathname can be written as `<Obey$Dir>.Templates`.

But if between your !Run file being run and your application loading in a file, another !Run file from another application is run, Obey$Dir will change. So in your !Run file you must make a copy of Obey$Dir that can be used instead.
If your application is called “MyApp” then you would use:

```
     Set MyApp$Dir <Obey$Dir>
```

and then you would get to your “Templates” file inside your application directory by using: `<MyApp$Dir>.Templates`.

System variables can also be used for other things. Eg: if you wrote a database application then you might have to set the maximum number of records allowed for the amount of memory that you have allocated. The limit could be put in the !Run file instead of the !RunImage file so non-programmers don’t have to go and delve into your code. So you could have a line in your !Run file like:

```
Set MaxRecords 5000
```

Then in your !RunImage file you could read the value with something like:

```
MaxAllowed% = VAL(FNwimp_sysvariable(“MaxRecords”))
```

Security

If you are going to distribute a new and brilliant application that you have slaved over for weeks, then you are going to want some security.

There are many ways in which you could give your code more security, and I have found a method which I think does the job perfectly.

As an example the total size for the source code for !Linker (!RunImage + DrWimp) is 55611 bytes (54.3k). After being compressed and mangled the resulting (and much more secure) !RunImage is 8732 bytes (8.6k).

All the utilities to do this are included with the DrWimp package and the procedure (as reproduced in the “Security” file) is:

1. Link your RunImage with DrWimp using !Linker. Store your original RunImage somewhere safe.

2. Compress your RunImage using !BSquasher.

3. Turn the BASIC RunImage into an absolute file by dropping it onto !MakeApp2 and saving under a different name.

3. Compress the RunImage by dropping it onto !Crunch, to give you your final RunImage to distribute.

Section 2 Tutorials

Notes

I would highly recommend using !TemplEd for editing/creating template files. It is very easy to use and it is included with DrWimp. Full instructions can be found in Manual inside the !TemplEd directory. However you can use any template editor you wish.

The tutorials use some support files and pre-made templates files. They can be found inside the Tutorial directory. The template files contain standard windows like Info and Save windows. Feel free to use these in your programs (PD or commercial). There are no conditions attached to them.

Before you start, make a copy of the "blank" application !MyApp to work on. Put it on a fresh disc or in a new directory.

If you get stuck or come across a problem then please, don't hesitate to get in touch. Your information could benefit other users.

Most learning is done through experimentation. In most of the tutorials, you will be invited to fiddle and generally muck about with the code. The worst that can happen through doing this is your application crashes and is quitted by the Task Manager.

To help you to understand further how to use DrWimp, some applications are supplied with the DrWimp system. They have fully commented !RunImage files so you can study them.

A short list of what each demonstrates is given in the "!!ReadMe!!" file.

1. Getting going

Double-click on !MyApp. Nothing should happen. If you now open the task manager display and look at the list of tasks, you should see “MyApp” at the bottom.

Load !RunImage into !Edit and take a look at how it works. The !Run file copies the system variable Obey$Dir into MyApp$Dir. The LIBRARY command can then be used with the system variable.

The application name is put into a string to make it easier to change later on. The next line is in case of an error. All it does is call PROCwimp_error. This wimp function will be looked at in more detail later on.

The next line is very important and your application can’t do anything until it has been called. It registers your application with the task manager and it returns a handle for your task. This has been put into task%.

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

For the moment the second and third parameters are just set to largish values. When the application is finished you can reduce them to as low as possible with the application still working fully. Basically they are (from left to right) sizes of areas to reserve for window definitions (including all the icons in the window), and the indirected text (like large menu entries and long window titles). The first parameter is the text that appears in the task manager window, and can be anything. Try it and see!

- Note: the wimp automatically truncates it if it is too long. The last parameter is the minimum version number of RISC OS that the application is allowed to run on multiplied by 100. The default is 300, so !MyApp will run on RISC OS 3.00 or better.
- Note: the version number returned by the WIMP to DrWimp isn’t always very accurate. For example, if you want to set the minimum version to RISC OS 3.11, then set the last parameter to 316! This is not the fault of DrWimp.
- Note: don't add any lines before the FNwimp_initialise line, as this could cause problems with the error handling.

The final line calls PROCwimp_poll which makes it multitasking. This is a continuous loop until your application quits for some reason, such as “Quit” being chosen from the iconbar menu.

The rest of the !RunImage file is all the user functions. They are all empty at the moment and do nothing or return default values.

It doesn’t do much at the moment and there is nowhere for the user to get access to our application. The best thing to do is add an icon to the iconbar.

After FNwimp_initialise, add the following line:

```
bar%=FNwimp_iconbar(“!myapp”,“”,1)
```

The function returns a handle to the window that contains the icon and where “!myapp” is the name of the sprite to use for the icon.

Try changing it to “!draw” and see what happens. The next parameter is the text to place under the icon (like the floppy drive icon). If it is an empty string then no text is printed. Try entering some text and reload the application. The final parameter controls the side of the iconbar that the icon appears on. 0 for left and 1 for right.

2. Menus

What is needed now is a menu for the iconbar. Menus are created by DrWimp from a shorthand form which is passed as a string. DrWimp translates it into the correct layout in a block of memory for the wimp to understand. After the FNwimp_iconbar, add the following line:

```
barmenu%=FNwimp_createmenu(“MyApp/Info/Quit”,0)
```

This creates a menu and returns a handle for it, which is put into barmenu%. Each entry or item on the menu is separated by a “/”. The first item is the menu title, so from the line just added, a menu will be created with the title “MyApp” and two entries. The top one is “Info” and the bottom one is “Quit”. The last parameter is the maximum number of items allowed to be used. It is set to 0, so this means that just allow space for the items specified.

At the moment “MyApp” doesn’t know about the menu, all we have done is set it up in memory. Whenever the Menu button is pressed over a window belonging to the application, DrWimp calls the user function FNuser_menu. Passed to it is the window handle (in window%) and the icon number (in icon%). If there is a menu for that window/icon then we have to return the handle to it, otherwise return a 0.

Move down to FNuser_menu and change it so it is like:

```
DEF FNuser_menu(window%,icon%)
CASE window% OF
WHEN bar% : =barmenu%
ENDCASE
=0
```

As you can see, whenever the window whose handle is bar% (the window containing the iconbar icon) has menu pressed over it, the handle to the iconbar menu is returned. Try running the application.

As you will see from running it, Quit doesn't do anything. When a menu item is selected, DrWimp calls another user function: PROCuser_menuselection. This time nothing has to be returned, you only have to act on the selection made by the user. Passed to the function is the handle of the menu that the user chose an item from (menu%) and the number of the item (item%). The top-most item is number 1, so for Quit:

menu% = barmenu% and item% = 2.

Add the following to make PROCuser_menuselection look like:

```
DEF PROCuser_menuselection(menu%,item%)
CASE menu% OF
WHEN barmenu% :
  CASE item% OF
  WHEN 2 : finished%=TRUE
  ENDCASE
ENDCASE
ENDPROC
```

Recall from Section 1, setting finished% to TRUE quits the application as soon as possible. Run the application and confirm that it does indeed work.

Try adding some more items to the menu. Don't forget to increase the "2" in PROCwimp_menuselection accordingly though! Try getting the computer to make a beep when you select one of the items. `VDU7` would be ideal for this. Note: there is no need to change the last parameter to FNwimp_createmenu (that parameter is explained later).

Menus may also be created automatically from message files. See the section on Messages for details.

3. Windows

What is needed now is an info window that can be accessed from the iconbar menu. In the tutorials folder you will find the file "Template1". Copy this into the !MyApp application directory and rename as "Templates". You can examine it if you want by loading it into !TemplEd.

Add the following line below FNwimp_iconbar (above FNwimp_createmenu):

```
   info%=FNwimp_loadwindow(“<MyApp$Dir>.Templates”,“info”,0)
```

This function loads in a window from the templates file, whose full pathname is given in the first parameter. The second parameter is the name of the window in the templates file, and the last parameter tells DrWimp where to get the sprites from for the window. A 0 means use the wimp sprite pool (RMA), and this is the most common option. A handle to a sprite area instead would mean use that sprite area (as well as RMA sprites), but there are no sprites (user or RMA) in this window, only the borders built into RISC OS 3. so the last parameter is set to 0.

The function returns a handle to the window which we have put into info%.
Now we need to modify the iconbar menu to take into account the window. This is very simple to do and it is mostly done automatically. All you have to do is add the following line after the FNwimp_createmenu:

```
     PROCwimp_attachsubmenu(barmenu%,1,info%)
```

The first parameter is the menu handle to attach the submenu to, the second parameter is the item on that menu to attach the submenu to (in this case 1 which is the top item), and the third parameter is either a handle to another menu or a window handle. In this case it is the handle to the info window.

And that’s it! Re-load !MyApp and you will now see that the Info menu item now has a submenu which leads to the info window.

You will see in the info window that the Author and Version fields don’t have the correct information in them. This is very simple to fix, and the same technique that I am about to describe can be used for any icon that has text, whether it is a Radio button, a default action button, or just a comment (if it is indirected, see !TemplEd).

First of all the Author field. This is icon number three. You can find the icon number by loading the Templates file into !TemplEd, opening the info window and moving the pointer over the icon. The small window at the top right will give you the icon number. Add the following line just before PROCwimp_poll:

```
     PROCwimp_puticontext(info%,3,“© Joe Bloggs 1996”)
```

Replace “Joe Bloggs” with your own name (there is a limit on the length of the string though). This is fixed by the indirected size and the physical size of the box. These can easily be changed using !TemplEd.

Now for the Version field. It is common practice to display the version number and date in the form “X.XX (Dt-Mth-Yr)”, eg:

1.42 (16-Oct-99)

Now add the following line just above PROCwimp_poll (changing the date to today):

```
PROCwimp_puticontext(info%,4,“1.00 (29-Mar-98)”)
```

The parameters to the function are quite simply. From left to right they are: the window handle, the icon number, and the text to put into the icon (in the window).

- Note that it is important to get the string length correct or you will lose the slabbed effect in the info box, and if you are using RISC OS 3.5+ the desktop font may revert back to the system font.

Check the application works as it is supposed to and then try using the same method to change the “Purpose” field to something more interesting!

4. Window & icon control

Most applications have a window that appears when the user clicks on the iconbar icon. This is very easy to do:

Copy “Template2” from the tutorial folder into the !MyApp application directory, and rename as “Templates” thus overwriting the previous one. Add the following line below FNwimp_loadwindow to load in a second window:

```
main%=FNwimp_loadwindow(“<MyApp$Dir>.Templates”,“main”,0)
```

Whenever a Select or Adjust mouse button is pressed over one of !MyApp’s windows or icons, the user function PROCuser_mouseclick is called, passing to it the window handle, the icon number, a number relating to the mouse button pressed and the work area coordinates the point was at. These are in window%, icon%, button%, workx% and worky% respectively.

If you change the window's work area to a button type of “Click” in a template editor, then the work area has an icon number of -1. This means that mouse clicks on the work area where there are no icons will also result in PROCuser_mouseclick being called.

Change PROCuser_mouseclick so it looks like:

```
    DEF PROCuser_mouseclick(window%,icon%,button%,workx%,worky%)
    CASE window% OF
    WHEN bar% : PROCwimp_openwindow(main%,1,-1)
    ENDCASE
    ENDPROC
```

As you can see, when a mouse button is pressed over the iconbar icon, then PROCwimp_openwindow is called.
The first parameter is the handle of the window to open. The second means open the window in the centre of the screen, and the third parameter means open it on top of all other windows.

If the second parameter was 0 then the window would open where you last left it (ie. before it was closed), or if it was being opened for the first time then it would open as it was positioned in the templates file.

If the second parameter was 2 then the window would open centred on the mouse pointer. Useful if you want to open save windows below the pointer so the user doesn't have to move the pointer very much to start dragging the file icon.

If the third parameter was -2 then the window would open behind all the others. If it was a window handle then it would open behind that window.

Run !MyApp and check that it works, then try changing the parameters to PROCwimp_openwindow and see what happens.

The window is divided up into three sections. I will use each section to demonstrate one or more aspects of icon control using DrWimp. The icon number for each icon can be found by loading the templates file into !TemplEd as described before.

The first section contains a writable icon which can take up to 19 characters. This amount is set by using !TemplEd and changing the indirected icon size for the writable icon.

Entering text into writable icons is fully automated by the wimp. Click in the icon and the caret will appear so you can enter some text.

What we want to do is enter some text and when Return is pressed, or ‘OK’ is clicked on (icon number =2), the text is read from the icon and copied into the text icon below.

The wimp function FNwimp_geticontext reads text from icons. ie. it is the dual of PROCwimp_puticontext. The parameters passed are the window handle and the icon number. The function returns the text in a string. All that we need to do then is use PROCwimp_puticontext to put the text into the other icon.

Alter PROCuser_mouseclick so it looks like:

```
DEF PROCuser_mouseclick(window%,icon%,button%,workx%,worky%)
CASE window% OF
WHEN bar% : PROCwimp_openwindow(main%,1,-1)
WHEN main% :
  CASE icon% OF
  WHEN 2 :
  text$=FNwimp_geticontext(main%,1)
  PROCwimp_puticontext(main%,11,text$)
  ENDCASE
ENDCASE
ENDPROC
```

Re-load !MyApp and check that it works. You should be able to see from the code above that the writable icon is number 1, the text icon is number 11, and the OK icon is number 2.

Note that you can only put text into icons if they are what is called “indirected”. You can make an icon indirected by turning the option on when editing the icon in a template editor. The indirected number is the maximum number of characters you can put plus one (the terminator).

So making an icon indirected with a value of 11 is enough for putting a filename in that icon. Reduce the indirected value to as low as possible to save on memory. If you have an icon like a label that you are never going to change the text of then you can make it non-indirected to save memory.

If you use a lot of indirected icons then you may run out a allocated memory. If this happens then increase the value which is the the third parameter to FNwimp_initialise until you have enough space and the errors stop.

Now we have to get the Return key to perform the same action when it is pressed and the caret is in the writable icon. Pressing the return key in this situation should always act the same as clicking on the icon with the yellow trough/border around it (called the Default Action icon).

FNuser_keypress is called whenever a key is pressed. Passed to it is the window handle and the icon number to locate the caret. Also passed to it is the ASCII number of the key, which for Return is 13. If we use the keypress to do something, then we must return a 1. This is to stop the keypress being passed onto other tasks.

Alter FNuser_keypress so it is like:

```
DEF FNuser_keypress(window%,icon%,key)
used=0
CASE window% OF
WHEN main% :
  CASE icon% OF
  WHEN 1 :
    IF key=13 THEN
      text$=FNwimp_geticontext(main%,1)
      PROCwimp_puticontext(main%,11,text$)
      used=1
    ENDIF
  ENDCASE
ENDCASE
=used
```

There is a lot of code there for what it actually does, but it has be written so it is easy to add more pieces of code for lots of windows and icons, with the minimum amount of effort. Also it allows the code to be executed to be put onto several lines, making easier to read. Note that the writable icon is number 1, where the caret is when return is pressed.

There is just one thing missing now. When you click on the ‘OK’ icon it presses in briefly. When you press Return it doesn’t. It is much more reassuring for the user to see it press in, because then they know exactly what has happened, and that they haven’t just wiped half their document or any other unsaved data away.

DrWimp provides a wimp function to invert or select an icon, and un-select it again, so all we have to do is select the ‘OK’ icon, copy the text, then un-select the ‘OK’ icon.

Alter the part between “`IF key=13 THEN`” and “`ENDIF`” so it looks like:

```
IF key=13 THEN
  PROCwimp_iconselect(main%,2,1)
  text$=FNwimp_geticontext(main%,1)
  PROCwimp_puticontext(main%,11,text$)
  used=1
  PROCwimp_iconselect(main%,2,0)
ENDIF
```

Re-run !MyApp and check it works.

PROCwimp_iconselect has three parameters (window%,icon%,state%). The first is the window handle and the second is the icon number. In this case it is 2 for the ‘OK’ icon. The last parameter controls the inversion/selection. If it is a 1 then the icon is selected, a 0 makes it unselected. If you removed the second PROCwimp_iconselect then the ‘OK’ button would stay pressed in. Try it and see!

On fast machines the short press in may be too quick and just produce a flicker. You may like to add the following line just above PROCwimp_iconselect(main%,2,0):

```
PROCwimp_singlepoll
```

The use of that function is explained in a later section on multitasking operations but for now you can treat it in this case (and really only in this case) as a short delay.

The second section contains two radio buttons and an button labelled ‘Swap’. Both the radio buttons have the same ESG group (out of an ESG group, only one radio button can be selected at once (see !TemplEd documents inside the application and the example application)), so clicking on one de-selects the other. i.e. when the user clicks on ‘Swap’ the selected and de-selected radio icons swap over, as if you had clicked on the de-selected one. The button has an icon number of 6.

If you write an application with radio buttons, you have to keep track of which ones are selected. For !MyApp we will use choice% which will be either 1 or 2 depending on which one is selected. When you first load !MyApp we will have it so Choice 1 is selected, so we will set choice% to 1 at the start. Although you can select the radio icon when editing the templates it is always best to do this in your program so there is no chance of it falling over if a user fiddles with the templates file. When ‘Swap’ is clicked on, we look at choice% to decide which to select and which to de-select.

