

ClearWin+TM

Fortran Edition



Table of Contents

1.	Introduction	1
	Windows applications.....	1
	The ClearWin+ library	1
	How to use this guide.....	2
2.	Getting started	5
	Introduction.....	5
	Using FTN77 and FTN95	6
3.	ClearWin+ tutorial.....	7
4.	Format windows	19
	Introduction.....	19
	Call-back functions.....	21
	Window formats.....	22
	Closing a window.....	25
	Updating windows	26
	Advanced use of format windows.....	27
	Format codes grouped by function	30
	Alphabetical index of format codes	31
5.	Input and output	37
	Interactive I/O	37
	Data output	45
6.	Button controls.....	51
7.	Controls other than buttons.....	61
8.	Bitmaps, Cursors, Icons.....	75
9.	Menus and accelerator keys.....	81

10.	Layout and positioning	89
11.	Displaying text	97
	Displaying text.....	97
	Edit boxes.....	103
	File selection and filter.....	105
12.	Edit box (%eb).....	107
13.	Help.....	119
14.	Window attributes.....	121
15.	Graphics	135
	Introduction.....	135
	Graphics format %gr[<i>options</i>]	136
	Basic graphics primitives.....	138
	Drawing text	138
	Re-sizing a graphics surface	141
	The %gr call-back function	142
	Graphics selections	143
	Drawing device independent bitmaps.....	145
	Direct manipulation of the graphics surface.....	148
	Off-screen graphics.....	150
	Graphics icons.....	152
	Windows graphic modes.....	154
	Owner draw box format - %dw[<i>options</i>]	157
	Updating the screen.....	159
16.	OpenGL	161
	Introduction.....	161
	OpenGL - %og[<i>options</i>].....	162
	The conversion of a Red Book example.....	163
	Argument types	165
	Call-backs and initialisation.....	166
	Animation and multiple windows.....	171
	Using a printer with OpenGL.....	179
	References.....	179
	Online information.....	179

Table of Contents

17.	SIMPLEPLOT.....	181
	Introduction.....	181
	A simple example.....	182
	%pl options.....	183
	Direct calls to SIMPLEPLOT	185
18.	Using the printer	189
	Introduction.....	189
	Selecting a printer	190
	Customising the printer dialog box.....	192
	Printer graphics	194
	Using a metafile to print multiple copies.....	195
19.	Hypertext windows	199
	Hypertext %N.Mht.....	199
	Conditional Hypertext.....	203
	Linking to the web	204
20.	Standard call-back functions	205
	A simple text editor.....	212
21.	Format code reference	215
22.	Compiling and linking	293
	Compiler options and directives for Windows	293
	SLINK commands	295
	Using the Salford Resource Compiler	296
	FTN95.....	297
23.	ClearWin windows	299
	Introduction.....	299
	Simple programs	300
	Explicit <i>ClearWin</i> window creation	301
	Text output to <i>ClearWin</i> windows	301
	Getting text from a <i>ClearWin</i> window	303
	Obsolete <i>ClearWin</i> window functions.....	303
24.	Using the Windows API.....	305
	Calling Windows API routines from Fortran	305
	The Fortran STDCALL statement.....	306

The Fortran STOP and PAUSE statements.....	309
Using Windows API structures from Fortran.....	310
25. Programming tips	311
Tricks with boxes.....	311
Debugging with an auxiliary terminal	312
Recording mouse and keyboard events	313
Breaking into the debugger	314
26. Library overview	315
ClearWin window functions.....	315
Clipboard functions	316
File mapping functions	317
Format window functions	317
General functions.....	318
Graphics functions (bitmaps,cursors,icons).....	318
Graphics functions (device context)	319
Graphics functions (device independent bitmap)	319
Graphics functions (font,text).....	320
Graphics functions (lines,fill)	321
Graphics functions (metafiles).....	322
Graphics functions (palette).....	322
Graphics functions (printer,files).....	323
Graphics functions (screen,regions)	324
Hypertext functions.....	324
Information functions	325
Mouse functions.....	325
Movie functions	325
Standard dialog functions	326
Sound functions	326
Useful API functions.....	327
27. Library reference.....	329
28. Glossary	421
Appendix A. Functions ported from DBOS.....	429
Graphics.....	429
Graphics plotter/screen	431
Graphics printer	432
Mouse.....	432
Table of alternative routines.....	433

IMPORTANT NOTICE

Salford Software Ltd gives no warranty that all errors have been eliminated from this manual or from the software or programs to which it relates and neither the Company nor any of its employees, contractors or agents nor the authors of this manual give any warranty or representation as to the fitness of such software or any such program for any particular purpose or use or shall be liable for direct, indirect or consequential losses, damages, costs, expenses, claims or fee of any nature or kind resulting from any deficiency defect or error in this manual or such software or programs.

Further, the user of such software and this manual is expected to satisfy himself/herself that he/she is familiar with and has mastered each step described in this manual before the user progresses further.

The information in this document is subject to change without notice.

17 February, 2000

© Salford Software Ltd 2000

All copyright and rights of reproduction are reserved. No part of this document may be reproduced or used in any form or by any means including photocopying, recording, taping or in any storage or retrieval system, nor by graphic, mechanical or electronic means without the prior written consent of the Salford Software Ltd.

FTN77 is a registered trademark of Salford Software Ltd.

FTN95, DBOS, Salford C++, and ClearWin+ are trademarks of Salford Software Ltd.

MS-DOS, Visual Basic, Windows, Windows 95 and Windows NT are all trademarks of Microsoft Corporation.

PREFACE

This user's guide discusses the binding from Salford Fortran compilers to the Microsoft Windows Applications Programming Interface (API). It also describes the Salford ClearWin+ library. ClearWin+ offers an easy-to-use interface to the Windows API.

ClearWin+ is used together with a Salford Win32 compiler in order to develop Windows (Win32) applications that will run under Windows 95 and Windows NT. This guide describes how to write Win32 applications.

Details of how to use ClearWin+ to develop Win16 applications will be found in the *ClearWin+ User's Supplement*. An electronic copy of this supplement can be found on the Salford Tools CD. The supplement also includes details of functions that are considered to be obsolete together with details of functions that can be ported from old DBOS programs but which do not add any extra functionality those that are described in this guide. Programmers who are familiar with the older functions and who prefer to use them should refer to the supplement.

The guide does not try to describe or to document the Windows API, which is a subject too extensive to be covered here. Full documentation of the Windows API is provided with the Microsoft Windows Software Development Kit (SDK).

The Salford Tools CD contains example programs that illustrate some of the more common operations that developers will need in order to produce Windows applications. It also provides an HTML version of this manual that can be used to extract the sample code that appears throughout the text.

This guide is for version 6.0 of ClearWin+. It also includes details of the new branch-view format (%bv). Enhancements to ClearWin+ that have been added after the preparation of this guide are described in a README and .ENH enhancement files on the CD and also in the BROWSE example program.

The chapter headings in this guide are as follows:

	<i>page</i>
1. Introduction	1
2. Getting started	5
3. ClearWin+ tutorial.....	7
4. Format windows	19
5. Input and output	37
6. Button controls	51
7. Controls other than buttons	61
8. Bitmaps, Cursors, Icons	75
9. Menus and accelerator keys.....	81
10. Layout and positioning.....	89
11. Displaying text.....	97
12. Edit box (%eb).....	107
13. Help	119
14. Window attributes.....	121
15. Graphics.....	135
16. OpenGL	161
17. SIMPLEPLOT	181
18. Using the printer.....	189
19. Hypertext windows	199
20. Standard call-back functions.....	205
21. Format code reference	215
22. Compiling and linking.....	293
23. ClearWin windows	299
24. Using the Windows API	305
25. Programming tips	311
26. Library overview	315
27. Library reference	329
28. Glossary.....	421
Appendix A. Functions ported from DBOS.....	429

1.

Introduction

Windows applications

ClearWin+ is a library of functions and subroutines that enables developers to produce Windows applications very quickly. It can be used together with a Salford Win32 compiler (**FTN77**, **FTN95** or **SCC**) in order to produce Windows (Win32) applications for Windows 95 and Windows NT.

Many developers will see the need to take advantage of the graphics user interface (GUI) that is now available in Windows environments in order to produce applications that are simple to use and professional in their appearance. At the same time there will be very few developers who have the time and interest to work through the long learning curve that is required in order to get to grips with the Windows API - a Microsoft library that can be used to write Windows applications. Salford's solution to this dilemma is ClearWin+. ClearWin+ is designed to be used in a way that closely emulates traditional programming techniques. The complexities of the Windows API are avoided by using the ClearWin+ interface with the result that Windows applications can be developed in a very small fraction of the time that otherwise would be needed. The resulting programs are much shorter than equivalent Windows programs produced in other ways.