Just before PROCwimp_poll, add the following lines:

```
choice%=1
PROCwimp_iconselect(main%,4,1)
PROCwimp_iconselect(main%,5,0)
```

After PROCwimp_puticontext(main%,11,text$) in PROCuser_mouseclick, before the ENDCASE, add the following:

```
WHEN 6 :
IF choice%=1 THEN
  PROCwimp_iconselect(main%,4,0)
  PROCwimp_iconselect(main%,5,1)
ENDIF
IF choice%=2 THEN
  PROCwimp_iconselect(main%,4,1)
  PROCwimp_iconselect(main%,5,0)
ENDIF
choice%+=1
IF choice%>2 choice%=1
WHEN 4 :
choice%=1
WHEN 5 :
choice%=2
```

It is mostly self explanatory. "`choice%+=1:IF choice%>2 choice%=1`" toggles choice% between 1 and 2 to keep track of which one is selected.

There isn't much point in doing this as it is much easier for the user to simply click on the radio icons instead, but it is a good demonstration in selecting and de-selecting radio icons.

There is a small problem with radio buttons. Try pressing Adjust over a selected one. You will find that it becomes un-selected. Having none of the radio buttons selected may be an undesirable situation. It is quite simple to solve this.

If you have a look at PROCuser_mouseclick you will see that the third parameter passed to is is called button%. Basically, if button%=4 then Select was pressed. If button%=1 then Adjust was pressed. There are also numbers for combinations of mouse buttons, but we need not concern ourselves with that at the moment.

Alter the WHEN 4 and WHEN 5 parts so they look like:

```
WHEN 4 :
IF button%=1 PROCwimp_iconselect(main%,4,1)
choice%=1
WHEN 5 :
IF button%=1 PROCwimp_iconselect(main%,5,1)
choice%=2
```

Re-load !MyApp and make sure that you always have to have a radio icon selected.

- Note: the act of clicking on a radio icon with any mouse button will automatically deselect the others in the same ESG group, so that's why we only set the icon clicked on with Adjust and not deselect the other here.

The last section has an option icon and two buttons. What we want to do with this section is control the option icon using the buttons, in much the same way as for the radio icons. There is a difference though, in that at any one time, one of the buttons will have to be disabled; there is not much point clicking on 'On' if it is already turned on.

Just before PROCwimp_poll, add the following:

```
option%=0
PROCwimp_icondisable(main%,10)
```

option% records the state of the option icon, so if the application ever needed to act upon its state, it could just look at option%. The function disables (greys out) the 'Off' button (icon number 10). Its duel PROCwimp_iconenable enables icons and has the

same parameters.
In PROCuser_mouseclick, modify it so it is like:

```
WHEN 5 :
IF button%=1 PROCwimp_iconselect(main%,5,1)
choice%=2
WHEN 9 :
PROCwimp_icondisable(main%,9)
PROCwimp_iconenable(main%,10)
PROCwimp_iconselect(main%,8,1)
option%=1
WHEN 10 :
PROCwimp_iconenable(main%,9)
PROCwimp_icondisable(main%,10)
PROCwimp_iconselect(main%,8,0)
option%=0
WHEN 8 :
IF option%=0 THEN
  PROCwimp_iconenable(main%,10)
  PROCwimp_icondisable(main%,9)
ENDIF
IF option%=1 THEN
  PROCwimp_icondisable(main%,10)
  PROCwimp_iconenable(main%,9)
ENDIF
option%=ABS(1-option%)
```

You should be able to see how it works, and know exactly what will happen when you click on the icons. Run it and check!

Using PROCwimp_icondisable and PROCwimp_iconenable you can grey out and un-grey out any icon you wish. If the user clicks on a disabled icon, then the mouse click is ignored, and PROCuser_mouseclick is not called.

As you can see, PROCuser_mouseclick is probably one of the most important functions and will contain most of the code to control your WIMP front end.

You must have noticed that as soon as you have some un-saved data in a program (eg. !Edit, !Draw et al), the title of the window has an asterisk added to end. Using DrWimp this is quite easy to achieve, once you know that you have some unsaved data...

For !MyApp we will assume that we have some unsaved data when ‘OK’ or Return is pressed. FNwimp_getwindowtitle returns a string containing the name.

PROCwimp_putwindowtitle changes the title to the supplied string. All we have to do therefore is read the title, add “ *” to the end and put it back.

Change the WHEN 2 : section in PROCuser_mouseclick to:

```
WHEN 2 :
text$=FNwimp_geticontext(main%,1)
PROCwimp_puticontext(main%,11,text$)
r=1
title$=FNwimp_getwindowtitle(main%)
IF RIGHT$(title$,2)=“ *” r=0
IF r=1 PROCwimp_putwindowtitle(main%,title$+“ *”)
```

The title is put into title$, then the end is checked to see if “ *” has already been added (we don’t want to add “ *” each time ‘OK’ is pressed!). If it hasn’t then the asterisk is added. You should now be able to alter a part of FNuser_keypress so the title changes when Return is pressed as well.

If your program has a save feature (details on how to do that coming up later) then when the data has been saved you can remove the asterisk from the title.

- Note: In order to change the window title it must be indirected. This is set up using a template editor like !TemplEd. The indirected size only needs to be 1.

Just a quick thing for you to try while I am dealing with windows: Quite a few programs have a window that appears in the centre of the screen when it loads. It stays there for a bit before closing again. Using banners is a good way of announcing your program or reminding people to register it. DrWimp has a function that handles banners for you. While the banner is on the screen, you can still use the desktop and your application.

This is how you use it: Just before PROCwimp_poll enter the following line:

```
PROCwimp_banner(info%,3)
```

When you load !MyApp, the info window will appear for three seconds in the middle of the screen. The info window isn’t really suitable, so I’ll leave it to you to design a window, load it in, and put things like version number and date, etc in using PROCwimp_puticontext.

5. Advanced menus

So far we have only got a simple two item menu with an info window (windows off menus are called dialogue boxes). There is a lot more you can do with menus. This part looks at submenus, ticks, greying out, dotted lines, changing item's text, and writable menu items.

First of all, lets create a menu for our main window. Just below the FNwimp_createmenu line that we have already, add the following lines:

```
menu$="MyApp/Info/Item 2/Item 3/Item 4/Save"
mainmenu%=FNwimp_createmenu(menu$,0)
PROCwimp_attachsubmenu(mainmenu%,1,info%)
```

And attach the menu to the main window by altering FNuser_menu so it looks like:

```
DEF FNuser_menu(window%,icon%)
CASE window% OF
WHEN bar% : =barmenu%
WHEN main% : =mainmenu%
ENDCASE
=0
```

When you press Menu anywhere over the main window, you should get a menu with 5 items. The first should have a submenu leading to the info window.

Now create another menu by adding the following line above the menu$=... you have just entered:

```
i3menu%=FNwimp_createmenu("Item 3/Help/Tick me!",0)
```

Now you can see from the title of this menu that it is obviously a submenu for Item 3 on our main menu. Here is how to add it: instead of entering a window handle to the submenu attaching function, you simply enter a menu handle. For example below PROCwimp_attachsubmenu(mainmenu%,1,info%), add:

```
PROCwimp_attachsubmenu(mainmenu%,3,i3menu%)
```

Run !MyApp and take a look at the menu. And that is all there is to it! Here is another example of how easy it is to use:

Before the i3menu%=... line add the following:

```
tickmenu%=FNwimp_createmenu("Tick me!/Bored/Dull/Waffle",0)
```

And after the i3menu%=... line add:

```
PROCwimp_attachsubmenu(i3menu%,2,tickmenu%)
```

Hence, you can see that it is quite painless to build up very complex menu structures.

As you have seen before, if you choose a menu item then PROCuser_menuselection is called. If you choose the first item from the menu tickmenu%, then menu%=tickmenu% and item%=1.

When you create a menu it has no ticks by the side of any items, no dotted lines separating items, and none of the items are greyed out. Any that need to be done when the application loads, have to be set up after the menu has been created.

PROCwimp_menutick toggles a tick next to the menu item specified. Add the following line to the end of PROCuser_menuselection and try it out:

```
     IF menu%=tickmenu% AND item%=1 PROCwimp_menutick(menu%,item%)
```

PROCwimp_menudisable and PROCwimp_menuenable surprisingly enough enable and disable menu items!

At the end of PROCuser_menuselection, add the following lines and try it out:

```
     IF menu%=mainmenu% AND item%=2 PROCwimp_menudisable(menu%,3)
     IF menu%=mainmenu% AND item%=4 PROCwimp_menuenable(menu%,3)
```

PROCwimp_menudottedline(menu%,item%) puts a dotted line on the menu below the item specified. Try it out by adding the following after i3menu% is created:

```
     PROCwimp_menudottedline(i3menu%,1)
```

There are a few more functions related to menus:

```
     PROCwimp_menupopup(menu%,bar%,x%,y%)
```

This brings up the menu menu% at the co-ordinates x%,y% if bar%=0. If bar%=1 then the menu is positioned for the iconbar icon. This can be useful for the menu icons (the icons that depict a menu and are usually to the right of writable icons, offering the user a choice of strings to enter into the writable icon). You detect a click with Select or Adjust on the icon, read the mouse co-ordinates with MOUSE X,Y,B and bring up the menu. You will need to add some offsets to x% and y% so then menu appears to the side.

The text of a menu item can be read by using FNwimp_getmenutext, passing the menu handle and the item number. The dual of this is PROCwimp_putmenutext. This allows you to change the text of an item. The text can be up to 78 characters wide but this would cross the screen, so if you don’t know the length of the string then truncate it possibly by using some sort of code like LEFT$(....,78).

Delete the PROCwimp_menudisable and PROCwimp_menuenable lines, and put the following lines in the same place:

```
    IF menu%=mainmenu% AND item%=2 THEN
      PROCwimp_putmenutext(mainmenu%,3,“ABCDEFGHIJKLMNOPQRS”)
    ENDIF
```

Try choosing the second menu item with Adjust. The menu will automatically adjust its width to accommodate the longest line.

The last major menu function turns an item into a writable one. A block of memory is reserved to put the text into automatically. After the main menu has been defined, add the following line:

```
    PROCwimp_menuwrite(mainmenu%,4,20)
```

The first parameter is the menu handle, the second is the item number. It doesn’t matter if some text is already there. The third parameter is the maximum length allowed to be entered. To read it simply use FNwimp_getmenutext when the item is selected.

The final handful of menu functions don’t really need much description. For more details see !Fnc’n’Prc or section 3 in this manual.

FNwimp_menusize(menu%)	- Returns the number of items in a menu.
PROCwimp_menuclose	- Closes any open menu.
FNwimp_getmenutitle(menu%)	- Returns the title of the menu.
PROCwimp_putmenutitle(menu%,title$)	- Changes the menu title to title$.

6. Panes

Using DrWimp, it is very easy to attach a pane to a window. From the tutorials folder, copy “Template3” into the !MyApp directory and rename as “Templates”.

Where the other windows are loaded in, load in the window “pane”:

```
    pane%=FNwimp_loadwindow(“<MyApp$Dir>.Templates”,“pane”,0)
```

The pane wants to be attached to the main window.
When the main window is dragged about, it is actually being continually re-opened all the time. This means that PROCuser_openwindow is also being called continually just before the window in question is opened.

Every time the main window is opened we want to first open the pane in the window stack position that is supplied. Then, by telling DrWimp that the window has a pane using FNuser_pane, the main window is opened below the pane.

In PROCuser_openwindow, add the following lines:

```
IF window%=main% THEN
  xoff%=x%-FNwimp_getwindowsize(pane%,0)
  PROCwimp_openwindowat(pane%,xoff%,y%,stack%)
ENDIF
```

The pane also needs to be closed when the main window is, so in PROCuser_closewindow add the following line:

```
IF window%=main% PROCwimp_closewindow(pane%)
```

There is one last thing to do; add the following line to FNuser_pane to tell DrWimp that the window has a pane:

```
IF window%=main% =pane%
```

Re-load !MyApp and check that it is working. And that is all there is to it! Just a quick explanation of a few things:

PROCwimp_openwindowat opens the specified window so the top left corner is at the co-ordinates x%,y%. If you want the window to open at the top, then stack%=-1, or at the bottom, then stack%=-2. If you want the window to open behind a certain one, then stack%=the handle of the window to open behind.

If you want a window to open with another one but not follow it around, then in PROCuser_openwindow, use PROCwimp_openwindow. If you set the second parameter to 1 then it will continually re-centre, so it is best to set it to 0. Also, just pass stack% straight through.

Multiple panes are easily achieved. Add the following line to load in a second pane:

```
pane2%=FNwimp_loadwindow("<MyApp$Dir>.Templates","pane2",0)
```

Now, we have to consider more carefully the order of the window stack.

When you have one pane you look to see where the wimp is going to open the window in the stack (stored in stack%), then first open the pane in that stack position. DrWimp then sees that the window has a pane so opens the window behind the pane. This is what we have done so far.

With two panes you have to decide which one is going to be top-most, with the second pane behind that, followed by DrWimp opening the window behind the second pane (this can be extended to as many panes as you like).

We are going to make DrWimp open the second pane behind the current pane, so change the section in PROCuser_openwindow to:

```
IF window%=main% THEN
  xoff%=x%-FNwimp_getwindowsize(pane%,0)
  PROCwimp_openwindowat(pane%,xoff%,y%,stack%)
  yoff%=y%-FNwimp_getwindowsize(main%,1)
  PROCwimp_openwindowat(pane2%,x%,yoff%,pane%)
ENDIF
```

So we have the first pane opening in the stack position, then the second pane is opened below the first pane, but in order to get DrWimp to open the window below the second pane we have to change FNuser_pane to return the handle of the bottom-most pane (the second one):

```
IF window%=main% =pane2%
```

Change the relevant parts in PROCuser_closewindow so when the main window is closed both the panes close and check it all works:

```
IF window%=main% THEN
  PROCwimp_closewindow(pane%)
  PROCwimp_closewindow(pane2%)
ENDIF
```

- Note that you can attach panes to any window, so you could have a window with three panes and another with one, etc, but be careful not to attach panes to panes!

7. Save windows

Adding and controlling save windows is very easy. Copy the file “Template4” from the Tutorials folder into !MyApp and rename as “Templates”. Load in the save window:

```
save%=FNwimp_loadwindow(“<MyApp$Dir>.Templates”,“save”,0)
```

and after PROCwimp_attachsubmenu(mainmenu%,3,i3menu%), add:

```
PROCwimp_attachsubmenu(mainmenu%,5,save%)
```

If you run !MyApp now, the save window will behave like any other window that hasn’t got any code to tell it what to do. Everything starts to work when you add the following line to FNuser_savefiletype:

```
IF window%=save% =“FFF”
```

DrWimp now knows that save% is a save window and that it saves text files. Don't try it just yet as DrWimp will try to filetype the saved file and it is not created, so will throw up an error.

When the icon is dragged to a destination, FNuser_savedata is called. This is where you do the actual saving. path$ is the full pathname of the file that you are saving to and window% is the handle of the window that the icon was dragged from.

Don’t always assume that the pathname in path$ will be something like “IDEFS::Gromit.$.TextFile”, It could be a system variable that the wimp expands to a full pathname later on.

So, enter the following into FNuser_savedata:

```
IF window%=save% THEN
  file%=OPENOUT(path$)
  BPUT#file%,“This is a text file,”
  BPUT#file%,“created by MyApp.”
  CLOSE#file%
  PROCwimp_menuclose
ENDIF
```

• Note: the filetyping is taken care of by DrWimp.

You can now drag the icon to the filer or other applications. The error messages “You must drag the icon to a filer window to save...” are taken care of by DrWimp.

It is very easy to add lots more save windows. Simply load them in, add them to a menu (or provide some way the user can get to them), return their filetype from FNuser_savefiletype, and do the saving in FNuser_savedata!

FNuser_savedata returns a number. This can be a 1 or a 0. Only return a 1 if some data was saved, otherwise return a 0. This allows you to do some checking in FNuser_savedata then decide whether to save or not depending on the results.

For example at the start of FNuser_savedata, you could check to see whether the file path$ exists. If it does then use FNwimp_errorchoice to allow the user to decide whether to overwrite the file or not. If they decide not to then return a 0, otherwise save the file and return a 1.

A rather neat trick that can be performed is to open windows as menus. You can see the effect if you add the following to the bottom of PROCuser_menuselection:

```
    IF menu%=mainmenu% AND item%=5 THEN
      MOUSE X,Y,B
      PROCwimp_menupopup(save%,0,X,Y)
    ENDIF
```

Now select the save item from the save menu. You will see that the save window opens and stays open until you complete the save or click elsewhere. You can tidy this rough example up so that the window is centred on the pointer, etc.

You may have noticed that some save windows have a rather attractive looking border around the file icon. Impression is an example of this, but there is a small complication as you will see. Change the line where the save window is loaded in to:

```
    save%=FNwimp_loadwindow(“<MyApp$Dir>.Templates”,“nicesave”,0)
```

so a different template is used for the save window. Now run !MyApp and try using the save window now. You will probably get an error and the application will quit.