The ClearWin+ library

The ClearWin+ interface may be used at three levels:

At the first and simplest level, DOS programs that interact with the user via standard Fortran input and output statements (for example READ, WRITE and PRINT) may be recompiled to produce simple Windows applications with little or no modification to

the source code. ClearWin+ will automatically create a window (called a *ClearWin* window) complete with scroll bars and manage it for the developer.

At an intermediate level, ClearWin+ can be used to provide a professional graphics interface, without incurring the burden of directly using the full Windows API. In particular, the `WINIO@` function can be used to produce one or more fully featured GUI windows (called *format* windows) both quickly and easily. This function dispenses with the need to provide detailed resource scripts for menus, dialog boxes, etc.. It also largely eliminates the requirement for the user to provide complex callback functions in order to control the interface. The effect is that much of the complexity of using the full API becomes transparent to the programmer. At this level, Windows API functions are rarely needed.

Finally at the lowest level, it is possible for the developer to use ClearWin+ to write programs that use Windows API functions. However, even those who are familiar with the Windows API will probably find it quicker and easier to work at the intermediate level using the ClearWin+ library.

ClearWin+ is supplied with a full Windows debugger. Detailed information on how to use the debugger is given the user guides for each of the Salford compilers.

A convention has been adopted in this guide of using mixed case names for the routines present in the Windows API (for example, `CreateWindow`, `GetDC`, `GetTextMetrics`). Routines in the ClearWin+ library do not use mixed case names.

How to use this guide

All integer constants and variables in this guide that are declared as `INTEGER` are assumed to be 32 bits long (i.e. `INTEGER(KIND=3)` or `INTEGER*4`). Real values are usually declared as `DOUBLE PRECISION` and assumed to be 64 bits long (i.e. `REAL(KIND=2)` or `REAL*8`).

This guide distinguishes between three types of window.

- 1) An *API* window is produced using only Windows API functions. The term does not imply any added functionality from the ClearWin+ library.
- 2) A *ClearWin* window is a window that has the added ability to make direct use of standard Fortran I/O (`READ`, `WRITE` and `PRINT`). The *ClearWin* window was the central feature of ClearWin versions 3.0 and 3.1 and is still of interest in the current version of ClearWin+. A *ClearWin* window example is used in the “getting started” section of Chapter 2.
- 3) A *format* window is a window that has been created using the ClearWin+ `WINIO@` function. This function is the distinguishing feature of ClearWin+ (it is, to all intents and purposes, the “+” in ClearWin+). Like a *ClearWin* window, a *format*

window also has added functionality which provides all of the standard GUI facilities in an easily programmed form - enabling the programmer to bypass the complexities of the Windows API. Chapter 3 provides a tutorial to get you started with *format* windows whilst chapters 4 to 21 provide detailed information.

The guide includes a Glossary. Words that have a special meaning in the context of Windows and ClearWin+ are often presented in italic font and included in the Glossary.

Please note that Windows 95 has been used to create the illustrations in this user guide. Other versions of Windows may produce a slightly different appearance.

2.

Getting started

Introduction

The Salford Tools CD includes a number of programs that illustrate the use of ClearWin+. Particular attention should be given to the demonstration programs that illustrate the use of the ClearWin+ function `winio@` and to the ‘Browse’ example.

The following instructions explain how to compile, link and execute a very simple Windows application without using `winio@`.

Consider a program to calculate the squares of a sequence of numbers and print them out.

In Fortran this takes the form¹:

```
WINAPP
INTEGER idx
DO idx=1,10
    PRINT *,idx,' Squared is ', idx*idx
ENDDO
END
```

Using your editor, type in the above program and save it as (say) *test.for*.

Open a DOS box and enter a command line of the form:

```
FTN77 TEST /LINK
```

This will produce an executable file called *test.exe* that can be run from the Program Manager in the normal way or by entering:

¹ Throughout this guide INTEGER values are assumed to be 32 bits long unless otherwise stated.

TEST

at the DOS box command line.

Tip Under Windows 95 or Windows NT 4.0 (or later) you may wish to create a shortcut icon on your desktop to run your program. To do this simply right click on an empty part of the screen and select New and Shortcut. Then supply the full path name of your executable program. If you change the program this will not destroy the shortcut, which will continue to reference the latest version of your program

The above procedure will run your test program and display a window that presents the output. This program illustrates how ClearWin+ can be used to convert a simple DOS application without the need to attend to any Windows programming features. It automatically generates what is called a *ClearWin* window that handles the input and output in a Windows environment.

Further information about compiling and linking is given in Chapter 22 whilst Chapter 23 gives detailed information about using *ClearWin* windows. The intervening chapters deal with what are called *format* windows using `winio@`. *Format* windows provide a quick and easy method for developing fully featured Windows applications.

Using FTN77 and FTN95

Salford Win32 compilers use 32 bit integers (unless configured otherwise). This guide assumes that integers are 32 bits long unless otherwise stated.

The examples in this manual have been written using Salford Fortran 77 syntax. Since this syntax is a subset of Fortran 90/95, the programs can be used for the FTN95 compiler with little or no change. However, users of FTN95 may wish to use ClearWin+ modules instead of an include file called `windows.ins`. The relevant module is called `mswin.mod` which is accessed via the Fortran 90/95 statement:

USE MSWIN

`mswin.mod` makes reference to the following modules which can be used separately.

<code>mswin32.mod</code>	Windows API functions and parameters.
<code>clrwin.mod</code>	ClearWin+ functions and parameters.
<code>msw32prm.mod</code>	Windows API printer functions and parameters.

3.

ClearWin+ tutorial

This tutorial shows you how to develop a simple Windows application using a Salford Fortran compiler and ClearWin+. It takes you through a step-by-step process leading to an interactive program that produces the factors of a user-supplied integer. This “Number Factoriser” is one of the demonstration programs that is supplied with ClearWin+. The full source code can be found in the file called *factor.for*.

The tutorial is based on a number of files called *factor1.for*, *factor2.for*, etc. that can be found in the ClearWin+ demonstration directory installed on your hard disk. These files present stages in the development of the full program. The files are listed in the tutorial together with a detailed line-by-line explanation of the purpose and effect of the code. The idea is that you should compile, link and run *factor1.for* and then read the explanation for this file. Then proceed to *factor2.for* and so on.

Step 1

Begin by loading Windows and opening a DOS box.

Now use your text editor to list *factor1.for*.

Here is the text of *factor1.for*. Line numbers have been added so that each line can be referenced in the explanation below¹.

```
1      WINAPP 0,0,'CWP_ICO.RC'
2
3      PROGRAM factor1
4      IMPLICIT NONE
5      INCLUDE <windows.ins>
6      INTEGER ans
```

¹ Throughout this guide INTEGER values are assumed to be 32 bits long unless otherwise stated.

```

7      ans=winio@('%ca[Number Factoriser]')
8      END

```

You can compile and link this program with a command line of the form:

```
FTN77 FACTOR1 /LINK
```

This produces a Windows executable *factor1.exe* that can be run by typing

```
FACTOR1
```

from the DOS box command line.

Alternatively you can run the executable from the Program Manager.

factor1.for simply opens a *format* window with a caption like this:



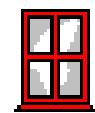
The illustrations throughout this manual have been produced using Windows 95. If you are using another version of Windows then the output will differ slightly in its appearance.

The quickest way to terminate the application is to press Alt-F4.

Now that you have seen the application running, have a look at the code. Before looking at line 7, which is the key line in this program, let's get some preliminaries out of the way first.

Line 1 provides information for the linker that is called when you use /LINK on the compiler command line. It causes the linker to produce a Windows executable. For a Win32 application, the zero values on this line are redundant. (Some later compilers allow you to omit these values.)

cwp_ico.rc is the name of what is called a *resource script* that can be found in the same directory as *factor1.for*. This particular resource script provides information about an icon that is embedded in the application. In the program this "Window" icon is not used in the application itself but is simply available as a icon that identifies the application when it is installed in the Program Manager or its equivalent.



WINAPP must be the first compiler directive in the file and must appear before the main program.

Line 4 is recommended as a standard line for all programs and routines that use the file *windows.ins*. This line is a compiler directive that ensures that all variables are

given an explicit type. It is included because the names of many of the Windows parameters are very long and consequently they are easy to misspell. Without this directive, such bugs are very hard to find. The file *factor.for* is too short for this to matter, but never-the-less we adopt the practice here as a matter of routine.

Line 5 is a call to include the file *windows.ins*. The diamond brackets indicate that the file is located within the Salford compiler directory. This file contains type declarations for many of the ClearWin+ and Windows API routines and parameters. The only information of relevance to this particular main program is that *winio@* returns an INTEGER result (in this guide, all integers are 32 bits long unless otherwise stated). Note that, if you are only using *winio@* from the ClearWin+ library, then you will reduce the compilation time by replacing the INCLUDE line by

```
INTEGER winio@
```

This brings us to line 7 and the heart of this and most ClearWin+ applications. *winio@* is a ClearWin+ function that allows you to create a *format* window that displays all manner of graphical user interface (GUI) objects such as menus, buttons, pictures, formatted text, list boxes and icons. It also allows you to specify what action is to be taken in response to the options that you present to the user. This means that this one function can be used to avoid the complexities of writing a GUI application based on calls to the standard API library combined with a detailed resource script and complex call-back functions.

The first (and in this case only) argument of *winio@* is called a *format string*. In the present case the format string starts with %ca and this is an example of what is called a *format code*. The format code %ca stands for ‘caption’ and it supplies a title for the *format* window. The title itself appears in square brackets after the format code.

There are many format codes available and all of them are represented by a % sign and a two letter format code. Additional information is often supplied (as here) by adding text enclosed in square brackets. This text is called a *standard character string*. We shall see later that sometimes extra information is placed between the % sign and the two letters. In other cases the format code is allowed to collect information from additional arguments that are presented after the initial format string argument of *winio@*.

This brings us to the end of step 1 in our development process.

Step 2

Here is the text of *factor2.for*.

```
1   WINAPP 0,0,'CWP_IC0.RC'  
2
```

```

3  PROGRAM factor2
4  IMPLICIT NONE
5  INCLUDE <windows.ins>
6  INTEGER ans,number
7  number=1
8  ans=winio@('%ca[Number Factoriser]&')
9  ans=winio@('%il&',1,2147483647)
10 ans=winio@('Number to be factorised: %rd',number)
11 END

```

Note that *number* has been added to line 6 and & has been inserted towards the end of line 8. Lines 7, 9 and 10 have also been included.

If you compile, link and run this program the output looks like this.



A flashing text cursor appears in the inner box and the user is able to enter any integer in the range from 1 to 2147483647. As before Alt-F4 terminates the application.

The “&” character in line 8 is simply a device to allow the description of the *format* window to continue in the next call to *winio@* (on line 9 in this case). Using this device avoids long format strings with long lists of associated arguments.

Line 9 contains the format code %il. This specifies the *integer limit* for the input on line 10. %il takes two (INTEGER) values that are provided as extra arguments to *winio@*. The lower limit is 1 and the upper limit (2147483647) is the largest possible 32 bit integer. The “&” character allows continuation in the next *winio@* call.

In line 9 the format string begins with some text that comes before the next format code which is %rd. This text is simply copied to the window. The text includes the space before %rd. Without the %il, the user would be able to enter negative or zero values which would not make sense in this application.

The %rd format code is used for user integer input. It creates an edit box and displays the value of the INTEGER variable (in this case *number*) that is provided as the next argument of *winio@*. The initial value of *number* in line 7 has been chosen to be in the permitted range. Users will find that they are only able to type in integers in this range. Each time a single digit is typed, *number* is updated so that it always holds the value that is visible on the screen.

Now we are ready for step 3.

Step 3

Now that we have a window and the user is able to input an integer, the next stage is to add a button to initiate the factorising process. *factor3.for* includes code for such a button. Here is the text:

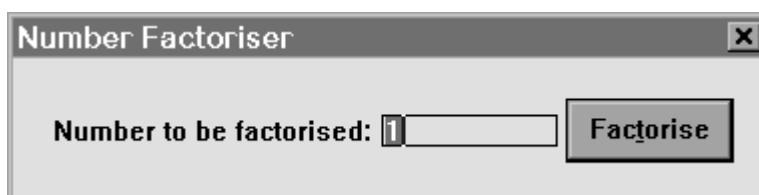
```

1  WINAPP 0,0,'CWP_IC0.RC'
2
3  PROGRAM factor3
4  IMPLICIT NONE
5  INCLUDE <windows.ins>
6  EXTERNAL factoriser
7  INTEGER ans,number
8  number=1
9  ans=winio@('%ca[Number Factoriser]&')
10 ans=winio@('%il&',1,2147483647)
11 ans=winio@('Number to be factorised: %rd&',number)
12 ans=winio@('%ta%`^bt[Fac&torise]',factoriser)
13 END
14
15 INTEGER FUNCTION factoriser()
16 factoriser=1
17 END

```

Note that lines 6, 12 and 15 to 17 have been added to *factor2.for*. Also an “&” character has been inserted after %rd on line 11. We need to look at line 12 in detail. The other new lines simply declare an external function which for the moment merely returns the value 1.

When you compile, link and run this program the output looks like this.



But at the moment nothing happens when you click on the button.

Line 12 of the program uses the format code %bt to provide the button. The text on the button is given as a *standard character string* (in square brackets). The “&” character in “Fac&torise” has the visual effect of producing an underscore on the next letter (‘t’ in this case). The result is that ‘t’ on the keyboard can be used as an

alternative to clicking on the button when used in combination with the Alt key i.e. Alt-T.

Note that two special characters, a grave accent (`) and a caret character (^), have been included after the % sign. These are two out of a set of four characters called *format modifiers* (the others are the tilde (~) and the question mark (?)). As it happens, all four modifiers may be used with %bt, but the number of modifiers that a format code can take and the effect each modifier has varies from one format code to another.

In the present case of the button format %bt, the grave accent means that this button is the default button. The default button has a slightly different appearance and is the one that is selected when the **Enter** key is pressed.

The caret means that a call-back function (called *factoriser* in this case) is provided as the next argument of `winio@`. This is the function that is to be called when the user clicks on the button. Call-back functions that are used with `winio@` must have no arguments and must return an integer value. It is advisable that this function be declared EXTERNAL.

Finally we note that a tab (represented by %ta) has been placed before the %bt format code. This has been used to separate the button from the text before it. Alternatively, you could simply use spaces.

A call-back function must return one of the following values:

-
- | | |
|---------------|--|
| 0 or negative | Closes the <i>format</i> window. |
| 1 | The window remains open and the whole window is refreshed to allow any changes to be displayed. |
| 2 | The window is left open with no refresh. If anything needs to be updated before the call-back returns, a call to <code>window_update@</code> is used to refresh individual components. |
| 3 | Used only with %mg. |
-

Step 4

The next thing we need is a child window in order to present the results of the factorising process. *factor4.for* includes the code for such a window but we shall leave out the code that calculates the results until the end. Here is the text of *factor4.for*:

```

1  WINAPP 0,0,'CWP_IC0.RC'
2
3  PROGRAM factor4
4  IMPLICIT NONE

```

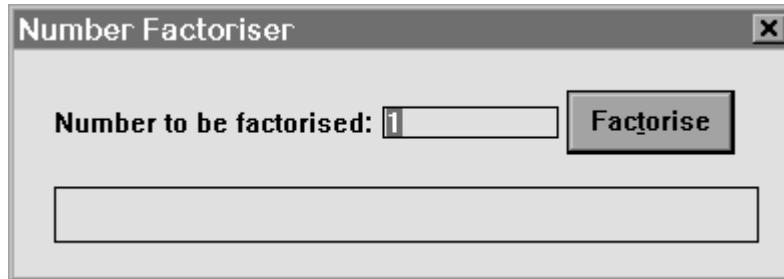
```

5  INCLUDE <windows.ins>
6  EXTERNAL factoriser
7  INTEGER ans,number
8  CHARACTER*50 str
9  COMMON number,str
10 number=1
11 str=' '
12 ans=winio@('%ca[Number Factoriser]&')
13 ans=winio@('%il&',1,2147483647)
14 ans=winio@('Number to be factorised: %rd&',number)
15 ans=winio@('%ta%`^bt[Fac&torise]&',factoriser)
16 ans=winio@('%2n1%ob%42st%cb',str)
17 END
18
19 INTEGER FUNCTION factoriser()
20 INTEGER number
21 CHARACTER*50 str
22 COMMON number,str
23 WRITE(str,*)number
24 CALL window_update@(str)
25 factoriser=1
26 END

```

Lines 8, 9, 11, 16 and 19 to 26 have been added and as before an “&” character has been placed at the end of the format string on line 15.