Loading the templates into TemplEd you will see that the icon to be dragged is now icon number 3, where it was 0 in the old save window. PROCuser_saveicon is used to tell DrWimp which icon to drag for each save window. The default value is 0, so if you are using a modified save window like this then you must set a new value. So add the following line to PROCuser_saveicon:

```
    IF window%=save% drag%=3
```

Reload the application and check that it works.

PROCuser_saveicon is strange in that it works rather like a function (FN) in that it can return values. But whereas a function (FN) can only return one, PROCuser_saveicon can return three by using the RETURN keyword. drag% is the first one we have come across, and if you don't want to use the default value of 0 for the icon to drag, you can set it to something else. There is also write% and ok% which you can set for a save window as well. They are the icon numbers of the writeable icon where the filename or pathname goes, and the OK button to click on to save to the pathname.

The default for write% is 1 and for ok% its 2. You can simply set these to whatever you have used in your window template to override the defaults. For example:

```
    IF window%=save% drag%=1:write%=0:ok%=8
```

or:

```
    IF window%=mainsave% write%=6:ok%=5
```

8. Errors

If you get an error or you are trying to debug a program, error windows can be very useful. They can tell you information while the program is running.

Error windows can be altered quite a bit, so there are several parameters needed to bring one up. There are two wimp functions for error windows:

```
PROCwimp_error(title$,error$,button%,prefix%)
```

This one is the most common.

title$	is the title of the window. This is usually the application name.
error$	is the error message itself. The text is wordwrapped automatically.
button%	controls the default button. If it is 1 then you get an ‘OK’ button. If it is a 2 then you get a ‘CANCEL’ button.
prefix%	allows you to tailor the title to suit the error message. If prefix% is 0 then the title is title$. If it is 1 then the title is prefixed with “Error from ”. And if it is 2 then the title is prefixed by “Message from ”.

This is the other function.

```
FNwimp_errorchoice(title$,error$,prefix%)
```

The parameters act in exactly the same way as for PROCwimp_error. This function displays an error window with an ‘OK’ button and a ‘CANCEL’ button. If ‘OK’ is clicked on then the function returns a TRUE (-1). If ‘CANCEL’ is clicked on then it returns a FALSE (0).

9. Message files

It is getting increasingly common for applications to have Message files. These files have most of the text for the application in, so it can be easily translated by someone who doesn’t need to know anything about programming.

Copy the “Messages” file from the tutorial folder into the !MyApp directory. Load it into !Edit and have a look at it.

Comments start with a hash “#”. Lines which have text on to be used in the application start with what is called a token. This is just a few letters that the line can be identified by. The token and the text are separated by a colon. For example:

```
LIB:Doctor Wimp
```

If we wanted to use the line of text “Doctor Wimp” then we would reference it by using the token “LIB”.

- Note: The message file must be terminated by a Return, otherwise the last message in the file will not work.

Lines of text can also have strings inserted into them when they are read into the application. For example:

```
VER:1.00 (%0-%1-99)
```

When you read in this line, you supply two strings. These could be “29“ and “Mar” for example. When the line is read in it ends up as “1.00 (29-Mar-99)”.

PROCwimp_initmessages(path$) sets up blocks of memory ready to read in the lines of text. path$ is the full pathname to the Messages file.

Somewhere before the PROCwimp_poll, call the function for the Messages file inside the !MyApp directory.

To read a line without substituting any strings you can use FNwimp_messlook0(token$). This returns the line of text.

To read a line and replace “%0” with a string you can use FNwimp_messlook1(token$,a$), where “%0” in the messages file is replaced with a$.

To read and substitute two strings you can use FNwimp_messlook2(token$,a$,b$). Add the following lines after PROCwimp_initmessages and run your application. When you have checked that it works you can delete the lines:

```
lib$=FNwimp_messlook0(“LIB”)
PROCwimp_error(appname$,“LIB=”+lib$,1,2)
os$=FNwimp_messlook1(“OS”,“3.11”)
PROCwimp_error(appname$,“OS=”+os$,1,2)
ver$=FNwimp_messlook2(“VER”,“29”,“Mar”)
PROCwimp_error(appname$,“VER=”+ver$,1,2)
```

It is possible to automatically create menus from a message file. For example, say you had the following in a messages file:

```
BMenuT:Icnbar menu
BMenu1:Info
BMenu2:Quit
```

Then using the following, a basic iconbar menu could be created with the title “Icnbar menu”, and two items, the first being “Info” and the second being “Quit”:

```
barmenu%=FNwimp_createmessagesmenu("BMenu","",0)
```

The first parameter is the tag, and is used to find the bits of the menu in the messages file. tag+“T” is the title, tag+“1” is item one, tag+“2” is item 2 and so on.
The second parameter is the menu title. In this case its an empty string so the one in the messages file is used. If we did:

```
barmenu%=FNwimp_createmessagemenu("Bmenu",appname$,0)
```

instead, then the menu title would be the string appname$, overriding the title in the messages file. In fact if you specify a menu title, then the tag in the messages file for the title doesn't have to be included, and you can just specify the items.
The final paramter is the maximum size of the menu, and allows the menu to be dynamic. It is exactly the same as the last parameter of FNwimp_createmenu.

10. Loading data

This is very simple to do. Whenever a file (or directory or application) is dropped onto a window belonging to your application, then FNuser_loaddata is called. This is where you actually load it in to a block of memory or an array.

path$	is the full pathname of the file to be loaded. Don’t always assume that it is something like : “IDEFS::Andy.$.TextFile”.
window%	is the window handle.
icon%	the icon number that the file was dropped on to. Most of the time you would only need to check if it was a window or the iconbar icon, but it can be used for drop-boxes. These are boxes with text in saying something like: “Drop file to load here”.
ftype$	is the filetype of the file to load. Eg: for text files it is “FFF”. Files have a three character hexadecimal filetype, but directories are “1000”, and applications are “2000”.
workx%,worky%	is the window window% work area coordinates that the file was dropped at.

If you decide to load in the file after looking at all the parameters, then you must load it in and return a 1. This is so DrWimp can send a message to the filer or any other application that the file came from, saying that it has been loaded.

Here is an example piece of code for loading in a text file in FNuser_loaddata

```
DEF FNuser_loaddata(path$,window%,icon%,ftype$,workx%,worky%)
used=0
IF ftype$="FFF" THEN
file%=OPENIN(path$)
L=1
REPEAT
A$(L)=GET$#file%
L+=1
UNTIL EOF#file%
CLOSE#file%
lines%=L-1
used=1
ENDIF
=used
```

What it does is load in a text file into an array called A$ (which you will need to create before PROCwimp_poll). At the end, lines%=the number of lines read in.

Try altering !MyApp so it can load in a text file, then using the save box, allow the user to save the text file to somewhere else. All you have to do is alter the save code so it saves the array.

Files can also be loaded in with double-clicks on the file. They can be made to load in your application first, if it isn't already. The only thing you need to add to achieve this is a line like the following in your !Run file:

```
Set Alias$@RunType_000 Run <Obey$Dir> %%*0
```

Change “000” for the filetype you are using, eg. “FFF” for text files.
Beware though, when a file is loaded in this way, the parameters window% and icon% passed to FNwimp_loaddata are both set to 0.

You may wish to give your filetype a name. Eg. to give the filetype &000 the name “TestFile” put a line like the following in both your !Run file and !Boot file:

```
Set File$Type_000 TestFile
```

11. Interactive help

DrWimp supports interactive help applications like Acorn’s !Help. All you have to do is return the help string for a given window handle and icon number. FNuser_help is where you do this.

Enter the following lines into FNuser_help, reload !MyApp, load !Help, and move the pointer over the info window and the iconbar icon:

```
DEF FNuser_help(window%,icon%)
h$=“”
CASE window% OF
  WHEN info% :
  CASE icon% OF
    WHEN 1 : h$=“This application is called ‘MyApp’ ”
    WHEN 2 : h$=“It tests the DrWimp library”
    WHEN 3 : h$=“MyApp was written by Joe Bloggs”
    WHEN 4 : h$=“This is the version number and date”
    OTHERWISE : h$=“This is the MyApp info window.”
  ENDCASE
  WHEN bar% : h$=“This is the MyApp icon.”
ENDCASE
=h$
```

FNuser_help is only for windows and icons, if you want to return interactive help for menu items, then use FNuser_menuhelp, eg:

```
DEF FNuser_menuhelp(menu%,item%)
h$=“”
CASE menu% OF
  WHEN barmenu% :
    CASE item% OF
      WHEN 2 : h$=“Click Select to quit"
    ENDCASE
ENDCASE
=h$
```

12. Sprite areas & Mouse pointers

Whenever the pointer moves in and out of one of !MyApp’s windows, the functions PROCuser_enteringwindow and PROCuser_leavingwindow are called. This makes it very easy to change the mouse pointer when it is over one of your windows. There is a function to change the pointer for you, but first I will look at sprite areas as it is closely related:

All the sprites in the windows so far are from the wimp sprite pool. An application can however, create it’s own private sprite areas that are stored in the wimpslot, or memory used by the application. This has the advantage that when the application quits, all the memory is regained. FNwimp_loadsprites does all the work for you and must be called before loading in any windows that use user sprites. Passed to it is a pathname to a sprite file and a handle and the file is loaded in. The sprites in that area can then be used in windows (see PROCwimp_loadwindow for parameters, but briefly you set the last parameter to the handle for the sprite area), or for pointers.

PROCwimp_pointer controls the mouse pointer. Pointer number two is always used for the second user definable pointer.

Copy the file “Sprites” from the tutorials folder into the !MyApp directory. Enter the following lines after FNwimp_initialise:

```
size%=FNwimp_measurefile("<MyApp$Dir>.Sprites")
DIM sprites% size%
a%=FNwimp_loadsprites(“<MyApp$Dir>.Sprites”,sprites%)
```

a% is an arbitary variable not being used. This is explained later.
In PROCuser_enteringwindow, add the following lines:

```
IF window%=main% THEN
  PROCwimp_pointer(1,sprites%,“ptr_hand”)
ENDIF
```

And in PROCuser_leavingwindow, add the following:

```
IF window%=main% THEN
  PROCwimp_pointer(0,0,“”)
ENDIF
```

Run !MyApp and you should find that when you move the pointer over the main window, it turns into a hand. The first parameter tells DrWimp whether to use the default pointer, or a user defined one. The second parameter should be 0 for the wimp sprite pool, and a handle for a user sprite area. The default pointer can be found in the RMA, so it should be a 0 to use it. The last parameter is the name of the sprite to use for the user defined pointer. If you are using the default pointer, then you don’t need to put anything into the string.

It is a good idea to change the pointer into one looking like a caret when it is over a writable icon. In the validation field of the icon enter:

```
R7;Pptr_write
```

•Note: the sprite used for the pointer is usually defined in mode 8 (lo-res) or 19 (hi-res), and should be in the Wimp sprite area.

You can do similar things with other icons like the menu ones. Impression is a good example. See the section on validation strings for more details.

The method for loading in a sprite file may seem a bit strange, especially the idea of passing the handle of the area to the function, instead of it returning it to you. This is the same arrangement as loading in drawfiles, and allows one block of memory to hold lots of sprite files, all with their own handles.

Things might become a little clearer if you look at the following example code to load in two sprite files:

```
size%=FNwimp_measurefile("<MyApp$Dir>.Sprites1")
size%+=FNwimp_measurefile("<MyApp$Dir>.Sprites2")
DIM sprites% size%
sprites1%=sprites%
sprites2%=FNwimp_loadsprites(“<MyApp$Dir>.Sprites1”,sprites1%)
a%=FNwimp_loadsprites("<MyApp$Dir>.Sprites2",sprites2%)
```

This first measures both the sprite files whose paths are shown, adding the size of the second to the first. Then DIM is used to create a block of memory big enough to hold both the sprite files. The handle of the first sprite file is assigned to the name of the block. The name of the block can now be forgotten.

The final two lines load in the sprite files. The last parameter is the handle of the sprite file being loaded in, and returned is the handle to the next sprite file to be loaded into that block.

As after loading in both files the area of memory is full, the handle returned by the last call of FNwimp_loadsprites cannot be used for anything, so is just put into an arbitary variable and forgotten about.

13. Redraws, leafnames & versions

If you have some user graphics in a window, that can’t be redrawn by the wimp, then it will ask you to do the redraw. For example, you could be using the CIRCLE command to draw a circle in a window.

When a redraw is required, PROCuser_redraw is called. The handle of the window is passed, and the position of the rectangle. The rectangle is like a graphics window, so you can redraw all the window contents if you like (although that is a bit slow). Anything outside is clipped.

Sometimes it can be useful to get a leafname from a pathname.

pathname = “IDEFS::Andy.$.Progs.Project1.!Wow.Sprites”
leafname = “Sprites”

DrWimp can do this for you with FNwimp_getleafname(path$). It returns the leafname.

It is very likely that the DrWimp library will have some modifications or improvements made to it at some point. I very much hope that they will not affect the way any existing wimp or user functions are called, but just in case, you can call a function to read the version number of the library. So if someone decides to replace the library in your application with a newer version, then your program can read the version number and refuse to work with it.

FNwimp_libversion returns the version number x 100. So if it is version 1.43 then it will return 143.

14. Changing sprites

If you have an icon which is a sprite (like the text file icon in the save window) then you can change it to another sprite by using PROCwimp_puticontext. This will only work however, if the icon is indirected. That is set up using a template editor.

For example: insert the following line just before PROCwimp_poll:

```
PROCwimp_puticontext(save%,0,“file_ffd”)
```

Re-load !MyApp and the file icon in the save window will now be a Data one.

15. Large menus & rebuilds

Yes, this section is again concerned with menus. In particular: how to create very large menus, how to completely change a menu (ie. a re-build), and add and remove items.

Menus can be created so that they are dynamic. In other words, then can grow and shrink in accordance with what your application wants.

You should recall from the introduction section that when a menu is created a block of memory of a fixed size is reserved to put the data that the wimp needs in. So for example:

```
menu%=FNwimp_createmenu(“MyApp/Info/Quit”,0)
```

will reserve a block of memory just big enough to hold the menu with only the two items specified.

But what happens if you want to add another item to the menu. This will create three items, so all the data for one item will be pushed into the next part of the memory. This could be holding the contents of variables that you are using, thus corrupting them, or even more likely you will crash the application, because you are trying to write to some memory addresses that don’t actually exist (address exception errors are the result of this).

What is needed is a way of making sure the block of memory is big enough. This is where the last parameter to FNwimp_createmenu comes in:
If it is less than or equal to the number of items specified in the string, then the block of memory will be just big enough to hold the items given. If it is bigger, then it is the maximum number of items that can be on that menu.

So if you used:

```
    menu%=FNwimp_createmenu(“MyApp/Info/Quit”,20)
```

then you would create a menu the same as before, but you can have up to 20 items added to it later on.

PROCwimp_putmenuitem and PROCwimp_removemenuitem add and remove items from menus. Note: no check is made to ensure that the menu is not bigger than the block of memory; you will have to make sure that the block is big enough.

```
    PROCwimp_putmenuitem(menu%,item%,item$)
    PROCwimp_removemenuitem(menu%,item%)
```

The parameters are mainly self explanatory. If item% in PROCwimp_putmenuitem is greater that the total number of items+1 then it will just be added onto the end.

As items are added or removed, items below are shuffled down and up respectively.

If you wanted to re-build or re-create a menu from scratch again, but still have the same handle as the last one, then you could call FNwimp_createmenu, which would get another chunk of memory and put the data needed into it. This means that the block of memory with the original menu in is still occupied and therefore wasted. If you do this repeatedly then more and more memory is taken up until your application runs out, crashing it.

A much better way is to use the wimp function PROCwimp_recreatemenu, which updates the data in the block of memory containing the old menu.

```
    PROCwimp_recreatemenu(menu%,menu$)
```

menu% is the handle of the menu to re-create. menu$ is a string to build the menu from, and is in the usual form, eg:

```
    “MyApp/Info/Quit”
```

The number of items in the new menu shouldn’t be greater than the specified maximum value when FNwimp_createmenu was called. For instance, the following would probably crash the application, because not enough memory has been allocated for all the items in the larger re-created menu. e.g.:

```
    menu%=FNwimp_createmenu(“MyApp/Quit”,1)
    PROCwimp_recreatemenu(menu%,“MyApp/Info/Quit”)
```

When a menu is re-created, all the item attributes like dotted lines, greying out, and ticks are removed. If you want to change one or two menu items to reflect something in your application like: “Save selection” or “Save” depending in this case on whether anything was selected or not, then use PROCwimp_putmenutext instead as the attributes are retained.

If, by recreating your menu an item is moved up or down, then its item% item number which is passed to user function will change accordingly. The top item is always number 1.

Now we come to the last part in this section: FNwimp_createmenu and PROCwimp_recreatemenu have a major limitation: very large menus cannot be built.

If you think about it, the maximum length of a line allowed in BASIC is something like 255 characters. Now, when you create a menu some of these are used up with the function name and other parameters, leaving maybe 210 characters left for the string that the menu is created from. So what happens if you want to create a font menu where the number of fonts could be in the hundreds?

DrWimp provides a very easy solution to this which requires modified versions of FNwimp_createmenu and PROCwimp_recreatemenu.

The solution is to put all the menu items into an array, then build the menu from the array. This lends itself very well to reading in items like font names, or data from support files for your application.