When the program is compiled the result looks like this:



At the current state of development, when you click on the “Factorise” button, the number that has been entered is simply copied to the new child window.

The allocation of the variable *number* to a common block in lines 9, and 22, provides a method for passing this variable to the *factoriser* function. You will recall that call-back functions like *factoriser* do not have any arguments. Line 23 stores the value of *number* as a character string in *str*. *str* is also held in common because it is used for output on line 16:

```
ans=winio@('%2n1%ob%42st%cb',str)
```

On line 16, %2n1 provides two line feeds so that the child window that follows is placed below the existing controls.

The %ob format code is used to open a box at the current position. It automatically marks the position of the top left-hand corner of a box that will be drawn when a corresponding %cb (close box) is encountered. %cb automatically marks the position of the bottom right-hand corner of the box. The box is simply provided as a border. The enclosed area has no special attributes. In the present case the box is used to enclose a string that is produced by the %st format. The string is located in *str* and the width of the associated area is 42 characters. However, the standard width of a character in this context will be the maximum width of all the characters of the proportionally spaced font.

Finally, line 24 has the effect of updating the string on the screen in order to reflect a change in the contents of *str*.

Step 5

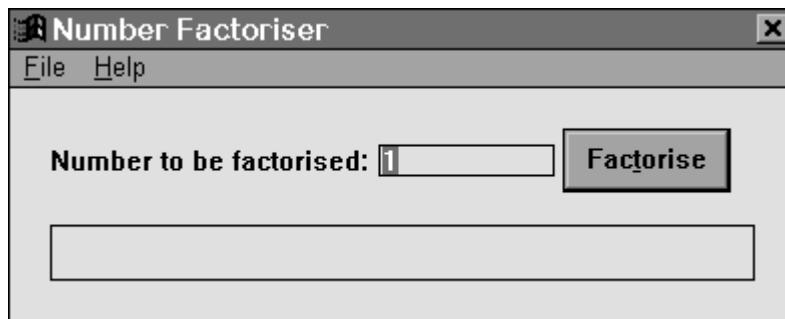
Now we shall add a menu bar with an associated “About” dialog box in order to create *factor5.for*. Here is the text, but to make it easier to understand, only those parts of *factor4.for* that have changed are listed in full.

```
6   EXTERNAL factoriser,about
12  ans=winio@('%ca[Number Factoriser]&')
13  ans=winio@('%mn[&File[E&xit]]&','EXIT')
14  ans=winio@('%mn[&Help[&About Number Factoriser]]&',about)
15  ans=winio@('%il&',1,2147483647)
19  END
20
21  INTEGER FUNCTION factoriser()
28  END
29
30  INTEGER FUNCTION about()
31  IMPLICIT NONE
32  INCLUDE <windows.ins>
33  INTEGER ans
34  ans=winio@('%ca[About Number Factoriser]&')
35  ans=winio@('%fn[Times New Roman]%ts%bf%cnTutorial&',2.0D0)
36  ans=winio@('%ts%4n1&',1.0D0)
37  ans=winio@('%cnProgram written to demonstrate%2n1&')
38  ans=winio@('%ts%tc%cn%bfClearWin+&',1.5D0,RGB@(255,0,0))
39  ans=winio@('%tc%sf%2n1%cnby&,-1)
```

```
40 ans=winfo@('%2n1%cnSalford Software&')
41 ans=winfo@('%2n1%cn%9`bt[OK]')
42 about=1
43 END
```

A new call-back function called *about* has been added to line 6 and the code for this function has been added to the end of *factor4.for* in lines 30 to 43. The new menu bar has been included as lines 13 and 14 after the caption in line 12.

This is what the compiled program displays on the screen.



When you click on the “Help” menu and then on the “About Number Factoriser” item, you get the dialog box below.

Now for the details of how this effect is created. First look at lines 13 and 14 for the menu bar.

```
13 ans=winfo@('%mn[&File[E&xit]]&','EXIT')
14 ans=winfo@('%mn[&Help[&About Number Factoriser]]&',about)
```

The menu format is provided by %mn. Square brackets enclose a list of menu topics, with each topic having an optional embedded list of associated menu items. There are two menu topics here, “File” and “Help”. These provide for two drop down menus each of which (in this example) has one item . The “&” character is used in the same way as on a button in order to provide an accelerator key.



Each menu item requires a call-back function and these are provided as further arguments to `winioc@`. The first one ('EXIT') refers to what is called a *standard ClearWin+ call-back function*. When a *standard* call-back function is used, the code for the function is supplied by ClearWin+ rather than by the programmer. In this case the effect of calling the function is simply to close the application.

The second call-back function (*about*) is supplied by the programmer. In this case the *about* function displays a dialog box that describes the application. When the user clicks on the OK button, the dialog box closes but the application remains active. If *about* returned zero instead of a 2, the application would also close when the OK button was used.

All we need to do now is look at the new call-back function for the “About” dialog. Let’s concentrate on the format codes that we have not seen before in this tutorial.

`%fn` defines a new font for the text that follows. In this case the font is “Times New Roman” and this font is used for the subsequent text up to the `%sf` format code on line 39. `%sf` stands for “standard font” and has the effect of resetting all text attributes (font, size, colour, bold, italic, and underline) to the default settings.

`%ts` is used to set the text size. It takes one DOUBLE PRECISION argument. The value `2.0D0` on line 35 doubles the standard text size and then the standard size is restored on line 36 with the value `1.0D0`. Line 38 uses a text size of one and a half times the standard and then the standard size is restored with `%sf` on line 39.

`%bf` on line 35 is used to provide bold faced font whilst `%cn` has the effect of centring the line of text in the dialog box. In this example `%cn` is cancelled by the next

newline format (%nl). On line 38, %tc changes the colour of the text to red. The colour is provided by the call to the ClearWin+ function RGB@. RGB@ takes three INTEGER arguments in the range 0 to 255. These provide the red, green and blue intensities. Note that %ts comes before %tc and uses the 1.5D0 argument on line 38. %tc is also used on line 39. In this case the argument (-1) restores the system text colour. This is one of the screen colours that the user can set from the Control Panel in the Program Manager.

Finally line 41 produces the OK button. As before the grave accent makes this the default button, whilst the value 9 has the effect of widening the button to a 9 character width. This button does not have a call-back function. As a result, when the user clicks on the button, the *format* window closes. If there are a number of buttons like this, then the value returned by winio@ (*ans* in this case) gives the number of the button that was pressed.

Step 6

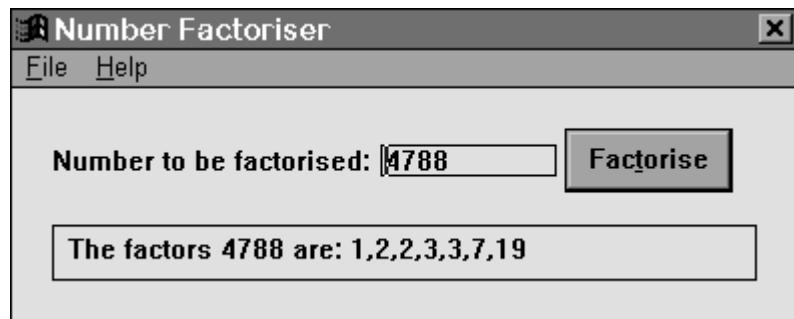
factor6.for contains the finished program. In this program the *factoriser* function has been developed to do the calculations needed to factorise the number that is supplied by the user. The details of this simple Fortran code are not of direct interest to us in this tutorial but here is the code for those who want to read it.

```

INTEGER FUNCTION factoriser()
INTEGER number,n,k
CHARACTER*50 str,val
COMMON number,str
WRITE(val,'(i11)')number
CALL trim@(val)
str='The factors of '//val(1:LENG(val))//' are: 1'
n=number
DO k=2,n
  IF((n/k)*k.EQ.n)THEN
    WRITE(val,'(i11)')k
    CALL trim@(val)
    CALL append_string@(str,', '//val)
    n=n/k
    IF(n.GT.1)GOTO 1
  ENDIF
END DO
CALL window_update@(str)
factoriser=1
END

```

This is what the final program displays on the screen:



This brings us to the end of the tutorial. In this one demonstration program you have been introduced to some of the important format codes that come with `winio@`. More importantly you have been able to experience the ease with which it is possible to develop a GUI application with ClearWin+.

4.

Format windows

Introduction

This chapter presents an overview of the ClearWin+ function `winio@` that was introduced in Chapter 3. Further details are given in the following chapters up to Chapter 21 which provides a detailed reference.

`winio@` is used to lay out and control a window in order to include a wide variety of controls, child windows, toolbars, etc.. Of course, this task could be carried out by making direct use of the Windows API. Although the direct method does provide maximum flexibility, it also requires large amounts of complex coding in order to control the interface. In contrast, the `winio@` function requires a bare minimum of code to achieve elaborate window interfaces.

Windows applications typically have a great deal of “low level” functionality. For example, a window with several input controls will let the user move between fields using the tab key or the mouse. Likewise, buttons change their appearance slightly when they acquire focus. Fortunately you do not need to write code to program this functionality into your application, as it is already supplied by ClearWin+.

The following example illustrates the power and flexibility of `winio@`. In this example `winio@` is used to produce a window that includes a menu bar with associated standard call-back functions. The window contains a so-called *edit box* that is used to display and edit a file. The menu and edit box is fully functional. Three buttons are also shown on the right hand side of the window. These are used to illustrate how buttons are added to the window. As the example stands, clicking on any of these buttons has no effect since code for the corresponding processes has been omitted.

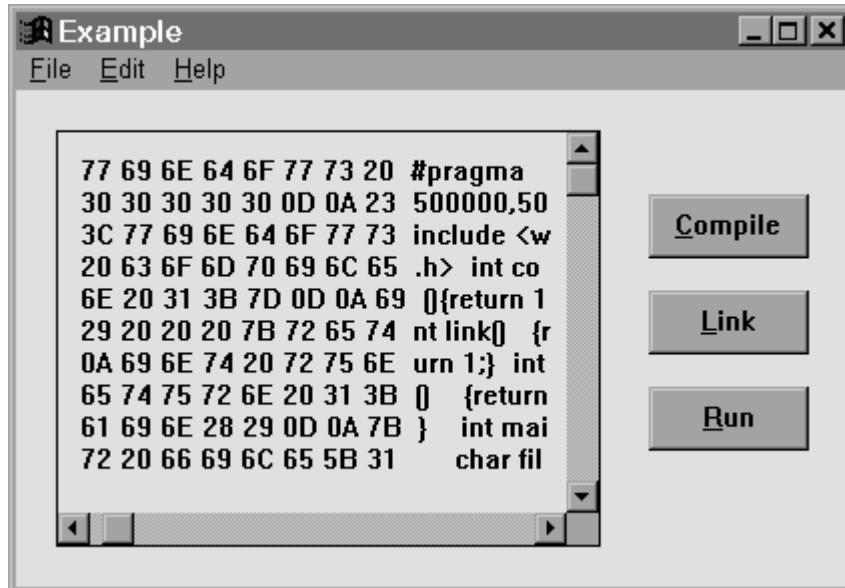
```
WINAPP
INCLUDE <windows.ins>
```

```

CHARACTER*129 file,new_file,help_file
INTEGER compile,link,run,i
EXTERNAL compile,link,run
help_file='myhelp.hlp'
i=winio@('%ca[Example]&')
i=winio@('%mn[&File[&Open]]&','EDIT_FILE_OPEN','.*',
+           file)
i=winio@('%mn[[&Save]]&','EDIT_FILE_SAVE','.*',
+           new_file)
i=winio@('%mn[[Save &As]]&','EDIT_FILE_SAVE_AS','.*',
+           new_file)
i=winio@('%mn[[E&xit]]&','EXIT')
i=winio@('%mn[&Edit[&Copy]]&','COPY')
i=winio@('%mn[[Cu&t]]&','CUT')
i=winio@('%mn[[&Paste]]&','PASTE')
i=winio@('%mn[&Help[&Contents]]&','HELP_CONTENTS',
+           help_file)
i=winio@('%mn[[&Help on help]]&','HELP_ON_HELP',
+           help_file)
c--- Define a 30x10 edit box %eb ---
i=winio@('%ww%pv&')
i=winio@('%30.10eb[vscrollbar,hscrollbar]&','*',0)
c--- Define buttons that are to be displayed ---
i=winio@('%2n1      %^7bt[&Compile]&',compile)
i=winio@('%2n1      %^7bt[&Link]&',link)
i=winio@('%2n1      %^7bt[&Run]',run)
END
c--- Call-back functions that do nothing ---
INTEGER FUNCTION compile()
compile=1
END
INTEGER FUNCTION link()
link=1
END
INTEGER FUNCTION run()
run=1
END

```

This is what the window looks like on the screen:



The remainder of this chapter and the following chapters up to and including Chapter 21 provide a detailed explanation of how to set out the parameters of `winio@`.

Call-back functions

A ClearWin+ call-back function takes no arguments and returns an integer result. As a result, such functions may be coded in C, C++, or Fortran.

When a call-back function returns, it indicates what is to happen next by its return value. A negative or zero value signals an exit and closes the parent format window. The corresponding call to `winio@` will return the absolute value of this quantity. A return value of 1 will cause a full update of the format window so that any changes are made visible. This should be used sparingly because if a call-back is used repeatedly, then the whole display area may flicker due to the frequent screen updates. A more efficient approach is to use a return value of 2. This will not cause the format window to close nor will it update the window. In order to refresh any part of the display that needs to be refreshed, a call to `window_update@` will suffice. This will also improve the response since only small sections of the display will be redrawn.

In summary return values can be:

≤ 0	Closes associated window.
1	Returns from call-back with display refresh.
2	Returns from call-back with no refresh.
3	Used with %mg only.
4	Used with %lv and %bv only.
≥ 5	Reserved for future use.

Tip Returning a negative value can be very convenient. Consider a window with a list box and OK and CANCEL buttons. Typically it is required that a double click on a list box item will select that item and close the window as if the OK button had been pressed. To achieve this the list box call-back function should return -1, so that `winio@` will return 1 as if the OK button had been pressed - thus saving a flag and more complex program logic.

It is often convenient to use one call-back function for a variety of reasons. Whenever a call-back function is invoked, ClearWin+ defines a string that specifies the reason for the call. This string is obtained by calling `clearwin_string@('CALLBACK_REASON')`. Testing the string makes it easy to determine the reason for the call. See page 342 for further details.

Window formats

The ClearWin+ function `winio@` has the following form:

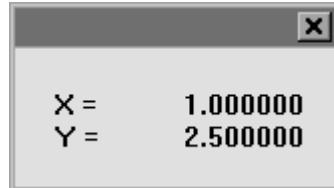
```
INTEGER winio@(format,...)
CHARACTER*(*) format
```

where the ellipses ... represents a list of arguments whose length depends on what appears in the character variable *format*. The character string is called a *format string*. The list must only contain arguments of type INTEGER, DOUBLE PRECISION, CHARACTER or EXTERNAL functions.

At its very simplest, `winio@` may be used to display information:

```
WINAPP
INCLUDE <windows.ins>
DOUBLE PRECISION x,y
x=1.0
y=2.5
```

```
c--- %wf display a floating point value ---
i=winio@('X = %ta%wf%nlY = %ta%wf',x,y)
END
```



%wf is an example of a *format code*. The %wf *format code* displays a floating point value with six decimal places as the default. New lines are represented by %nl, and tabs %ta operate on a grid of 8-character intervals or by setting tab stops with the %tl format (see page 90). %2nl (for example) is used for two new lines and %3ta (for example) is used for three tabs. Form-feed (%ff) has a special meaning in a format window. It moves the current position to the left margin, beneath any existing controls. The window is automatically constructed with a size to suit its contents.

%wd, %wx, %ws, %wc, %we, and %wg are alternatives to %wd & %wf with the following usage:

%wd	Integer output.
%wx	Hexadecimal integer output.
%ws	String output.
%wc	Character output.
%wf	Floating point output in decimal form.
%we	Floating point output in exponent form.
%wg	Floating point output in decimal or exponent form.

Each of these output format codes can be modified in order to control the manner in which the information is presented. Details are given on page 45.

By using the additional format codes described below and in the following chapters, the resulting windows can prompt for and display a wide variety of information.

Each format code begins with the percent (%) character, followed by optional size information of the form *n* or *n.m* where *n* and *m* are integer constants (e.g. %6.4wd). *n* and/or *m* can be replaced by an asterisk “*” with the corresponding value(s) being supplied as one or two INTEGER arguments in the argument list. After the optional size information, a two-letter code is used to define the format. This two letter code is not case sensitive. Some format codes have mandatory size information. These sizes are represented by uppercase *N* and *M* in the text.