First of all you need to decide the maximum number of items in the menu. If you are creating a font menu, then you can use the SWI “Font_ListFonts” to find out the number of fonts. Anyway, add one onto the total size, and DIM an array.

The first element of an array (number 0) is the menu title. The last item must be “END”, so DrWimp knows how much of the array to use.

- Note: “END” will not appear as a menu item; the element before it will be the last one.

Here is some example code:

```
DIM menu$(20)
menu$(0)=“MyApp”
menu$(1)=“Info”
menu$(2)=“Quit”
menu$(3)=“END”
barmenu%=FNwimp_createmenuarray(menu$(),20)
PROCwimp_attachsubmenu(barmenu%,1,info%)
```

FNwimp_createmenuarray is passed the name of the array (with empty brackets) and the maximum number of items. The last parameter is exactly the same as for FNwimp_createmenu, so it could just as easily be “0”, which means that more items can’t safely be added, but it makes more sense to set it as the size of the array.

Menus can be re-built using arrays as well. Instead of using PROCwimp_recreatemenu, use:

```
PROCwimp_recreatemenuarray(menu%,array$())
```

Menus created with a string can be recreated with an array, and vice-versa. Note: menus created with arrays can be manipulated using the same functions, such as PROCwimp_putmenuitem, PROCwimp_menuwrite, etc. The only difference is the way that the data to build the menu initially is stored (string or array).

16. Multitasking operations

This section is about multitasking raytracing, calculating numbers, loading data, file finding, etc. No, I am not going to show you how to write a raytracer, etc. but show you how to make operations like these multitask easily.

At the moment, if you wanted to do a multitasking operation, then you would have to set NULL to TRUE so PROCuser_null is called continuously, and every time it is called, remember where you were up to and do a small bit. This can make very tangled code. The difficulty lies in storing where you got up to, and this can require a multitude of variables for just one operation.

A much easier method is to use PROCwimp_singlepoll. When called, it goes through the polling loop once. Your operation will already be in some sort of loop, so all you have to do is call PROCwimp_singlepoll inside it! This very simple technique makes

powerful multitasking operations very easy to achieve.
Change PROCwimp_menuselection so it has lines like:

```
CASE menu% OF
WHEN barmenu% :
  CASE item% OF
  WHEN 1 : PROCchangeauthor
  ENDCASE
ENDCASE
```

And add the following function at the end of !RunImage:

```
DEF PROCchangeauthor
FOR L=1 TO 200
  PROCwimp_singlepoll
  a$=“”
  FOR M=1 TO 6
    a$+=CHR$(RND(26)+64)
  NEXT M
  PROCwimp_puticontext(info%,3,a$)
NEXT L
ENDPROC
```

Now run !MyApp and choose the first item on the iconbar menu. If you now look at the info window, the author field should be constantly changing with random letters. You can still use the desktop, and even quit the application.

Note: PROCwimp_singlepoll acts just like PROCwimp_poll. If one of your icons is clicked on, then PROCwimp_mouseclick is still called, and if your application received messages, then they are acted on, and so on.

Also note that there is another function that can be used for polling; PROCwimp_pollidle. If you remember, if NULL=TRUE then every time that the wimp is polled and no events have occurred, PROCuser_null will be called. If you used the following instead:

```
PROCwimp_pollidle(30)
```

then PROCuser_poll will be called only every 30 seconds. This reduces the load on the processor and is best used for things like clocks, etc, where you would only need to update the clock once a second or minute.

Note: if you have put a banner up and then used PROCwimp_pollidle with a time longer that the banner period, then the banner will stay up until the next PROCuser_null.

There is also the dual of PROCwimp_pollidle, PROCwimp_singlepollidle, which is the same but polls the wimp only once instead of repeatedly.

17. “Grubby” tasks

You will sometimes see tasks that load onto the iconbar, but when the icon is clicked on they leave the desktop and monotask. When the user has finished, they are returned to the desktop, with the application still loaded onto the iconbar. Acorn calls these tasks “Grubby tasks”, and they are very simple to implement.

Alter PROCwimp_mouseclick so it has a line like:

```
CASE window% OF
WHEN bar% :
PROCwimp_starttask(“BASIC -quit <MyApp$Dir>.Mono”)
ENDCASE
```

Create a BASIC file called “Mono” inside the !MyApp directory containing the following:

```
MODE12
PRINT “This is monotasking!”
A$=GET$
*DESKTOP
END
```

Re-load !MyApp and click on the iconbar icon. Press a key to return to the desktop. Note: change the mode number to one suitable for your monitor.

You will probably want to mangle up the second BASIC file as well as !RunImage with DrWimp to give you more security. This is possible if you don’t use DrWimp in the second BASIC file. Mangle it up with !MakeApp2 and !Crunch in the usual way, and then change the PROCwimp_starttask to something like:

```
PROCwimp_starttask(“Run <MyApp$Dir>.Mono”)
```

18. Bars

When you format a disc a bar increases in size to show the amount of the disc that has been formatted so far. When you look at the free space on a floppy or hard drive you have several bars to show you how much space has been used up, is free, and there is in total. When you open the task display you are shown lots of bars that depict the amount of memory something is using up.

Using DrWimp it is simple to control bars like those yourself. They can make information look much more attractive than numbers.
DrWimp allows the length (or height) of the bars to be changed. This means that they can be changed all the time, or only just before a window containing them is opened.

Make a fresh copy of !MyApp. From the Tutorial folder, drag the “Template5” file into the !MyApp directory, and rename it as “Templates”. Add the following lines at the start of !RunImage, just above `PROCwimp_poll`:

```
    main%=FNwimp_loadwindow(“<MyApp$Dir>.Templates”,“main”,0)
    bar%=FNwimp_iconbar(“!MyApp”,“”,1)
    barmenu%=FNwimp_createmenu(“MyApp/Quit”,0)
```

and in PROCuser_mouseclick:

```
    IF window%=bar% PROCwimp_openwindow(main%,1,-1)
```

and in FNuser_menu:

```
    IF window%=bar% =barmenu%
```

and in PROCuser_menuselection:

```
    IF menu%=barmenu% AND item%=1 finished%=TRUE
```

If you now double-click on !MyApp you should get an icon on the iconbar with a menu with a ‘Quit’ item. Clicking on the icon should produce a small window with a red bar in it. What we are going to do is set the bar to a random length when it is clicked on.

First we need to know what the maximum length is, so load the templates into !TemplEd by dropping the file onto !TemplEd’s iconbar icon.

Open the main window by double-clicking on it in the window at the top left. Expand the icon info window at the top right to full size and move the pointer over the bar.

The icon info window gives the dimensions of 340x36, so the max length is 340. Of course we could extend the icon to whatever size we want using !TemplEd, and then using that length.

Take a look at all the details of the bar icon by double-clicking on it. This is how you should set up any icons you want to use as bars. Obviously you can change the colour and turn the border on, etc.

Returning to !RunImage, add the following line to PROCuser_mouseclick:

```
    IF window%=main% PROCchangelength
```

Now add the following function to the end of !RunImage:

```
    DEF PROCchangelength
    len=RND(340)
    PROCwimp_bar(main%,1,len,0)
    ENDPROC
```

Re-load !MyApp, and click on the bar or frame icon behind it.

It is quite easy to specify the length as a percentage. Alter PROCchangelength to:

```
    DEF PROCchangelength
    len=RND(100)
    len=(340/100)*len
    PROCwimp_bar(main%,1,len,0)
    ENDPROC
```

As you can see, len is a percentage chosen at random.

And just to finish off, alter PROCchangelength to:

```
    DEF PROCchangelength
    pcent=0
    REPEAT
      nlen=(340/100)*pcent
      PROCwimp_bar(main%,1,nlen,0)
      PROCwimp_singlepoll
      pcent+=2
    UNTIL pcent>100
    ENDPROC
```

You should be able to see that it is now the basis for a multi-tasking operation with the percentage done depicted by the bar. Put the operation inside the loop, and each time round the loop calculate the percentage done instead of incrementing it as I have done.

You can change the bar to look like however you want it, but I would advise against adding any text, sprites, or indirected text.

One thing you might like to do is add a border around the bar by clicking on the “Border” icon in the relevant !TemplEd window. However, if the bar is going to be changing in size rapidly then the part of the border at the right edge will flicker a lot as that part of the screen is constantly redrawn.

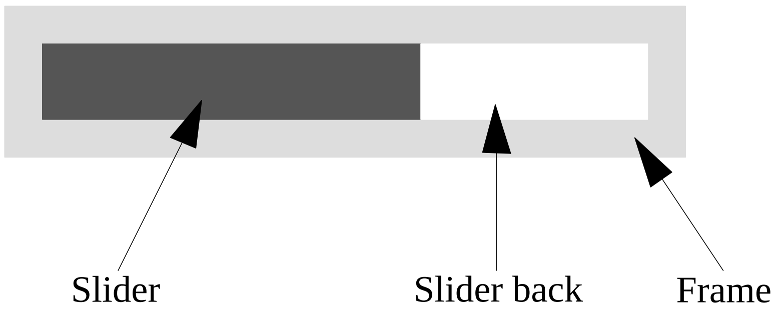
If you want the bar to be a vertical one, i.e. it resizes vertically instead of horizonally, then set the fourth parameter of PROCwimp_bar to 1 instead of 0.

Finally, take a look at !Bar in the Examples folder.

19. Sliders

Sliders can be very useful. You see them probably most in colour selection windows, where you can use them to drag amounts of red, green and blue.

Sliders consist of three icons, and must be constructed in a certain way in order to work:



The slider back icon goes completely under the slider icon, and defines the area over which the slider can be dragged.

The slider and slider back icons must both have a button type of “Click/Drag”, be filled and have no borders. They however can be any thickness, length or colour you like.

Look at how the sliders are constructed in the supplied templates file if it is still not quite clear.

Copy the file “Template6” from the Tutorials folder into !MyApp and rename to “Templates”.
After the line where the main window is loaded in add:

```
    slide%=FNwimp_loadwindow("<MyApp$Dir>.Templates","slide",0)
```

Alter the “window%=bar%” part in PROCuser_mouseclick so that is is like:

```
    IF window%=bar% PROCwimp_openwindow(slide%,1,-1)
```

Run !MyApp and you will see that the slider window that appears has two sliders in it. Currently they do nothing.

Just like FNuser_savefiletype makes save windows work as soon as some value is returned for them, sliders work as soon as you return relevant values from two user functions.

FNuser_sliderback must return the icon number of the slider back icon, when given the slider icon number. For the top slider (loading the templates into !TemplEd will show this), the slider is icon number 2, and the slider back icon is icon number 1. So add the following line to FNuser_sliderback:

```
    IF window%=slide% AND icon%=2 =1
```

FNuser_slider must return the icon number of the slider icon, when given the slider back icon number. Add the following line to FNuser_slider:

```
    IF window%=slide% AND icon%=1 =2
```

Hopefully that should be clear, and running !MyApp will now enable you to click on the top slider or the slider back to make it jump to various positions. You can also drag the slider left and right.

A slider is no good unless you can get a value for it. When a slider is moved or dragged, PROCuser_slidervalue is called, and the percentage of the slider concerned is passed. So add the following lines to it:

```
    IF window%=slide% AND icon%=2 THEN
      PROCwimp_puticontext(slide%,3,STR$pcent%)
    ENDIF
```

STR$pcent% converts the percentage to a string, suitable for passing to PROCwimp_puticontext.

Icon (2) is the slider icon. Run the application now and you will see the percentage displayed in the box on the right.

Add the following line to FNuser_sliderback:

```
    IF window%=slide% AND icon%=6 =5
```

and this line to FNuser_slider:

```
    IF window%=slide% AND icon%=5 =6
```

and finally these lines to PROCuser_slidervalue:

```
    IF window%=slide% AND icon%=6 THEN
      PROCwimp_puticontext(slide%,7,STR$pcent%)
    ENDIF
```

and the bottom slider should now work, so you can see how lots of sliders in lots of windows can be used.

The percentage can be scaled up or down so different ranges can be used, e.g. 0-255.

The percentage of a slider can be read using FNwimp_getsliderpcent and the percentage of a slider can be set using PROCwimp_putsliderpcent. Try these out to see their effect. Vertical sliders can also be made. If the slider is made higher than it is wide, then DrWimp will automatically assume its a vertical slider.

20. Loading & Saving Drawfiles

I am not going to add drawfiles to !MyApp here as the example in the Examples folder shows how it is all done without the clutter we have added. Instead I will cover some points that are worth mentioning.

Recall that a handle points to the start of a memory block. Drawfiles are all loaded into the same block of memory and so in order to have a handle to each one, the handles point to the start of the drawfiles in the block.

Using one block of memory makes the programming much easier than having to cope with lots of blocks, one for each drawfile, but it makes the loading in of drawfiles seem a little over complicated.

However if you follow and extend the methods used here then everything will work fine, and you don't really need to understand it all.

Loading in 3 drawfiles, which are inside !MyApp:

```
    PROCwimp_initdfiles
    path1$="<MyApp$Dir>.Drawfile1"
    path2$="<MyApp$Dir>.Drawfile2"
    path3$="<MyApp$Dir>.Drawfile3"
    size%=0
    size%+=FNwimp_measurefile(path1$)
    size%+=FNwimp_measurefile(path2$)
    size%+=FNwimp_measurefile(path3$)
    DIM drawfiles% size%
    DIM dfilehan%(3)
    dfilehan%(1)=drawfiles%
    dfilehan%(2)=FNwimp_loaddfile(path1$,dfilehan%(1))
    dfilehan%(3)=FNwimp_loaddfile(path2$,dfilehan%(2))
    d%=FNwimp_loaddfile(path3$,dfilehan%(3))
```

So you end up with dfilehan%(1) is the handle for the first drawfile, dfilehan%(2) is the handle for the second, etc.

First of all FNwimp_measurefile is used to add up the total sizes of all the drawfiles, so the size of block of memory to create is known. Then the block (called drawfiles%) is made.

An array (dfilehan%) is made to hold the handles, and the first handle is the start of the block of memory (drawfiles%).

The other handles are all obtained from FNwimp_loaddfile, which loads in the drawfile. Passed to it is the handle (ie the position in the block as to where it should be loaded) and returned is the handle to the next drawfile.

Upon loading the final drawfile we don't need to know the handle for the next, and indeed there will be no more room in the block to load any more, so it is stored in an arbitrary variable (I used d%), and forgotten about.

So, to load in more drawfiles, include them in the size totalling part, then dim the handle array to the right size, then expand the list of FNwimp_loaddfile calls to the required number.

Never try to work out the sizes of the drawfiles using any other method but FNwimp_measurefile, otherwise you may run into trouble.

As a final example, how to load in a single drawfile:

```
PROCwimp_initdfiles
path1$="<MyApp$Dir>.Drawfile1"
size%=0
size%+=FNwimp_measurefile(path1$)
DIM drawfiles% size%
DIM dfilehan%(1)
dfilehan%(1)=drawfiles%
d%=FNwimp_loaddfile(path1$,dfilehan%(1))
```

You could in this case remove the dfilehan% array and just have drawfiles% as the handle, but that is up to you to choose which you prefer.

Drawfiles can easily be saved using PROCwimp_savedfile. Eg. to save the first drawfile to a path stored in path$:

```
PROCwimp_savedfile(path$,dfilehan%(1))
```

21. Rendering Drawfiles & Sprites

Drawfiles and sprites can be rendered straight on the screen at a specified position or in a window. The latter is much more useful, but is a derivative of the former.

Drawfile Text areas are not rendered as they are rarely used and very complex. Text column objects are therefore not rendered either.

The rendering is “simple”. i.e. no rotation, scaling, shearing, however some may be added to DrWimp in later versions.

PROCwimp_render renders drawfiles onto the screen. The coordinates of the bottom left corner are supplied, and two pairs of coordinates to define a clipping rectangle. Any objects lying totally outside the clipping rectangle are not rendered.

PROCwimp_rendersprite is the sprite equivelent and works in exactly the same way, but instead of having a drawfile handle, there is the sprite name and the sprite area handle which contains that sprite.

PROCwimp_renderwindow renders a drawfile inside a window, and is called in PROCuser_redraw like:

```
IF window%=main% THEN
  PROCwimp_renderwindow(main%,dfilehan%(1),50,-50,minx%,miny%,
maxx%,maxy%)
ENDIF
```

See the example application in the Examples folder for a real example of this. Note that in order for PROCuser_redraw to be called for a window, it must have its “auto-redraw” flag unset from the template editor.

As an example for rendering a sprite (assuming it has already been loaded in and is in a sprite area whose handle is sprite%):

```
    IF window%=main% THEN
    PROCwimp_renderwindowsprite(main%,"test",sprite%,50,50,
                                            minx%,miny%,maxx%,maxy%)
    ENDIF
```

FNwimp_getdfilesize returns the width and depth of a drawfile so a window can be resized to the size of the drawfile, and then displayed in it. The equivalent for sprites is FNwimp_getspritesize.

22. Saving time

You will find that after writing a few applications, in most if not all you are having to find a templates file with an info window in it, rename the sprites and all occurrences of “MyApp” to your application's name, create an iconbar icon and iconbar menu, fill in the details in the info window, set the version, etc, etc.