Four special characters, known as *format modifiers*, are often inserted after the size information and before the two-letter code. The first of these is the caret character (^) which indicates that a call-back function is supplied in the argument list (after any other arguments required by the format code). The tilde character (~) is used with certain format codes to control the disabling (greying) of the control. The grave accent character (`) is also used to modify the action of certain codes. These characters may only be used with the codes that define them as detailed in Chapter 21.

The fourth character is the question mark (?) that is used with any format that defines a control (as opposed to formats such as %ww that modify the appearance of the window as a whole). The question mark signifies that a help string is supplied in the format. This string is displayed at the bottom of the window or as a help “bubble” whenever the user’s cursor lies over the corresponding control. The text is either surrounded by square brackets, or a “@” character is placed in the format string to indicate that the help string is supplied as an extra argument. The text string that is supplied in either of these two forms is known as a *standard character string*.

Many format codes are supplied with a list of options. When used, these are placed in a comma-separated list and set between square brackets after the two-letter format code. For example

```
%gr[black,rgb_colours]
```

If none of the options are used then the list, together with the square brackets, may be omitted. However, if the option list is empty and a help string is required, then square brackets must be inserted to represent the empty list. These brackets are followed by the help string. For example:

```
%20.10?eb[] [Edit box]
```

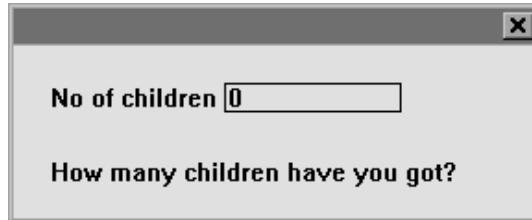
Here is an example that uses a help string. It prompts the user for an integer (%rd) and augments the window with an additional help string.

```
WINAPP
INCLUDE <windows.ins>
INTEGER n_ch
n_ch=0
i=winio@('No of children %?rd'
+ //'[How many children have you got?]',n_ch)
END
```

The following code would produce the same result:

```
CHARACTER*80 help
help='How many children have you got?'
n_ch=0
i=winio@('No of children %?rd@',n_ch,help)
END
```

Help strings may include line feed characters (char(10)) if necessary. The area to contain the help text will be sized to fit the dimensions of the largest string. The help text can be positioned at another location (e.g. inside a box) by using the %he format. Alternatively, %bh provides for ‘bubble’ help and %th for ‘tooltip’ help (see page 119).



A toolbar requires a help string for each button (see the %tb example on page 56).

Calls to `winio@` may become quite complex. For this reason, a method is available to continue a format over several calls to `winio@`. If the last character in a format string is an ampersand (&) character, this character is removed from the string and the format information is joined seamlessly to the next call to `winio@`.

For example:

```
i=winio@('Enter an integer      %ta%rd%nl&',n)
i=winio@('Enter a second integer%ta%rd%nl&',m)
i=winio@('%cn%`bt[Done]')
```

This creates a single window containing two integer edit boxes together with a button.

Note that, using this technique, it is possible to build a dialog box with a structure that is controlled by your program logic. For example, if the user had already supplied a file name you could omit a file open box, possibly replacing it with a suitable message.

If you choose to write some routines in Fortran and others in C++, then the programming language that you use for the first call to `winio@` for a given format window must be the same as that used in any continuation calls. However, it is possible to create one format window using Fortran and another using C++.

Closing a window

ClearWin+ programs may create a number of windows, some of which might be open at the same time. A ClearWin+ program terminates when both a) the END point of the program (or its equivalent) is reached, and b) all windows created by the program have been closed by the user.

Normally the user is provided with a number of ways to close a window. There is usually a system menu with a ‘Close’ item (Alt+F4) at the top left of a window. Selecting this item is equivalent to clicking on the box at the top right of a window. The programmer can use %cc with a call-back function to trap this event.

The programmer can also add an ‘Exit’ item to a %mn menu. This item will either be linked to a call-back function provided in the program or to the standard ‘EXIT’ call-back.

Other exit routes can be provided via buttons (using for example %bt or %tt):

- a) When you click on the button that has no call-back function attached, the parent window will close. If there is only one window open, the effect is the same as if the standard ‘EXIT’ call-back were attached to the button.
- b) If there is no call-back function attached to a button and the associated window has been created within a call-back function, then the window will close. The affect on the window’s parent will depend on the value returned by the call-back (see page 22). A zero return will close the window’s parent. The same is true for any call-back, not just those attached to buttons.

Another approach to window closure is to attach a control variable to a window. When a window (or dialog) is created, by default it will be equivalent to a *modal* dialog box. A modal dialog box is one that must be closed before other windows in the same application can be accessed. Any code that follows the `winiow@` calls for a particular modal dialog box, will not be processed until the user closes the dialog.

If you want more than one window to be accessible (the equivalent of a *modeless* dialog box) then you can use %lw (see page 122) to leave a window open and to attach a control variable to the window. The presence of %lw causes any code that follows the `winiow@` calls for a particular window to be processed immediately. ClearWin+ sets the value of a %lw control variable to -1 (minus one). If the control variable is subsequently set a non-negative value, the window will close when that variable is updated (see below).

Notes

1. %lw is used together with %fr and %aw to create a multi-document interface.
2. %cv also provides a control variable for use with %aw but in this case the window will be a child window and will not be left open.

Updating windows

Many of the format codes of the `winiow@` function take pointers to data that may change. For example, the bar control variable in the %br format will typically change

as some time consuming process proceeds. Likewise, consider the following partial window definitions:

```
CHARACTER*50 myform  
myform='TEST 1'  
i=winfo@('%ca@&',myform)
```

The address of the variable *myform* has been passed to *winfo@*, and if the contents of the string *myform* alter, it is desirable to update the window accordingly. In general, if you alter data that is in use by a window, the window will be updated if it is obscured and restored for any reason.

However, to obtain an immediate update, the subroutine

```
SUBROUTINE window_update@(variable)
```

should be called. The argument is the variable whose value has changed. In reality the address of the variable is passed and this address must be the exact address originally passed to *winfo@*. For example, in the above example, the address of the first character in the string would be passed to *window_update@* using

```
CALL window_update@(myform)
```

The update would not work if the address of the second character (i.e. *myform(2:2)*) were passed, even though this address points to part of the caption. The *window_update@* function will update all controls, captions, etc. that depend on the variable. It may also be used with a *grey-control* variable (see for example %bt, page 51), or the control variable associated with the %lw format. In the latter case, setting the control variable to a non-negative value means that the window will close.

Note that, in general, *window_update@* will only update those objects that depend on the variable whose address is supplied. However, there is no guarantee that other portions of the display will not become updated in the process.

It is desirable that *window_update@* be called at a sensible rate. Perhaps no more than once or twice per second for a given variable. Frequent updates may produce an unpleasant flicker.

Advanced use of format windows

It is convenient at this point to mention a few advanced approaches to ClearWin+ programming. If you are still unfamiliar with how to program with *winfo@*, it would be better to skip this section for the moment.

There are three ways in which the usage of `winio@` can be varied under program control. First, you can use `winio@` statements within flow control constructs such as DO and IF constructs. Here is a simple example.

```

WINAPP
  INTEGER, n,i,winio@
  COMMON n
  EXTERNAL cb_func
  i=winio@('Number of lines: %4rd&',n)
  i=winio@('%2n1%cn%^tt[Show]',cb_func)
END
-----
C-----
      INTEGER function cb_func()
      INTEGER n,i,j,winio@
      COMMON n
      DO j=1,n
        i=winio@('Line %wd%nl&',j)
      ENDDO
      i=winio@(' ')
      cb_func=1
END

```

The first dialog box allows you to enter a number. Clicking on the button then produces another dialog box with the given number of lines of output. Notice that the format string within the DO loop is terminated by an ampersand (&) to allow the description of the dialog box to continue on the next loop. The final call to `winio@`, acts as a terminator. It contains only a format string with a single space character. The same idea can be used with other flow control constructs containing calls to `winio@`.