I have found that doing that over and over again gets tedious, hence the utility !Fabricate in the Utils folder. It can do all those things for you and save boredom setting in, leaving you to get on with writing the interesting parts.

Full instructions are included in the !Help file inside !Fabricate.

23. Validation Strings

Each icon has a validation string. This can be empty or contain some options which are separated by semicolons.
The validation string for an icon is set in a template editor when editing the icon, and for those with the RISC OS 3 Programmers Reference Manuals all the options can be found on page 3-102.

The most common option is r*n*, where *n* is a number from 0 to 7, which specifies the border type. For example r6 is the default action type border and r7 is a writable icon type border. r0 is equivalent to not using the r option.

If the icon can be pressed in then the “slab in” colour can be set. If you create a default action button in TemplEd you will see that the validation string is r6,3. Try changing the 3 to 7 or 11.

Another very common option is P*spritename*, where *spritename* is the name of a sprite for the pointer when the pointer is over that icon. Usually this is something like Pptr_write which changes the pointer to a caret shape over writable icons.

If you have lots of writable icons in a window then you can make it so that the caret can be moved between them with the up and down arrow keys, Tab and Return. If you put Ktar in the validation string of all your writable icons then they can all be navigated in that way. t - Tab, a - arrows, r - Return. You can use any combination you like.

The caret will move in order of icon number so if you have writable icons numbered 3,11,7,8,2 then it will move in the order: 2,3,7,8,11 for the down arrow, tab or return and 11,8,7,3,2 for the up arrow. So make sure that if you are going to use this then your writable icons are numbered in such a way that the caret will move in a predictable manner.

As mentioned, more that one option can be used for each icon if they are separated by semicolons. So for a typical writable icon you may have: r7;Pptr_write;Ktar.

24. Indirection

Throughout this manual, you will have seen lots of references to “Indirected Icons”. Here is a brief description of what it means, and why they are important to using the Wimp.

Each icon on the desktop is defined by a small block of memory. This block is of a fixed size and contains a complete description of the icon so the Wimp can draw it.

This block contains the dimensions, a load of flags containing information such as whether it is a sprite, text, sprite and text, filled, has a border, etc and some data on what it contains, such as actual sprite name or text.
Most of the flags you can toggle between set and unset using a template editor, such as TemplEd.

The total size of the icon block is 32 bytes. 12 of these are used for holding the data. If the icon holds text or is a sprite using the “S” validation code then the maximum length of the string must be 11 (plus a terminator). If you want to hold more than this then you must make the icon indirected by setting the flag in a template editor, and making the value of the indirected field the maximum number of characters you wish to hold. The text is then stored elsewhere and a pointer to it is placed in the data section of the icon block.

Once an icon has been created (ie. your program has loaded in the window containing the icon) then you can only change certain things in the icon block. You cannot change the contents of the data section. However, if the icon is indirected then you can change the text or sprite name because it is not actually held in the icon block but in another block of memory elsewhere.

If you make an icon indirected with an indirected size of 16, then use PROCwimp_puticontext to put a string greater than 15 characters long (15 plus terminator equals 16), strange things may happen. The most common is that if the icon has a 3D border then it is lost. The more you go over the limit, the messier things get.

25. Text Handling

You should, by now, be able to see how to plot drawfiles and sprites in a window (using PROCuser_redraw). With DrWimp it is also possible to plot text in a variety of ways. Any outline font can be used at any size and in any colour.

Some of the functions require the name of the font to be provided. The names always have to be what I call *period seperated*. For example:

```
    "Trinity.Medium"
    "Homerton.Bold.Oblique"
    "Corpus.Medium.Oblique"
```

Instead of having spaces, there are periods (full stops).

PROCwimp_plottext is the most besic function provided, and simply plots a string of text on the screen at the required position. You pass to it the name of the font you want to use, the point size and foreground and background colours.

PROCwimp_plotwindowtext does the same but plots the text in a window, so you would use this in PROCuser_redraw like PROCwimp_renderwindow and PROCwimp_renderwindowsprite.

Fonts can have handles, just like windows, drawfiles, sprites, etc. In fact that is how the wimp uses fonts. Whenever PROCwimp_plottext or PROCwimp_plotwindowtext is called, a handle for the font is found, the string is plotted, then the handle is returned. If you are plotting many lines of text, all this handle finding and returning can waste a lot of time. DrWimp provides a way of optimising this by providing a wimp function to find the handles for you, use slightly different wimp functions to plot the text, then allow you to return the handles.

FNwimp_getfont returns a handle when you tell it the font you require and the point size. You would normally get all the handles you require when your application loads, before PROCwimp_poll. A 0 is returned if the font could not be found so you would have to take into account of this and perhaps find an alternative and warn the user.

```
     trinity12% = FNwimp_getfont("Trinity.Medium",12)
```

To plot text, PROCwimp_plottexth and PROCwimp_plotwindowtexth are used. Note the “h” on the end of the function name to denote that those functions accept font handles instead of fonts names and sizes.
PROCuser_redraw example:

```
     IF window%=main% THEN
     PROCwimp_plotwindowtexth(main%,"Test",trinity12%,50,50,0,0,0,
                                 255,255,255,minx%,miny%,maxx%,maxy%)
     ENDIF
```

Note the foreground and background colours. 0,0,0 sets the foreground to black, and 255,255,255 sets the background to white.

Finally, when you have finished with a font, you call PROCwimp_losefont. For example:

```
     PROCwimp_losefont(trinity12%)
```

This would normally be called when you application is quitting, like after the PROCwimp_poll, but before the END.

Not just simple strings of text can be plotted. Strings of control code sequences can be inserted into the string to turn underlining on and off, change the font, or change the font colour. DrWimp provides functions to produce these control strings. For example:

```
     t$="Cat "+FNwimp_underline(1)+"dog"+FNwimp_underline(0)+" hen."
```

when plotted would result in the word “dog” being underlined. Note the positioning of the spaces between the three words so they don't get underlined as well.
An example of turning the text red:

```
     t$="Cat "+FNwimp_fontcolour(255,0,0)+"dog."
```

would make the word “dog.” turn red. The text can be turned black again afterwards with 0,0,0. And for changing the font (assuming homerton20% is a font handle already found):

```
     t$="Cat "+FNwimp_fontchangeh(homerton20%)+"dog."
```

would make the word “dog.” appear in the font whose handle is homerton20%.

Note that there isn't an equivalent FNwimp_fontchange (no “h”) as the control sequence only works with font handles.

Any combination of these can be used to produce the effect you want. Note that if you plot a string and the end of it is in red for example, then the next line you plot will be in black again. There is no need to change the colour back to black as all effects such as colours, underlining and changing font only apply for that line only.

There are two functions for finding the width and height of a string, if it were plotted on the screen. They are FNwimp_gettextsize and FNwimp_gettextsizeh, with the latter using font handles instead of font names and sizes.

The last two functions also plot text, both on the screen and in windows, but they don't always plot with outline fonts. They allow text to be plotted using the current desktop font. On pre RISC OS 3.5 machines this is always the system font, so that is what will be used. On RISC OS 3.5+ it is the outline font that is currently being used for the desktop, which may be the system font.

These allow things like annotations or lists of choices to be constructed, knowing they will match the desktop and look like they are text in icons.

The functions are PROCwimp_deskplottext and PROCwimp_deskplotwindowtext. You supply the string to plot, the position and the foreground and background colours. There is an extra parameter which allows you to choose whether to plot with the left side of the text at the x-coordinate, or plot with the text centred on the x-coordinate. They plot stright to the screen and to a window respectively.

26. Printing

This section assumes you have a fairly good grasp of how to use DrWimp. As there a quite a bit to set up and nothing can be tested until its mostly all done, there is no tutorial for this section. However there are some example applications in the Examples folder.

As you have seen, it is possible to plot text, sprites and drawfiles in windows by doing the revelant work in PROCuser_redraw. But what if you want to print it out? For a start the work area has its origin at the top left of a window, making all y coordinates negative. Paper has its origin at the bottom left corner of the paper. Plus it seems that if you want to print out what you have described in PROCuser_redraw, you will have to duplicate it all again. Then there is the problem of fitting lots of pages onto one page by reducing them in size and rotation.

DrWimp makes it relatively easy to print out documents, even using what you have already written for PROCuser_redraw, but you will have to make some alterations to the coordinate calculations, which are fairly strightforward.

There are two methods you can use to print out and which you choose depends on the application. If you want to print out what is being shown in a window because it has all be written in PROCuser_redraw, then you use what we will call the “redraw” method. If on the other hand you want to print out something that isn't displayed, and only needs to be described for the printing then you do all the work in a user function, PROCuser_print. We will call this the “user” method.

DrWimp also provides facilities to find out about the paper, printer driver, convert between work area coordinates and screen coordinates, and also provides a means for the application to tell the user what is going on and giving them the option to cancel the printing.

The first thing your application needs to do is have some sort of print window where the user can specify the range of pages to print, how many to fit onto a page and the number of copies to print. Now its not a good idea to try and print out when there is no printer driver loaded as your application will produce errors and probably quit, so you have to check to see if a printer driver is actually loaded. For example:

```
loaded%=FNwimp_pdriverpresent
```

loaded% will equal 1 if a printer driver is loaded, and a 0 if one isn't.

Users can load and quit printer drivers at any time and they may be present or not whilst your application is runing, so you need to continually check. The best time to do this is just before your print window is opened. So you might have a function that does the check and you always call it just before every PROCwimp_openwindow or PROCwimp_openwindowat for your print window.

Depending on the result of the check you should disable some icons in the print window. For example the icon the user clicks on to initiate the printing. Also a common practice is to put the name of the current printer in the titlebar of the window or somewhere in the window. So for example you could use:

```
loaded%=FNwimp_pdriverpresent
IF loaded%=1 THEN
  printername$=FNwimp_getpdrivername
  PROCwimp_putwindowtitle(print%,printername$)
ELSE
  PROCwimp_putwindowtitle(print%,"No printer loaded")
ENDIF
```

with print% being the handle of your print window. Its easy to see that in that IF..ELSE..ENDIF structure you can enable and disable icons in the print window depending on whether the printer driver was loaded or not.

If you wish you can also get information about the paper size and the borders (which you can't print in), to display in the print window. You could even draw two small grey and white rectangles to depict the page and its borders. But be careful not to try and get any information about the page unless a printer driver is loaded!

Here is a sample piece of code that could go in PROCuser_redraw for the print window (assuming its auto redraw flag is off):

```
IF window%=print% THEN
  IF FNwimp_pdriverpresent THEN
    w%=FNwimp_getpapersize(0,0)
    h%=FNwimp_getpapersize(1,0)
    IF h%>=w% scale=h%/120
    IF w%>h% scale=w%/120
    w%=w%/scale
    h%=h%/scale
    x%=FNwimp_worktoscreen(print%,316,0)
    y%=FNwimp_worktoscreen(print%,-256,1)
    PROCwimp_setcolour(160,160,160)
    RECTANGLE FILL x%,y%,w%,h%
    lm%=FNwimp_getpapersize(0,1)/scale
    r%=FNwimp_getpapersize(0,3)/scale
    bm%=FNwimp_getpapersize(1,1)/scale
    t%=FNwimp_getpapersize(1,3)/scale
    PROCwimp_setcolour(255,255,255)
    RECTANGLE FILL x%+lm%,y%+bm%,r%-lm%,t%-bm%
  ENDIF
ENDIF
```

which will draw a small depiction of the paper in the window with borders. It may be slow though due to it having to find out if a printer driver is loaded every time a bit of the window needs redrawing, although you could optimise it to take into account the redraw clipping rectangle passed to PROCuser_redraw.

Currently DrWimp supports printing ranges of pages, multiple copies and fitting one, two or 4 A4 pages onto a single physical A4 page, so these are the options you can provide in your print window. For an example see the template file “Print” in the tutorials folder.

At any time the user could change the current printer, and PROCuser_printerchange will be called. When it is, you much check to see if the printer driver is still loaded, and update all your page measurements, what icons to enable and disable in the print window etc. Basically do all your checks again just as if the print window was being opened. Don't actually open your print window unless its already open, in which case it should update itself.

If you are printing with outline fonts, which is probably very likely, then there is one more consideration to take into account before we can start printing. Postscript printers either have fonts with different names or only a small selection of fonts, so some printer drivers, particually postscript ones may require all fonts that are going to be used, to be declared first. As we don't know what printer the user is going to use, or if they require fonts to be declared, we must always declare them. This is done in the user function PROCuser_declarefonts.

If you have any drawfiles you are going to print out that contain fonts, then the fonts in them also need to be declared.

In the user function PROCuser_declarefonts you must call PROCwimp_declarefont if you want to supply the font name, or PROCwimp_declarefonth if you want to give the font handle instead, for each font. For drawfiles you call PROCwimp_declaredfilefonts, giving the handle of the drawfile, for each drawfile. For example:

```
DEF PROCuser_declarefonts
  PROCwimp_declarefont("Trinity.Medium")
  PROCwimp_declarefonth(homerton18%)
  PROCwimp_declaredfilefonts(dfilehan%(1))
ENDPROC
```

See the section on text handling for more information on font handles and period seperated font names, and see the section on loading drawfiles for more information on drawfile handles.

Once you have everything set up you are ready to print and one wimp function initiates it.

```
PROCwimp_print(user%,window%,fpage%,lpage%,perpage%,copies%)
```

Setting user% to 1 prints using the user method, and setting it to 0 prints using the redraw method. Both of these will be discussed in more detail later.

If using the redraw method then window% is the handle of the window to redraw. In other words the window whose contents you want to print out. window% is ignored if printing with the user method.

fpage% and lpage% are the page numbers of the first page to print and the last page, respectively. These are inclusive, and don't really matter to DrWimp what they are as long as lpage% is equal to or larger than fpage%. The numbers you specify for the range of pages are arbitary as far as DrWimp is concerned, as the whole range of page numbers will be returned to you one by one as you are requested to print each page.

perpage% can currently be either 1, 2, or 4. If you set it to 1 then each page will fit on a physical A4 page. If you specify 2 then two A4 pages will be scaled to about 70%, rotated through 90 degrees, and placed side by side on a physical A4 page. If you specify 4 then each page will be scaled to 50% and four pages will be printed on a single physical A4 page. All this is done automatically and the printing as far as you are concerned is exactly the same, just that the actual output is affected.

copies% is the number of copies to print. For example if you want to print from page 1 to 4, with two copies, then eight pages will be printing, two lots of four. The number of physical pages printed depends on perpage% though.

During printing you may wish to keep the user informed of the progress of the printing and give them the option to cancel. This is easily achieved as FNuser_printing is repeatedly called. From it you can read the current copy being printed, the current page number, the total number of pages that are being printed, and the current page being printed (from one to the total number).

From the total number of pages and the current page being printed, you can calculate the percentage of pages already printed and display a progress bar using PROCwimp_bar. You can also display the current copy and current page out of interest.

Finally, in the progress window you can add a cancel button, and when it is clicked on you set a variable to indicate that the user wishes to cancel printing. Then, when PROCuser_printing is next called, you look at the variable and return a 1 if the printing is to be cancelled. Otherwise you return a 0.

Here is an example of the sort of thing you might put into the user function:

```
DEF FNuser_printing(copy%,page%,totpages%,pagepos%)
PROCwimp_puticontext(prog%,0,“Printing page ”+STR$page%+“ (copy
                                                 ”+STR$copy%+“)”)
PROCwimp_bar(prog%,2,(pagepos%/totpages%)*466,0)
=cancel%
```

In the example, prog% is the handle to the progress window, the progress bar is icon number 2, which is 466 OS units long, and icon 0 is just a text icon to display information.

cancel% is set to 0 just before PROCwimp_print was called, and in PROCuser_mouseclick there would be a line like the following:

```
    IF window%=prog% AND icon%=4 cancel%=1
```

where icon 4 is the cancel button. FNuser_printing can then simply return cancel% to indicate if printing is to be cancelled or not.

Just as a couple of examples so you understand the difference between page% and pagepos%:

Say the user wanted to print pages 2 to 8, one copy. Then as printing progresses copy% will stay at one, page% will start at 2 and increase to 8, totpages% will stay at 7 (printing pages 2 to 8 inclusive), and pagepos% will start at 1 and increase to 7 (totpages%).

If the user was printing pages 4 to 6, 2 copies, then copy% will start at 1 and increase to 2, page% will start at 4 and increase to 6, totpages% will stay at 3, and pagepos% will start at 1 and increase to 3 (totpages%).

Now we come to the final part of printing, the actual constructing of the pages. First the redraw method.

When PROCuser_redraw is called and printing is occuring, the variable printing% will be TRUE, and page% will contain the page number to print. Lets take a simple example of plotting one line of text in trinity medium at 12pt. We will assume a handle for the font has already been obtained and is in trinity12%. The window we are plotting in has the handle main% (ie. the window handle passed to PROCwimp_print was main%), and the text is placed at the work area coordinates 50,-100.

```
    DEF PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
                                                printing%,page%)
    IF window%=main% THEN
      x%=50:y%=-100
      string$="DrWimp printing test."
      PROCwimp_plotwindowtexth(main%,string$,trinity12%,x%,y%,
                          0,0,0,255,255,255,minx%,miny%,maxx%,maxy%)
    ENDIF
    ENDPROC
```

Now take a look at how it would be modified to allow the contents of the window main% to be printed.

```
DEF PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
                                                printing%,page%)
IF window%=main% THEN
  x%=50:y%=-100
  IF printing%=TRUE THEN
    x%=FNwimp_worktopaper(x%,0)
    y%=FNwimp_worktopaper(y%,1)
  ENDIF
  string$="DrWimp printing test."
  PROCwimp_plotwindowtexth(main%,string$,trinity12%,x%,y%,
                     0,0,0,255,255,255,minx%,miny%,maxx%,maxy%)
ENDIF
ENDPROC
```

And thats it! The wimp function FNwimp_worktopaper should be self explanatory.