A second method for varying the use of `winio@` under program control involves changes to the format string. The next example illustrates the idea.

```

WINAPP
  INTEGER n,i,winio@
  COMMON n
  EXTERNAL cb_func
  n=1
  i=winio@('Format 1 or 2: %4rd&',n)
  i=winio@('%2n1%cn%^tt[Show]',cb_func)
END
-----
C-----
      INTEGER function cb_func()
      INTEGER n,i,j,winio@
      COMMON n
      CHARACTER*10 fmt

```

```

fmt='Invalid value %wd'
IF(n.eq.1) THEN
    fmt='Format %wd'
ELSE if(n.eq.2) THEN
    fmt='FORMAT %wd'
ENDIF
i=winio@(fmt,n)
cb_func=1
END

```

In this example, the choice of format string for the output depends on the value entered into the edit box. The illustration is not very useful in itself but it does point the way to the idea that it is possible to construct format strings dynamically under program control.

The third and final approach to the dynamic use of `winio@` takes the last idea one stage further. As it is we cannot easily change the format codes within a format string because different format codes take different arguments in the `winio@` call. For example, if `%wd` were changed to `%wf` in the above example, then the call to `winio@` would require a real valued argument rather than an integer after the format string. Although you could solve this problem by using complex IF or CASE constructs, ClearWin+ provides an alternative and potentially more powerful approach via the following routines:

```

SUBROUTINE ADD_WINIO_INTEGER@(I)
INTEGER I

SUBROUTINE ADD_WINIO_FLOAT@(X)
DOUBLE PRECISION X

SUBROUTINE ADD_WINIO_CHARACTER@(S)
CHARACTER*(*) S

SUBROUTINE ADD_WINIO_FUNCTION@(F)
EXTERNAL F

```

Using these routines, you create the argument list before calling `winio@`. After creating the list, you then call `winio@` with the format string as its only argument. Here is a simple example.

```

WINAPP
INTEGER, n,i,winio@
COMMON n
EXTERNAL cb_func
n=1
i=winio@('Format 1 or 2: %4rd&',n)
i=winio@('%2n1%cn%^tt[Show]',cb_func)

```

```

      END
C-----
      INTEGER function cb_func()
      INCLUDE <windows.ins>
      INTEGER n,i
      COMMON n
      CHARACTER*20 fmt
      DOUBLE PRECISION x
      x=n
      IF(n.eq.1) THEN
          fmt='Integer %wd'
          CALL add_winio_integer@(n)
      ELSE
          fmt='Real %wf'
          CALL add_winio_float@(x)
      ENDIF
      i=winio@(fmt)
      cb_func=1
      END

```

This example is too simple to do justice to the concept but it is sufficient to point the way.

Format codes grouped by function

The following table presents the format codes under various headings. Details are given in chapters 5 to 18. A reference guide is presented in Chapter 21. An index and summary of the format codes appears in the next section, in addition to the index at the end of this user's guide.

1)	Interactive I/O:	%co, %dd, %df, %dl, %dr, %fl, %il, %rd, %rf, %rs
2)	Controls:	%bc, %br, %bt, %bv, %bx, %el, %ga, %hx, %ib, %ld, %ls, %lv, %ms, %rb, %sl, %tb, %tt, %tv, %vx
3)	Bitmaps, Cursors, Icons:	%bi, %bm, %cu, %dc, %gi, %ic, %mi, %si
4)	Menus and accelerator keys:	%ac, %es, %mn, %pm, %sm
5)	Layout and positioning:	%ap, %bd, %cn, %dy, %ff, %gd, %gp, %nd, %nl, %nr, %pv, %rj, %rp, %ta
6)	Boxes:	%cb, %ob
7)	Displaying text:	%bf, %eq, %fn, %gf, %ht, %it, %pd, %sd, %sf, %st, %su, %tc, %ts, %tl, %ul

8) Help:	%bh, %he, %ht
9) Graphics:	%dw, %gr, %og, %pl
10) Main window attributes:	%ca, %cv, %de, %lw, %mv, %nc, %ns, %sp, %sv, %sy, %sz, %wp, %ww
11) Background colour	%bg
12) Child windows:	%aw, %ch, %cl, %cw, %fr, %hw, %if, %lc, %ps, %sh, %uw
13) Formats for call-back functions:	%cc, %fs, %ft, %mg, %rm, %sc
14) Edit boxes:	%eb, %pb, %re, %tx
15) Data output:	%ss, %wc, %wd, %we, %wf, %wg, %ws, %wx

Alphabetical index of format codes

The following table provides a list and summary of the special format codes. Details are given in chapters 5 to 19. A reference guide is presented in Chapter 21.

	<i>page</i>
%ac Accelerator key format.	88
%ap Absolute positioning of next control	95
%aw Attach window format.	123
%bc Button colour format - specifies the colour of the next %bt button.	217
%bd Specifies all four window borders individually.	218
%bf Switch to bold font.	97
%bg Background colour format.	127
%bh Bubble help format.	119
%bi Supply an icon for the next button.	54
%bk Allow a right mouse click to be used on the next %bt button.	220
%bm Bitmap format draws a bitmap.	75
%br Bar format - draws a horizontal or vertical bar which is partially filled with a user-selected colour.	61
%bt Button format - defines a button with text.	51
%bv Hierarchical tree-view, alternative to %tv.	222
%bx Adds a raised grey bar to a tool bar.	59
%ca Caption format - defines the title of a dialog box.	121
%cb Box close format - closes a box opened by %ob.	82

%cc	Closure control format - provides a link to a call-back function by which the user controls the action to be taken when a window is closed.	130
%ch	Child window format - inserts a child window.	123
%cl	Displays a colour palette.	73
%cn	Centring format - forces everything which follows it, up to the next new line or form-feed character, to be centred in the window.	89
%co	Control option format - used to modify subsequent %rd, %rf, and %rs boxes.	40
%cu	Establishes a cursor for the next control in a format.	76
%cv	Control variable format.	123
%cw	Embeds a <i>ClearWin</i> window.	128
%dc	Establishes a default cursor for the window.	76
%dd	Increase/decrease button for an integer.	41
%de	Disables some other window while the current window is active.	232
%df	Increase/decrease button for a floating point value.	41
%dl	Allows a call-back function to be called at regular intervals via a timer.	130
%dp	Parameter box format.	263
%dr	Drag and drop.	125
%dw	Owner draw box format - provides owner draw boxes.	157
%dy	Provides a vertical displacement by a non-integral number of character depth units.	235
%eb	Edit box format - presents a edit box in which a text file is displayed and modified.	107
%el	Inserts an editable combo box.	236
%ep	Parameter box format.	263
%eq	Equation format - inserts a mathematical equation in a window.	100
%es	Escape format - causes the program to exit when the Escape key is pressed.	237
%ew	Exit Windows format - provides a call-back function which is called when the user shuts down.	238
%fb	Remember current font for subsequent buttons.	238
%ff	Form feed. Move down to below any existing controls.	239
%fh	Font handle - use a font created by the API function CreateFont .	239
%fl	Floating point limit format - specifies the lower and upper limits for subsequent %rf formats.	41
%fn	Font name format - selects a font for subsequent text.	97
%fp	Parameter box format.	263

%fr	MDI frame format - defines a frame to contain child windows attached by %aw.	123
%fs	File selection format - specifies the working directory and file filter for subsequent file open call-backs.	105
%ft	File filter format - specifies filter information for subsequent file open call-backs.	106
%ga	Gang format - enables radio buttons and/or bitmap buttons to be ganged together so that if one is switched <i>on</i> the others are switched <i>off</i> .	59
%gd	Programmer's grid format - supplies a temporary grid to help with the positioning of controls.	95
%gf	Get font handle format.	97
%gi	Draws a (possibly animated/transparent) GIF image.	243
%gp	Get window position format - used to get the current co-ordinates of the window position.	94
%gr	Graphics format - provides a rectangular area for Salford graphics routines.	ch 15
%he	Help format - specifies the location of help information.	119
%ht	Hypertext document attachment.	ch 19
%hw	Obtains the window handle of the format window.	121
%hx	Attaches a horizontal scroll bar to the next control.	247
%ib	Image bar - a replacement format for %tb.	247
%ic	Icon format - draws an icon.	77
%if	Initial focus format - specifies that the next control will have the initial focus.	249
%il	Integer limit format - specifies the lower and upper limits for subsequent %rd formats.	42
%it	Switch to italic font.	97
%lc	Obtain the control handle.	249
%ld	LED format - displays a round LED (a coloured disk with a black border).	251
%ls	List box format	62
%lv	Listview format	252
%lw	Leave window open format - allows <code>w i n i o @</code> to return without closing the window that it creates.	122
%mg	Message format - provides a call-back function for a given Windows message.	257
%mi	Minimise icon format - supplies the name of an icon resource to be used if the window is minimised.	77
%mn	Menu format - used to attach a menu to the window.	81