Now, take a look at this next example:

```
DEF PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
                                                printing%,page%)
IF window%=main% THEN
  x%=50:y%=-200
  PROCwimp_setcolour(255,0,0)
  x%=FNwimp_worktoscreen(main%,x%,0)
  y%=FNwimp_worktoscreen(main%,y%,1)
  RECTANGLE FILL x%,y%,100,100
ENDIF
ENDPROC
```

which will draw a red square with the bottom left corner at the work area coordinates 50,-200. Now here is the modified version which will allow the square to be printed out:

```
DEF PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
                                                printing%,page%)
IF window%=main% THEN
  x%=50:y%=-200
  PROCwimp_setcolour(255,0,0)
  x%=FNwimp_worktoscreen(main%,x%,0)
  y%=FNwimp_worktoscreen(main%,y%,1)
  IF printing%=TRUE THEN
    x%=FNwimp_screentopaper(main%,x%,0)
    y%=FNwimp_screentopaper(main%,y%,1)
  ENDIF
  RECTANGLE FILL x%,y%,100,100
ENDIF
ENDPROC
```

the screen coordinates are converted into paper coordinates via the windows work area coordinates. Essentially a screen -> work -> paper conversion.

One thing to note is that the top left corner of the window main% matches up with the top left corner of the paper.

Now for an example of printing the page number at the bottom of each page. You can work out the work area coordinates and/or the page coordinates to place the text, but the following example assumes the work area of the window main% has been set to the size of an A4 piece of paper. That can be done with:

```
width%=FNwimp_lengthtoOS(210,100,0)
height%=FNwimp_lengthtoOS(297,100,0)
PROCwimp_resizewindow(main%,width%,height%)
```

The FNwimp_lengthtoOS call converts 210mm and 297mm at 100% scale to OS units. The window is then resized. The example also assumes the programmer is keeping track of which page is being displayed with currentpage%. Now for the example:

```
DEF PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
                                                printing%,page%)
IF window%=main% THEN
  x%=50:y%=FNwimp_getwindowworksize(main%,1)+150
  IF printing%=TRUE THEN
    x%=FNwimp_worktopaper(x%,0)
    y%=FNwimp_worktopaper(y%,1)
  ENDIF
  string$="Page "+STR$currentpage%
  PROCwimp_plotwindowtexth(main%,string$,trinity12%,x%,y%,
                      0,0,0,255,255,255,minx%,miny%,maxx%,maxy%)
ENDIF
ENDPROC
```

The text is plotted 150 OS units up from the bottom of the actual paper to allow for printer borders, and to give a bit of space. As stated before, currentpage% contains the number of the page currently being displayed.

There is one problem with this however. It will be displayed fine, but when it comes to printing out, every time currentpage% will contain the page number of the current page being displayed, so it won't change on paper. What is required is a way of using the variable page% passed to PROCuser_redraw to make currentpage% change, but without corrupting it, so when the printing has finished the redraw won't try to show a different page. Here is one method:

```
DEF PROCuser_redraw(window%,minx%,miny%,maxx%,maxy%,
                                              printing%,page%)
IF window%=main% THEN
  IF printing%=TRUE c%=currentpage%:currentpage%=page%
  x%=50:y%=FNwimp_getwindowworksize(main%,1)+150
  IF printing%=TRUE THEN
    x%=FNwimp_worktopaper(x%,0)
    y%=FNwimp_worktopaper(y%,1)
  ENDIF
  string$="Page "+STR$currentpage%
  PROCwimp_plotwindowtexth(main%,string$,trinity12%,x%,y%,
                    0,0,0,255,255,255,minx%,miny%,maxx%,maxy%)
  IF printing%=TRUE currentpage%=c%
ENDIF
ENDPROC
```

If PROCuser_redraw is being called for printing (printing% is TRUE) then currentpage% is stored in c% for safekeeping, and set to the number of the page being printed. Then when the redraw has finished currentpage% is restored to what it was initially.

Now for the user method of printing. Instead of PROCuser_redraw being called, PROCuser_print is called instead, and the page is described in much the same way. As there is no window involved you do not need to convert between work area, screen and paper coordinates. You simply work in paper coordinates, which are exactly the same as OS units, with the origin at the bottom left corner of the paper. For example, to print a black square, slightly in from the bottom left corner of the paper:

```
DEF PROCuser_print(minx%,miny%,maxx%,maxy%,page%)
PROCwimp_setcolour(0,0,0)
RECTANGLE FILL 100,100,50,50
ENDPROC
```

You can be more precise with your position in “real” units, for example:

```
DEF PROCuser_print(minx%,miny%,maxx%,axy%,page%)
inch%=FNwimp_lengthtoOS(1,100,1)
4cm%=FNwimp_lengthtoOS(40,100,0)
PROCwimp_setcolour(0,0,0)
RECTANGLE FILL inch%,inch%,4cm%,4cm%
ENDPROC
```

will print out a black square one inch from the bottom of the page and one inch from the left edge of the page, with sides 4cm long.

With PROCuser_print there is no printing% or currentpage% variables to use because it is only called when printing is taking place, and page% holds the current page number of the page being printed.

There is a standard clipping rectangle supplied in page coordinates which you can use for optimisation, however I doubt the performance gains would be big enough to warrant the extra effort of using the clipping rectangle.

When developing the printing part of your application you may like to know of the method I used for saving on ink and paper when writing DrWimp. I added a postscript printer to Printers and set it to print to a file, thus creating a postscript file. I then loaded it into the PD application RiScript which allows you to view postscript files. The output could then be used to debug and verify printing code.
One word of warning though, RiScript is very slow. Especially on text. If you have a lot of text on the page you could be waiting a very long time!

27. Window & Icon creation

Creating windows and icons from BASIC instead of loading them from a templates file can have several advantages. From a security point of view, if you secure your code, then no-one can change the windows and icons. Another advantage is that icons can be created dynamically, say for search results, when you want the list to be a set of icons that can be clicked on.

DrWimp provides four functions for creating and deleting windows and icons, which will now be described in detail.

The creation of windows and icons requires the use of flags to describe certain aspects of the window or icon to be created. There are around 9 flags which are set and unset using a single number, which is passed as one parameter to the wimp functions. This method had to be used to keep the parameters concise. Flags for icons are not the same as flags for windows, but are passed in the same way, using a single number.

A flag is represented by a single binary bit, so 8 flags would be represented by 8 bits. For example:

```
    00101101
```

Which is a binary number, with bits 0, 2, 3 and 5 set (1), and the rest unset (0). So if that was used to create a window or icon, flags 0, 2, 3 and 5 would be set (turned on). Note that the right-most bit is called bit 0.

Binary numbers can be converted into decimal and hexadecimal. The decimal number is obtained by adding up each set bit, but with a weighting. The weighting for each bit is 2^i, where i is the bit number.

For example, the binary number we have just looked at is

$2^0+2^2+2^3+2^5 = 45$

(recall that bits 0, 2, 3 and 5 are the only ones set)

To convert to hexadecimal is just the same but the bits are taken in fours which are treated as separate numbers. So the left-most four will equal 2 in decimal, and the right-most four will equal 13 in decimal. In hexadecimal A is used for 10, B for 11, C for 12, D for 13, E for 14 and F for 15. This means the number becomes

2D

in hexadecimal. You can see that each symbol (0-9, A-E) in hexadecimal represents four bits, in the same order.

In BASIC this number could be represented in all three bases as:

```
%00101101
45
&2D
```

So any one of those can be passed as the flags parameter to the wimp functions, as they all represent the same number. You may find the binary easiest to use as you can quickly work out what to put to set and unset flags.

The function to create windows is

```
window%=FNwimp_createwindow(vminx%,vminy%,vmaxx%,vmaxy%,wminx%,
                            wminy%,wmaxx%,wmaxy%,flags%,workcol%,
                            button%,title$,maxind%,sarea%)
```

vminx%, vminy%, vmaxx%, vmaxy% are the screen coordinates that describe the area of screen that the window will cover when opened. If you open it centred or centred on the pointer or at specific coordinates, then the position vminx% and vminy% on the screen is ignored. However, vmaxx%-vminx% and vmaxy%-vminy% set the width and height of the window. The visible area has to be less than the work area. A window cannot be opened larger than its work area!

wminx%, wminy%, wmaxx%, wmaxy% are the work area coordinates that set the work area of the window. For example setting them to 0, -100, 200, 0 will create a window whose work area is 200x100 in size (remember that the work area origin is at the top left so y-values are always negative).

flags% is the number that represents the flags for the window. This is describe in more detail later.

workcol% is the colour of the work area in desktop colours. So it is a value in the range 0-15, with 1 being the usual value, and 0 giving white.

button% is the button type of the work area. It can be 0 to ignore clicks, and 1 to accept clicks (PROCuser_mouseclick will be called for icon -1).

title$ is the title of the window. If it is longer than 12 characters then the title will become indirected.

If the title is indirected, then maxind% is the maximum size that the title can be plus 1.

If sarea% os 0 then the wimp sprite area is used for the window, otherwise sarea% is the handle of a sprite area (just the same as FNwimp_loadwindow).

Flags:

bit:	meaning if set:
0	window has title bar
1	window has close icon
2	window has back icon
3	window has horizontal scroll bar
4	window has vertical scroll bar
5	window has adjust size icon
6	window has toggle size icon
7	window can auto-redraw
8	window is a pane
9	window has indirected title

For example

```
   %1011111001
```

passed as flags% would create a window which has an indirected title, isn't a pane, does auto-redraws (PROCuser_redraw isn't called), has a toggle size icon, has an adjust size icon, has vertical and horizontal scroll bars, doesn't have a back icon or a close icon, but does have a title bar.

Returned from the wimp function is a handle to the window. An example of a basic window is:

```
    main%=FNwimp_createwindow(406,572,790,816,0,-936,1236,0,
                                             %0011111111,1,0,“main”,5,0)
```

The function to create icons is:

```
    icon%=FNwimp_createicon(window%,wminx%,wminy%,wmaxx%,wmaxy%,
                            flags%,esg%,button%,fcol%,bcol%,fhan%,
                            text$,sprite$,sarea%,maxind%,valid$)
```

window% is the handle of the window to create the icon in.

wminx%, wminy%, wmaxx%, wmaxy% are the work area coordinates that define the position of the icon and the size. For example using 8,-48,48,-8 will create an icon that is 8 OS units in from the left edge of the window, 8 OS units down from the top edge, and 40x40 OS units in size.

flags% is the number that represents the flags for the icon. They are described in more detail later.

esg% is the Exclusive Selection Group number for the icon. These are used for radio icons. All other icons should have this set this to 0.

button% is the button type of the icon.

button%:	type:
0	ignore
1	click
2	click autorepeat
3	click/drag
4	writeable
5	radio

fcol% and bcol% are the foreground and background colours of the icon in desktop colours, so they are both in the range 0-15. Usually set to 7 and 1 respectively. Not used if the icon uses an outline font.

fhan% is the handle of an outline font, if once is being used. If not then this should be set to 0.

If the icon has some text, then it is put in text$, and if the icon has a sprite, then its name is put in sprite$. The sprite area handle is put into sarea%, with 0 denoting wimp sprite area.

If the icon is indirected, then maxind% is the maximum size of text that can be put plus

1.

valid$ is the icon validation string, which may contain information such as the pointer to use for the icon, the acceptable characters for writeable icons, the border type, etc.

Flags:

bit:	meaning if set:
0	icon has text
1	icon has sprite
2	text/sprite is horizontally centred
3	text/sprite is vertically centred
4	icon is filled
5	text/sprite is right justified
6	icon has border
7	icon uses an outline font
8	icon is indirected

For example

```
%101001101
```

will create an icon that is indirected, doesn't use an outline font, has a border, doesn't have right justified text, isn't filled, has vertically and horizontally centred text, doesn't have a sprite, but does have text.

The function returns a handle to the icon, which is the icon number.

An example of creating an OK button:

```
okbutton%=FNwimp_createicon(main%,8,-76,136,-8,%101011101,0,1,
                                        7,1,0,“OK”,“”,0,3,“R6,3”)
```

All this may seem like a lot of effort, but using the CodeTemps utility in the Utils folder, supplied with DrWimp, you can convert window templates containing icons into BASIC code which consists of the two wimp functions FNwimp_createwindow and FNwimp_createicon. It can them be dropped straight into your !RunImage. Full instructions on CodeTemps is contained in its !Help file.

Windows may be deleted using something like:

```
PROCwimp_deletewindow(main%)
```

which will delete the window whose handle is main% in this example. If the window was open then it will be closed. One point to note though is that not all memory used by the window is reclaimed. The memory used to store indirected data remains tied up for good, so a lot of creating and deleting windows will eventually result in running out of memory.

Icons can be deleted using PROCwimp_deleteicon, for example:

```
    PROCwimp_deleteicon(main%,2,0)
```

will delete icon 2 from the window main%. The icon will not disappear though as the window needs to be redrawn. This can be done automatically by setting the third parameter to PROCwimp_deleteicon to 1. Eg:

```
    PROCwimp_deleteicon(main%,2,1)
```

28. Bits & bobs

There are other features of DrWimp like user graphics functions etc which are demonstrated by the examples. See their commented !RunImage files.

Brief overview of some functions:

FNwimp_resizewindow resizes a window to the supplied width and height. The work area is set to the new values, and if it is smaller than the current visible area, then the window will shrink.

PROCwimp_colouricontext changes the colour of some text in an icon to one of the 16 desktop colours. This means that you can “grey-out” labels which are not filled. If you do that with PROCwimp_icondisable then the unfilled background turns white.

FNwimp_geticonsize is like FNwimp_getwindowsize in that is returns the dimensions of an icon.

FNwimp_geticonstate is the duel of PROCwimp_iconselect, and returns a 1 if the icon is selected, or a 0 if it isn't.

FNwimp_worktoscreen and FNwimp_screentowork convert between screen coordinates and a window's work area coordinates.

PROCwimp_setcolour sets the current GCOL colour for drawing, and uses dithering, if the exact colour is not available in the current mode.

Blocks can be used as a compact way of storing strings, if you have a maximum length for a string.

```
block%=FNwimp_createblock(10,25)
```

Will create a block that can store 10 strings, any of which can be up to 25 characters in length. block% is the handle to the block.

```
PROCwimp_putinblock(block%,"A string",3)
```

Will put the string shown into position 3 in the block, whose handle is block%.

```
s$=FNwimp_getfromblock(block%,3)
```

Will extract the string from position 3 in the block whose handle is block%.

Section 3 Functions

1. Misc

FNwimp_OStolength(coord,scale,inch%)

Converts OS units to mm or inches.
coord = value to convert (can be integer or floating point).
scale = scaling factor 0-100 (%).
If inch%=0 the value returned is in mm. If inch%=1 the value returned is in inches.

FNwimp_createblock(items%,length%)

Creates a block for storing strings in. Returns a handle to the block.
items% = maximum number of strings to store.
length% = maximum possible length of each string.

FNwimp_errorchoice(title$,error$,prefix%)

Reports an error using a standard error box.
It has both ‘OK’ and ‘CANCEL’ buttons.
title$ = title of error window.
error$ = error message.
IF prefix% = 0 then the title is title$. If prefix% = 1 then the title is prefixed by ‘Error from ’. If prefix% = 2 then the title is prefixed by ‘Message from ’.
Returns TRUE if ‘OK’ pressed. FALSE if ‘CANCEL’ pressed.

FNwimp_getfromblock(block%,pos%)

Returns a string stored in a block.
block% = handle of block.
pos% = position of string in block (ranging from 1 to maximum as passed to FNwimp_createblock).

FNwimp_getleafname(path$)

Returns a string containing the leafname from the pathname.
path$ = pathname.

FNwimp_getscreenres

Returns a 0 if the current screen mode is low resolution, and 1 if it is high resolution.

FNwimp_getscreensize(side%)

Returns the dimension of the screen required.
If side% = 0 returns width. If side% = 1 returns height.

FNwimp_initialise(name$,wimpmem%,iconmem%,ver%)

This function registers your application with the Task Manager and reserves some memory.
name$ = the name of your application eg. ‘Draw’.
wimpmem% = number of bytes to reserve for icon data or menu entries.
iconmem% = number of bytes to reserve for indirected text. eg. window titles or long menu entries.
ver% = minimum version of RISC OS that the application is allowed to run on multiplied by 100.

FNwimp_lengthtoOS(coord,scale,inch%)

Converts mm or inches to OS units.
coord = value to convert (can be integer or floating point).
scale = scaling factor 0-100 (%).
If inch%=0 the value coord supplied is in mm. If inch%=1 the value coord supplied is in inches.

FNwimp_libversion

Returns the version number of the library x 100. Eg. if the version of the library is 1.03 then 103 will be returned.

FNwimp_roundfloat(float)

Rounds the floating point number up or down and returns the integer.

FNwimp_screentowork(window%,coord%,side%)

Converts the x or y screen coordinate coord% to a work area x or y coordinate.
window% = handle of window whos work area coordinate is being converted into.
coord% = coordinate (x or y).
If side% = 0 then coord% is a x coordinate, and an x coordinate is returned.
If side% = 1 then coord% is a y coordinate, and a y coordinate is returned.

FNwimp_sysvariable(sysvar$)

Returns a string for the system variable sysvar$.
Note ‘<’ and ‘>’ are not required in sysvar$.

FNwimp_worktoscreen(window%,coord%,side%)

Converts the x or y work area coordinate coord% to a screen x or y coordinate.
window% = handle of window whos work area coordinate is being converted.
coord% = coordinate (x or y).
If side% = 0 then coord% is a x coordinate, and an x coordinate is returned.
If side% = 1 then coord% is a y coordinate, and a y coordinate is returned.

PROCwimp_bar(window%,icon%,length%,dir%)

Adjusts the length of a bar.
window% = handle of window containing the bar.
icon% = icon number of the bar.
length% = length of the bar.
If dir% = 0 then the bar moves horizontally keeping the height constant.
If dir% = 1 then the bar moves vertically keeping the width constant.

PROCwimp_error(title$,error$,button%,prefix%)

Reports an error using a standard error box.
title$ = title of error window.
error$ = error message.
IF button%=1 then will have an ‘OK’ button.
IF button%=2 then will have a ‘CANCEL’ button.
prefix% = prefix flag. If = 0 then the title is title$. If = 1 then the title is prefixed by ‘Error from ’. If = 2 then the title is prefixed by ‘Message from ’.

PROCwimp_increaseslot(bytes%)

Increases size of wimslot by bytes% bytes. If not enough available RAM then creates an error.

PROCwimp_pointer(pointer%,area%,pointer$)

Changes mouse pointer between the deafult (number 1) and the user defined pointer (number 2).
If pointer% = 0 default pointer is used. If pointer% = 1 user defined pointer is used.
If area% = 0 wimp sprite pool is used otherwise area% is a handle to a sprite area.
pointer$ = sprite name of pointer.

PROCwimp_putinblock(block%,string$,pos%)

Stores a string in a block which has been created using FNwimp_createblock.
block% = handle of block.
string$ = string to store.
pos% = position to store string in (ranging from 1 to maximum as passed to FNwimp_createblock).

PROCwimp_setcolour(red%,green%,blue)

Sets the current GCOL colour to the nearest possible for the current mode.
red% = amount of red in range 0-255.
green% = amount of green in range 0-255.
blue% = amount of blue in range 0-255.

PROCwimp_starttask(command$)

Sends command$ to the CLI. Omit ‘*’.

2. Polling

PROCwimp_poll

This function is the main loop of your application. When it has finished, your application has quitted. If something happens to your application eg. an icon has been clicked on, then the relevent function will be called from the loop.

PROCwimp_pollidle(seconds%)

If NULL=TRUE then PROCuser_null will be called every seconds% seconds instead of every time control is passed to the application and no event has occured.

PROCwimp_singlepoll

The same as PROCwimp_poll, execpt that it is called once and not in a loop. If something happens then the relevent action will still be taken before returning.
Useful for making loops multitask, eg: raytracing, printing, calculating, loading in data, etc.
Note: if calling in PROCuser_null, make sure NULL=FALSE before this call (can set to TRUE afterwards) otherwise recursion will occur.

PROCwimp_singlepollidle(seconds%)

The same as PROCwimp_pollidle, except that it is called once and not in a loop. If something happens then the relevent action will be taken before returning. If NULL=TRUE then PROCuser_null will be called after seconds% seconds.
Useful for incorporating delays into multitasking loops.
Note: if calling in PROCuser_null, make sure NULL=FALSE before this call (can set to TRUE afterwards) otherwise recursion will occur.

3. User

FNuser_help(window%,icon%)

Return a string to be used for interactive help for the window (and icon).
window% = handle of window (containing icon).
icon% = number of icon.

FNuser_keypress(window%,icon%,key)

If a key is pressed while one of your windows has the input focus, or a hotkey is pressed, then this function is called. If you don’t use the key then return a 0. If you do

then return a 1.
window% = handle of window with input focus.
icon% = number of icon with caret.
key = key code. For most keys it is an ASCII number.

Key	Alone	+Shift	+Ctrl	+Ctrl Shift
Escape	&1B	&1B	&1B	&1B
Print (F0)	&180	&190	&1A0	&1B0
F1-F9	&181-189	&191-199	&1A1-1A9	&1B1-1B9
Tab	&18A	&19A	&1AA	&1BA
Copy	&18B	&19B	&1AB	&1BB
left arrow	&18C	&19C	&1AC	&1BC
right arrow	&18D	&19D	&1AD	&1BD
down arrow	&18E	&19E	&1AE	&1BE
up arrow	&18F	&19F	&1AF	&1BF
Page down	&19E	&18E	&1BE	&1AE
Page up	&19F	&18F	&1BF	&1AF
F10-F12	&1CA-1CC	&1DA-1DC	&1EA-1EC	&1FA-1FC
Insert	&1CD	&1DD	&1ED	&1FD

FNuser_loaddata(path$,window%,icon%,filetype$)

You load data here. Return a 1 of data was loaded.
path$ = full pathname of source file.
window% = handle of window file was dragged on to.
icon% = number of icon file was dragged on to.
filetype$ = filetype of file to be loaded. Eg. “FFF”.

FNuser_menu(window%,icon%)

IF the specified window (and icon) has a menu which you want to appear when Menu is pressed over it, then this function should return the handle of the menu.
window% = handle of window.
icon% = number of icon.

FNuser_menuhelp(menu%,item%)

Return a string to be used for interactive help for the menu item.
menu% = handle of menu.
item% = number of item (starting from 1 at the top).

FNuser_pane(window%)

IF the window has a pane attached to it, then this function should return the window handle of the pane.
If the window doesn’t have a pane attached, then it should return a -1.
window% = handle of window.

FNuser_printing(copy%,page%,totpages%,pagepos%)

Called repeatedly during printing so application can keep user informed of current printing status, and give them the option to cancel printing.
copy% = number of current copy being printed.
page% = number of current page being printed.
totpages% = total number of pages being printed.
pagepos% = current page being printed (starts at 1 each time and goes up to totpages%).
Return a 1 to cancel printing or a 0 to continue.

FNuser_savedata(path$,window%)

When the user is required to save data, this function is called.
Return a 1 if some data was saved, otherwise return a 0.
path$ = full pathname of file to save data to.
window% = handle of save window icon was dragged from.

FNuser_savefiletype(window%)

You return the filetype for the save windows. eg. =‘FFF’. For windows that aren’t save windows, return an empty string eg. =‘’.
window% = handle of window%

FNuser_slider(window%,icon%)

In order to let DrWimp know that an icon is a slider, return the slider icon number.
window% = handle of window with slider in.
icon% = icon number of slider back icon.

FNuser_sliderback(window%,icon%)

In order to let DrWimp know that an icon is a slider, return the slider back icon number.
window% = handle of window with slider in.
icon% = icon number of slider.

PROCuser_closewindow(window%)

IF this function is called, then the window whose handle is window% has just been closed.

PROCuser_declarefonts

Any fonts being used in printing must be declared in this function using PROCwimp_declarefont or PROCwimp_declarefonth.

PROCuser_enteringwindow(window%)

This function is called when the pointer enters a window.
window% = handle of window.

PROCuser_leavingwindow(window%)

This function is called when the pointer leaves a window.
window% = handle of window.

PROCuser_menuopen(menu%,icon%)

Called just before menu is opened.
menu% = handle of menu jut about to open.
icon% = icon which pointer is over.

PROCuser_menuselection(menu%,item%)

This function is called when the user has chosen a menu item from one of your menus.
menu% = handle of menu.
item% = item number (top item is 1).

PROCuser_modechange

Called when the mode is changed.

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

IF an icon has been clicked on in one of your windows then this function is called.
window% = handle of window containing icon.
icon% = number of the icon clicked on.
button% = which mouse button was pressed. Eg. 4 for Select, 1 for Adjust.
workx%,worky% = work area coordinates of window window% that pointer was at when the mouse button was clicked.

PROCuser_null

This is called continuously. So if you are writing something like a clock, you would monitor the time here and change any windows as required.
IF you want to use this function then set NULL=TRUE.

PROCuser_openwindow(window%,x%,y%,stack%)

IF this function is called, then the window whose handle is window% has been opened with the top left of the window at x%,y% on the screen.
stack% = window handle to open behind, or -1 for top of window stack, or -2 for bottom.

PROCuser_print(minx%,miny%,maxx%,maxy%,page%)

Called to draw a page for printing, if user%=1 when PROCwimp_print was called.
minx%,miny% = coordinates of bottom left corner of clipping rectangle on page in paper coordinates.

maxx%,maxy% = coordinates of top right corner of clipping rectangle on page in paper coordinates.
page% = number of page to print.

PROCuser_printerchange

Called when the printer settings or the current printer has changed so you can update your page measurements, current printer name, etc.

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

When this function is called, the Wimp wants you to update the specified box on the screen. The box is in the work area of the window whose handle is window% or if printing then its in paper coordinates where the origin is at the bottom left of the paper.
printing% = TRUE if currently printing, FALSE otherwise.
page% = number of page currently being printed if printing%=TRUE.
minx%,miny% = bottom left co-ordinates of box in screen coordinates.
maxx%,maxy% = top right co-ordinates of box in screen coordinates.

PROCuser_saveicon(window%,RETURN drag%,RETURN write%, RETURN ok%)

This function allows the three save window icons (the one to drag, the writeable icon for the filename/pathname, the OK button) to have their icon numbers set, if you want to override the defaults.
Defaults:
  drag% - 0   write% - 1   ok% - 2
window% = handle of save window.

PROCuser_slidervalue(window%,slider%,pcent%)

When a slider is being dragged or has just finished being dragged, the percentage of the slider is passed to this function.
window% = handle of window with slider in.
icon% = icon number of slider.
pcent% = percentage of slider.

4. Windows

FNwimp_createwindow(vminx%,vminy%,vmaxx%,vmaxy%,wminx%, wminy%,wmaxx%,wmaxy%,flags%, workcol%,button%,title$,maxind%,sarea%)

Creates a window, returning the handle to it.
vminx%,vminy%,vmaxx%,vmaxy% = bounding box of window on the screen in OS

units.
wminx%,wminy%,wmaxx%,wmaxy% = bounding box of work area in work area coordinates.
flags% = number representing window flags.
workcol% = work area colour in desktop colours, in range 0-15.
button% = work area button type. 0 for ignoring clicks, 1 for allowing them.
title$ = title of window.
maxind% = maximum size of title if indirected.
sarea% = handle of sprite area, or 0 to use wimp sprite area.

flags%:	bit	meaning if set
	0	window has title bar
	1	window has close icon
	2	window has back icon
	3	window has horizontal scroll bar
	4	window has vertical scroll bar
	5	window has adjust size icon
	6	window has toggle size icon
	7	window can auto-redraw
	8	window is a pane
	9	window has indirected title

FNwimp_getwindowsize(window%,side%)

Returns the dimension required.
If side% = 0 returns width. If side% = 1 returns height.

FNwimp_getwindowtitle(window%)

Returns a string containing the window title.
window% = handle of window.

FNwimp_getwindowworksize(window%,side%)

Returns the size on OS units of a window work area.
window% = handle of window.
If side%=0 the width of the work area is returned. If side%=1 the height of the work area is returned.

FNwimp_iswindowopen(window%)

Returns a 1 if the window is open otherwise returns a 0.
window% = handle of window.

FNwimp_loadwindow(path$,window$,sprite%)

Loads in a window from a templates file and returns a handle for the window.

path$ = full pathname to templates file.
window$ = name of window in templates file.
sprite% = sprite flag. If = 0 then use sprites from wimp area (RMA). Otherwise sprite% is a handle to a sprite area.

PROCwimp_banner(window%,delay%)

Opens window in the centre of the screen for specified delay before closing it.
window% = handle of window to open.
delay% = number of seconds to keep window on screen.

PROCwimp_closewindow(window%)

Closes a window (removes it from the screen).
window% = handle of window to close.

PROCwimp_deletewindow(window%)

Deletes a window, closing it if it is open. All the memory apart from the indirected memory is reclaimed and the window handle becomes invalid.
window% = handle of window to delete.

PROCwimp_openwindow(window%,centre%,stack%)

Opens a window on the screen.
window% = handle of window to open.
If centre% = 0 opens window where it was last left on the screen, or if it hasn’t been opened before, then where it is positioned in the template file.
If centre% = 1 opens the window centred on the screen (mode independent).
stack% = window handle to open behind, or -1 for top of window stack, -2 for bottom, or -3 for current stack position.

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

Opens a window on the screen so the top left of the window is at co-ordinates x%,y%.
window% = handle of window to open.
stack% = window handle to open behind, or -1 for top of window stack, -2 for bottom or -3 for current stack position.

PROCwimp_putwindowtitle(window%,title$)

Changes the window title to title$
window% = handle of window.

PROCwimp_redrawwindow(whan%)

Causes the window whose handle is whan% to be redrawn.

PROCwimp_resizewindow(whan%,width%,height%)

Resizes the window to the specified width and height which are in OS co-ordinates. The work area and the visible area are both set to the values.

5. Messages

FNwimp_createmessagemenu(tag$,title$,size%)

Creates a menu automatically from a Messages file, with the same result as FNwimp_createmenu.
tag$ = tag for menu. Eg: if tag$=“MMenu” then the tag “MMenuT” will specify the title, “MMenu1” the first item, “MMenu2” the second etc.
If title$=“” then the title defined in the message file will be used, otherwise title$ will override whatever is defined in the messages file.
IF size%>number of items then the menu is dynamic, ie. the items can be increased up to size%.

FNwimp_messlook0(token$)

Returns the string in the messages file for the token token$.

FNwimp_messlook1(token$,a$)

Returns the string in the messages file for the token token$. Any ‘%0’s in the string are replaced with a$ before returning.

FNwimp_messlook2(token$,a$,b$)

Returns the string in the messages file for the token token$. Any ‘%0’s and ‘%1’s are replaced with a$ and b$ respectively before returning.

PROCwimp_initmessages(path$)

Reserves blocks of memory and sets up Messages file for use.
path$ = full pathname of messages file to use.

6. Icons

FNwimp_createicon(window%,wminx%,wminy%,wmaxx%,wmaxy%, flags%,esg%,button%,fcol%,bcol%,fhan%, text$,sprite$,sarea%,maxind%,valid$)

Creates an icon and returns the handle to it (icon number).
window% = handle of window to create icon in.
wminx%,wminy%,wmaxx%,wmaxy% = bounding box of icon in work area coordinates.
flags% = number representing flags for icon.
esg% = esg number of icon. 0 for icons which arn't radio buttons.

button% = button type of icon.
fcol%,bcol% = foreground and background colours of icons (if not using outline font) in dekstop colours, so both in the range 0-15.
fhan% = handle of outline font. 0 if not using a font.
text$ = text for icon.
sprite$ = sprite name for icon.
sarea% = handle of sprite area, or 0 to use wimp sprite area.
maxind% = if icon is indirected then maximum size.
valid$ = icon validation string.

flags%:	bit	meaning if set
	0	icon has text
	1	icon has sprite
	2	text/sprite is horizontally centred
	3	text/sprite is vertically centred
	4	icon is filled
	5	text/sprite is right justified
	6	icon has border
	7	icon uses an outline font
	8	icon is indirected

button%:	button%	type
	0	ignore
	1	click
	2	click autorepeat
	3	click/drag
	4	writeable
	5	radio

FNwimp_geticonsize(window%,icon%,side%)

Returns the dimension required.
If side% = 0 returns width. If side% = 1 returns height.

FNwimp_geticonstate(window%,icon%)

Returns a 1 if the icon is selected and a 0 if it is unselected. Useful for reading the state of radio and option icons.
window% = handle of window containing icon.
icon% = number of icon.

FNwimp_geticontext(window%,icon%)

Returns a string containing the text from the icon.
window% = handle of window containing icon.

icon% = icon number.

FNwimp_getsliderpcent(window%,icon%)

Returns the percentage of the slider. If the icon is not a slider then 0 is returned.
The number returned is a floating point number in the range 0-100.
window% = handle of window with slider in.
icon% = icon number of slider.

FNwimp_iconbar(sprite$,text$,pos%)

Places an icon on the iconbar.
sprite$ = name of sprite to put on iconbar.
text$ = text to put underneath the icon eg. like the floppy drive icon. If text$ = “” then no text will be used, and the icon will be positioned correctly.
pos% = controls the position of the icon. If pos% = 1 then the icon will appear on the right. If pos% = 0 then it will appear on the left.
Returns window handle.

PROCwimp_colouricontext(window%,icon%,colour%)

Sets colour of text in icon to colour%.
window% = handle of window containing icon.
icon% = number of icon.
colour% = colour in range 0-15.

PROCwimp_deleteicon(window%,icon%,redraw%)

Removes an icon from a window. The icon will not dissappear unless the window is redrawn.
window% = handle of window contaning icon.
icon% = icon number of icon to delete.
If redraw% is 1 then the window is redrawn. If redraw% is 0 then it isn't and the icon won't dissappear.

PROCwimp_iconbarsprite(sprite$)

Changes the sprite used for the iconbar icon to sprite$.

PROCwimp_iconbit(window%,icon%,bit%,state%)

Ensures a specific bit of an icon is set to the specified state.
window% = handle of window contaning icon.
icon% = number of icon.
bit% = number of bit to change.
state% = state to set bit to. Can be 0 or 1.

PROCwimp_icondisable(window%,icon%)

Greys out icon so it cannot be selected.
window% = handle of window containing icon.
icon% = number of icon.

PROCwimp_iconenable(window%,icon%)

Un-greys out icon so it can be selected.
window% = handle of window containing icon.
icon% = number of icon.

PROCwimp_iconselect(window%,icon%,state%)

Selects (inverts) and un-selects icon.
window% = handle of window containing icon.
icon% = number of icon.
If state% = 0 icon is un-selected. If state% = 1 icon is selected.

PROCwimp_losecaret

Removes the caret from the icon it is in.

PROCwimp_putcaret(window%,icon%)

Puts the caret in the icon.
window% = handle of window containing icon.
icon% = number of icon.

PROCwimp_puticonbartext(text$)

If the iconbar icon has text underneath it then it is replaced by text$.

PROCwimp_puticontext(window%,icon%,text$)

If the icon is indirected then the text in the icon is replaced with text$. If the icon is indirected then an error is caused.
window% = handle of window containing icon.
icon% = number of icon.

PROCwimp_putsliderpcent(window%,icon%,pcent)

Sets the percentage of the slider. If the icon is not a slider then this is ignored.
window% = handle of window with slider in.
icon% = icon number of slider.
pcent = percentage to set.
Can be integer or floating point number, but must have the range 0-100.

7. Menus

FNwimp_createmenu(menu$,size%)

Creates a menu structure from the string menu$. The menu handle is returned.
For more information on menu$ see the manual.
IF size%>number of items then the menu is dynamic, ie. the items can be increased up to size%.

FNwimp_createmenuarray(array$(),size%)

Creates a menu from the array supplied. Each item of the menu is in a seperate element of the array. eg:
array$(3)='Info'. The first element is the menu title, and the last must be 'END'.
array$() = array to get items from.
size% = maximum number of elements to allocate room for (doesn't have to be the current number).
Returns a handle to the menu.

FNwimp_createmessagemenu(tag$,title$,size%)

Creates a menu automatically from a Messages file, with the same result as FNwimp_createmenu.
tag$ = tag for menu. Eg: if tag$="MMenu" then the tag "MMenuT" will specify the title, "MMenu1" the first item, "MMenu2" the second etc.
If title$="" then the title defined in the message file will be used, otherwise title$ will override whatever is defined in the messages file.
IF size%>number of items then the menu is dynamic, ie. the items can be increased up to size%.

FNwimp_getmenutext(menu%,item%)

Returns a string containing the text of the menu item.
menu% = handle of menu.
item% = number of item (top item is 1).

FNwimp_getmenutitle(menu%)

Returns a string containing the title of the menu.
menu% = handle of menu.

FNwimp_menusize(menu%)

Returns the number of entries in the menu.
menu% = handle of menu.

PROCwimp_attachsubmenu(menu%,item%,submenu%)

Attaches a submenu to a menu item.
menu% = handle of menu.
item% = item number (top item is 1).
submenu% = handle of submenu or window handle.

PROCwimp_menuclose

Closes the currently active menu.

PROCwimp_menudisable(menu%,item%)

Greys out the menu item so it is un-selectable.
menu% = handle of menu.
item% = item number (top item is 1).

PROCwimp_menudottedline(menu%,item%)

Adds a dotted line to the menu below the item.
menu% = handle of menu.
item% = number of item (top item is 1).

PROCwimp_menuenable(menu%,item%)

Un-greys out the menu item so it is selectable.
menu% = handle of menu.
item% = item number (top item is 1).

PROCwimp_menupopup(menu%,bar%,x%,y%)

Brings up the menu whose handle is menu% at the co-ordinates x%,y%. If bar%=1 then the menu will be positioned as for an iconbar menu, otherwise use 0.

PROCwimp_menutick(menu%,item%)

If the item doesn’t have a tick next to it then this function places one. If the item does have a tick then it is removed.
menu% = handle of menu.
item% = item number (top item is 1).

PROCwimp_menuwrite(menu%,item%,length%)

Makes the menu item writeable.
menu% = handle of menu.
item% = number of item (top item is 1).
length% = maximum length of text allowed to be entered.

PROCwimp_putmenuitem(menu%,item%,item$)

If the menu is dynamic then item$ will be put into menu item item%. Any items below will be shuffled down. If item% is bigger than the current number of items+1, then it will be added to the bottom.
menu% = handle of menu.

PROCwimp_putmenutext(menu%,item%,text$)

Replaces menu items text with text$.
menu% = handle of menu.
item% = number of item (Top item is 1).

PROCwimp_putmenutitle(menu%,title$)

Changes the title of the menu.
If the title>11 characters then it is truncated.
menu% = handle of menu.
title$ = new title.

PROCwimp_recreatemenu(menu%,menu$)

Rebuilds the menu using the string menu$. More items can be included than the first time as long as you don’t go over the pre-defined limit.
menu% = handle of menu to rebuild.

PROCwimp_recreatemenuarray(menu%,array$())

Rebuilds the menu using the items in the array. The first array item (array$(0)) is the menu title, and the last has to be ‘END’.
Things like ticks and dotted lines are reset.
menu% = handle of menu to rebuild
array$() = array to get items from.

PROCwimp_removemenuitem(menu%,item%)

Removes the item from the menu. Any items below are shuffled up. If there is only one item on the menu, then it cannot be removed.
menu% = handle of menu. item% = number of item to remove.

8. Sprites

FNwimp_getspritesize(sprite$,sprite%,side%)

Returns the width or height in OS units of a sprite.
sprite$ = name of sprite.
sprite% = handle of sprite area containing sprite.
If side%=0 then returns width of sprite.

If side%=1 then returns height of sprite.

FNwimp_loadsprites(path$,ptr%)

Loads a spritefile into a block of memory at ptr%.
Returns a new value of ptr% to load the next in.
path$ = full pathname to sprite file.

FNwimp_measurefile(path$)

Returns the size in bytes needed to store the spritefile.
Always use this as opposed to any other form of measurement.
path$ = full pathname of spritefile.

**PROCwimp_rendersprite(sprite$,sprite%,bx%,by%,minx%,miny%,
maxx%,maxy%)**

Renders a sprite on the screen at the specified coordinates, using the clipping rectangle.
sprite$ = name of sprite to plot.
sprite% = handle of sprite area containing sprite.
bx%,by% = screen coordinates to put bottom left corner of sprite at.
minx%,miny% = coordinates of bottom left corner of clipping rectangle in screen coordinates.
maxx%,maxy% = coordinates of top right corner of clipping rectangle in screen coordinates.

**PROCwimp_renderwindowsprite(window%,sprite$,sprite%,bx%,by%,
minx%,miny%,maxx%,maxy%)**

Renders a sprite in a window. The window must have its auto-redraw flag unset.
window% = handle of window to render sprite in.
sprite$ = name of sprite to render.
sprite% = handle of sprite area containing sprite.
bx%,by% = work area coordinates of where to put bottom left of sprite.
minx%,miny% = coordinates of bottom left corner of clipping rectangle in screen coordinates.
maxx%,maxy% = coordinates of top right corner of clipping rectangle in screen coordinates.

9. Drawfiles

FNwimp_getdfilesize(dfile%,side%)

Returns the dimension required.
If side% = 0 returns width. If side% = 1 returns height.

FNwimp_loaddfile(path$,ptr%)

Loads a drawfile into a block of memory at ptr%.
Returns a new value of ptr% to load the next in.
path$ = full pathname of drawfile.

FNwimp_measurefile(path$)

Returns the size in bytes needed to store the drawfile.
Always use this as opposed to any other form of measurement.
path$ = full pathname of drawfile.

PROCwimp_initdfiles

Initialises various blocks of memory ready to use with drawfiles.

PROCwimp_render(dfile%,bx%,by%,xl%,yl%,xh%,yh%)

Renders a drawfile at bx%,by% using the clipping rectangle xl%,yl%,xh%,yh%. All coordinates are in OS units.
dfile% = handle of drawfile to render.
bx%,by% = coordinates of where to put bottom left corner of drawfile.
xl%,yl% = coordinates of bottom left corner of clipping rectangle in screen coordinates.
xh%,yh% = coordinates of top right corner of clipping rectangle in screen coordinates.

PROCwimp_renderwindow(w%,dfile%,tx%,ty%,xl%,yl%,xh%,yh%)

Renders a drawfile in a window. The window must have its auto-redraw flag unset.
w% = handle of window.
dfile% = handle of drawfile to render.
tx%,ty% = work area coordinates of where to put bottom left corner of drawfile.
xl%,yl% = coordinates of bottom left corner of clipping rectangle in screen coordinates.
xh%,yh% = coordinates of top right corner of clipping rectangle in screen coordinates.

PROCwimp_savedfile(path$,dfile%)

Saves a drawfile into a file.
dfile% = handle of drawfile to save.
path$ = full pathname to save to.

10. Text

FNwimp_fontcolour(fr%,fg%,fb%)

Returns control codes in a string to change the current font colour. Useful for using in the middle of a string of text being plotted using a font.
fr%,fg%,fb% = red, green and blue components of the foreground colour respectively, in the range 0-255.

FNwimp_fontunderline(on%)

Returns control codes in a string to turn underlining on or off. Useful for using in the middle of a string of text being plotted using a font.
If on%=0 turns underlining off. If on%=1 turns underlining on.

FNwimp_getfont(font$,size%)

Obtains a font handle for a particular font at a particular size.
font$ = name of font, period seperated. eg: “Trinity.Medium”.
size% = point size of font.
Returns 0 if the font cannot be found.

FNwimp_gettextsize(text$,font$,size%,side%)

Returns the size of some text in a particular font in OS units.
text$ = string to measure.
font$ = name of font, period seperated, eg: “Trinity.Medium”.
size% = point size of font.
If side%=0 then the width is returned. If side%=1 then the height is returned.

FNwimp_gettextsizeh(texts%,font%,side%)

Returns the size of some text in a particular font in OS units, using a font handle.
text$ = string to measure.
font% = handle of font.
If side%=0 then the width is returned. If side%=1 then the height is returned.

FNwinp_fontchangeh(font%)

Returns control codes in a string to change the current font. Useful for using in the middle of a string of text being plotted using a font.
font% = handle of font to change to.

PROCwimp_deskplottext(t$,c%,x%,y%,fr%,fg%,fb%,br%,bg%,bb%)

Plots text using the current desktop font (always the System Font on pre-RISC OS 3.50).
t$ = string to plot.
If c%=1 then text is horizontally centred around x%.
If c%=0 then left side of text is placed at x%.
x%,y% = coordinates to plot the text at.
fr%,fg%,fb% = foreground colour red, green and blue amounts in range 0-255.
br%,bg%,bb% = background colour red, green and blue amounts in range 0-255.

PROCwimp_deskplotwindowtext(window%,t$,c%,x%,y%,fr%,fg%,fb%,
br%,bg%,bb%,minx%,miny%,maxx%,maxy%)

Plots text using the current desktop font (always the System Font on pre-RISC OS 3.50) in a window.
window% = handle of window to plot in.
t$ = string to plot.
If c%=1 then text is horizontally centred around x%.
If c%=0 then left side of text is placed at x%.
x%,y% = coordinates to plot the text at.
fr%,fg%,fb% = foreground colour red, green and blue amounts in range 0-255.
br%,bg%,bb% = background colour red, green and blue amounts in range 0-255.

PROCwimp_losefont(font%)

Forgets about a font. Should be called when you have finished with the font, eg. when the application is quitting.
font% = handle of font to loose.

PROCwimp_plottext(t$,f$,s%,x%,y%,fr%,fg%,fb%,br%,bg%,bb%)

Plots text using a font.
t$ = string to plot.
f$ = name of font period spaced eg: “Trinity.Medium”
s% = point size of font.
x%,y% = screen coordinates to plot the text at.
fr%,fg%,fb% = foreground colour red, green and blue amounts in range 0-255.
br%,bg%,bb% = background colour red, green and blue amounts in range 0-255.

PROCwimp_plottexth(text$,font%,x%,y%,fr%,fg%,fb%,br%,bg%,bb%)

Plots text using a font (with font handle).
t$ = string to plot.
font% = handle of font.
s% = point size of font.
x%,y% = coordinates to plot the text at.
fr%,fg%,fb% = foreground colour red, green and blue amounts in range 0-255.
br%,bg%,bb% = background colour red, green and blue amounts in range 0-255.

PROCwimp_plotwindowtext(window%,t$,f$,s%,x%,y%,fr%,fg%,fb%,br%,
bg%,bb%,minx%,miny%,maxx%,maxy%)

Plots some text in a window.
window% = handle of window to plot in.
t$ = string to plot.
f$ = name of font to use, period seperated, eg: “Trinity.Medium”.

s% = point size of font.
x%,y% = work area coordinates to plot text at.
fr%,fg%,fb% = foreground colour red, green and blue components respectively, in the range 0-255.
br%,bg%,bb% = background colour red, green and blue components in the range 0-255
minx%,miny% = coordinates of bottom left corner of clipping rectangle.
maxx%,maxy% = coordinates of top right corner of clipping rectangle.

PROCwimp_plotwindowtexth(window%,t$,font%,x%,y%,fr%,fg%,fb%, br%,bg%,bb%,minx%,miny%,maxx%,maxy%)

Plots some text in a window using font handle.
window% = handle of window to plot in.
t$ = string to plot.
font% = handle of font to use.
s% = point size of font.
x%,y% = work area coordinates to plot text at.
fr%,fg%,fb% = foreground colour red, green and blue components respectively, in the range 0-255.
br%,bg%,bb% = background colour red, green and blue components respectively, in the range 0-255
minx%,miny% = coordinates of bottom left corner of clipping rectangle.
maxx%,maxy% = coordinates of top right corner of clipping rectangle.

11. Printing

FNwimp_getpapersize(side%,type%)

Returns various information about the current paper size.
IF side%=0 then a horizonal measurement is returned. If side%=1 then a vertical measurement is returned. Which measurement is determined by type%.
If type%=0 then the width or height is returned.
If type%=1 then the left or bottom margin is returned.
If type%=2 then the printable width or printable height is returned.
If type%=3 then the right or top margin is returned.

FNwimp_getpdrivername

If a printer driver is loaded, then this function returns a string containing the name of the printer driver. Check to make sure one is loaded first with FNwimp_pdriverpresent.

FNwimp_papertoscreen(window%,coord%,side%)

Converts a paper x or y coordinate to a screen x or y coordinate.
window% = handle of window whose work area to use.
coord% = coordinate (x or y).

If side%=0 then coord% is a x coordinate, and a x coordinate is returned.
If side%=1 then coord% is a y coordinate, and a y coordinate is returned.

FNwimp_papertowork(coord%,side%)

Converts a paper x or y coordinate to a work area x or y coordinate.
coord% = coordinate (x or y).
If side%=0 then coord% is a x coordinate, and a x coordinate is returned.
If side%=1 then coord% is a y coordinate, and a y coordinate is returned.

FNwimp_pdriverpresent

Checks to see if a printer driver is loaded.
Returns a 0 if one isn’t, returns a 1 if one is.

FNwimp_screentopaper(window%,coord%,side%)

Converts a screen x or y coordinate to a paper x or y coordinate.
window% = handle of window whose work area to use.
coord% = coordinate (x or y).
If side%=0 then coord% is a x coordinate, and a x coordinate is returned.
If side%=1 then coord% is a y coordinate, and a y coordinate is returned.

FNwimp_worktopaper(coord%,side%)

Converts a work area x or y coordinate to a paper x or y coordinate.
coord% = coordinate (x or y).
If side%=0 then coord% is a x coordinate, and a x coordinate is returned.
If side%=1 then coord% is a y coordinate, and a y coordinate is returned.

PROCwimp_declaredfilefonts(drawfile%)

Declares the fonts used in a drawfile (especially) for postscript printing.
drawfile% = handle of drawfile to be printed.

PROCwimp_declarefont(font$)

Declares a font for printing.
font$ = name of font to declare, period seperated. eg: “Trinity.Medium”.

PROCwimp_declarefonth(font%)

Declares a font for printing using font handle.
font% = handle of font to declare.

PROCwimp_print(user%,window%,fpage%,lpage%,perpage%,copies%)

Initiates printing of a document.
IF user%=0 PROCuser_redraw is called to draw the pages, with printing% set to TRUE

and the clipping rectangle set to the page coordinates. If user%=1 PROCuser_print is called to draw the pages.
window% = handle of window to redraw if user%=0.
fpage% = page number of first page to print.
lpage% = page number of last page to print.
perpage% = number of A4 pages to fit onto a physical A4 page. Can be 1, 2 or 4.
copies% = number of copies of the document to print.

Index