%ms	Defines a multi-selection box that stores its settings in an array. <i>Similar to %ls</i>	64
%mv	Move format - provides a call-back function that is to be called when the user moves or resizes the window.	259
%nc	New class name format - provides a specified class name for the application.	260
%nd	Never-down format - prevents controls from sliding down when sizing a window.	260
%nl	New line	260
%nr	Never-right format - prevents controls from sliding to the right when sizing a window.	261
%ns	Disable screen saver.	133
%ob	Box open format - defines the top left hand corner of a rectangular box into which subsequent objects are to be placed until a corresponding %cb format is encountered.	91
%og	Graphics format - provides a rectangular area for OpenGL graphics routines	ch 16
%pb	Parameter box format.	263
%pd	Is used to insert a Sterling pound symbol '£' regardless of editor mode.	100
%pm	Supplies a popup menu so that when the right mouse button is pressed in the main window a menu appears.	84
%ps	Property sheet - layer windows to produce a card index style.	129
%pv	Pivot format - used to create a pivotal point for any subsequent re-sizing of the window.	93
%rb	Radio button format - defines a radio button or check box with text supplied directly.	55
%rd	Integer input format - creates an edit box and displays an integer that can be updated.	37
%re	Creates an multi-line edit box.	268
%rf	Floating point input format - creates an edit box and displays a floating point value that can be updated.	37
%rj	Right justifying format - forces everything which follows it, up to the next new line or form-feed character, to be right justified in the window. A window margin, if any, is still applied.	90
%rm	Read message format - Provides the name of a call-back function to handle messages from another ClearWin+ application.	270
%rp	Relative position format - Sets the position of the next control relative to the current position.	270
%rs	Character string input format - creates an edit box and displays a character variable (i.e. a string) that can be updated.	40

%sc	Start-up call-back. Causes a call-back to be called once when a window is opened - useful for drawing initial %gr data.	123
%sd	Subscript text format.	272
%sf	Standard font format - resets to default text attributes after use of any combination of %bf, %it, %ul, %fn, and %ts.	98
%sh	Used with property sheet format.	130
%si	Standard icon format - defines a standard icon that is to be placed to the left of the block of text that follows the descriptor.	78
%sl	Produces a slider control using a floating point variable to determine its position.	44
%sm	System menu format.	86
%sp	Set window position format - used to initialise the position of the window on the screen.	94
%ss	Allows the settings of selected controls to be auto saved and loaded from an INI file.	48
%st	Variable string format - lays a string out in a field of <i>n</i> characters. The string is re-drawn each time the window is renewed.	100
%su	Superscript text format.	277
%sv	Allows the program to become a screen saver executable.	132
%sy	Style format - changes the style of a window.	277
%sz	Size window format.	126
%ta	Insert tab 8-character intervals or to predefined tab stops. <i>See %tl</i> .	279
%tb	Bitmap button and toolbar format - defines a bit mapped button or a whole tool-bar.	56
%tc	Sets the colour of subsequent text.	99
%th	Bubble help in the form of a ‘Tool tip’.	281
%tl	Sets up tab locations for use with %ta.	90
%tp	Parameter box format.	263
%ts	Text size format - used to scale the text font size either up or down.	99
%tt	Textual toolbar format.	54
%tv	Hierarchical tree-view	68
%tx	Displays an array of text in a rectangular region, defined by its parameters.	103
%ul	Underline text.	97
%up	Parameter box format.	263
%uw	User window allows windows code to be interfaced to ClearWin+.	131
%vx	Attaches a vertical scroll bar to the next control.	286
%wc	Character output.	45
%wd	Integer output.	45

%we	Floating point output in exponent form.	46
%wf	Floating point output in decimal form.	47
%wg	Floating point output in decimal or exponent form.	47
%wp	Supplies the name of a wallpaper bitmap which is used as a back drop to its contents.	131
%ws	Character string output.	48
%ww	Window control format - causes the resultant window to look like a normal application window, rather than a dialog window.	126
%wx	Hexadecimal integer output.	48

5.

Input and output

Interactive I/O

Reading numbers or character strings from a window under ClearWin+ is very similar to reading data in a traditional program using READ. However, it is important to remember that each variable must be given an initial value before the window is created since a default value will appear as soon as the window is displayed.

Tip Checkmate can be used to flag an error when a variable that has not been given a value is used.

Those controls that allow the user to edit text in a box (such as %rd) can also transfer text to/from the clipboard using the special ClearWin+ call-back functions 'CUT', 'COPY', 'PASTE', defined in the standard call-back functions described in Chapter 20.

Integer input - %nrd

%rd creates an edit box and displays the value of the corresponding (INTEGER) argument. The integer will be updated whenever the edit field is adjusted. The user will be prevented from creating an invalid or out of range integer (see %il below). By default the edit box will be made large enough to hold integers of the current range, although the integer parameter *n* can be used to override this.

For example:

```
INTEGER p  
i=winio@('%si*Enter an integer: %3rd',p)
```



A call-back function is supplied using the caret (^) modifier. This function is called whenever a change is made.

A grave accent format modifier (`) may be used to make the control read-only. In this form no surrounding box is supplied, and the value displayed can only be changed by the program, using `window_update@` to reflect the changes.

Floating point input - %nrf

`%rf` creates an edit box and displays the value of the corresponding (DOUBLE PRECISION) argument. The number will be updated whenever the edit field is adjusted. The user will be prevented from creating an invalid or out of range number (see `%fl` below). Numbers are entered in decimal or exponent form. For example: -0.0748 or -7.48e-2.

By default the edit box will be made large enough to hold values in the current range, although the integer parameter *n* can be used to override this.

A call-back function is supplied using the caret (^) modifier. This function is called immediately a change is made.

For example, suppose you wanted to prompt for a complex number, and the user is allowed to supply this as an (*x*, *y*) pair or as an (*r*, θ) pair. The box should show the current number in both formats and any change in one format should be reflected in the other. Here is some sample code that has the desired effect.

```

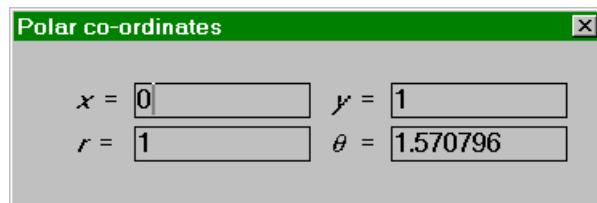
WINAPP
DOUBLE PRECISION x,y,r,theta
COMMON x,y,r,theta
INTEGER i,winio@
EXTERNAL convert_to_xy,convert_to_polar
x=1.0D0
y=0.0D0
r=1.0D0
theta=0.0D0
i=winio@('%ca[Polar co-ordinates]%bg[grey]&')
i=winio@('%3t1&',5,15,21)
i=winio@('    %itx%`it=%ta&')
i=winio@('%^rf&',x,convert_to_polar)
i=winio@('    %ity%`it=%ta&')

```

```

i=winio@('%^rf&',y,convert_to_polar)
i=winio@('%ff %itr%`it=%ta&')
i=winio@('%^rf&',r,convert_to_xy)
i=winio@(' %fn[Symbol]%itq%sf =%ta&')
i=winio@('%^rf',theta,convert_to_xy)
END
C-----
INTEGER FUNCTION convert_to_polar()
DOUBLE PRECISION x,y,r,theta,tiny
COMMON x,y,r,theta
tiny=1.0D-6
r=SQRT(x*x+y*y)
IF(x.LT.tiny.AND.y.LT.tiny)THEN
    theta=0.0
ELSE
    theta=ATAN2(y,x)
ENDIF
convert_to_polar=1
END
C-----
INTEGER FUNCTION convert_to_xy()
DOUBLE PRECISION x,y,r,theta
COMMON x,y,r,theta
x=r*COS(theta)
y=r*SIN(theta)
convert_to_xy=1
END

```



A grave accent format modifier (`) is used to make the control read-only. In this form no surrounding box is supplied, and the value displayed can only be changed by the program, using `window_update@` to reflect the changes.

The %co format (see page 40) is used to modify the way that %rf and %rd call-back functions behave.

Character string input format - %nrs

This format takes one character variable and displays it in a single line edit box so that the user can change it. Optionally, it can be modified with an integer *n* that sets the width of the box in characters. The edit box will then scroll horizontally when necessary. For multiple line edit boxes see %re (page 268) and %eb (page 107).

For example:

```
CHARACTER*80 str
str='Initial string padded with spaces if needed'
i=winio@('%10rs',str)
```

It is important to note that because there is an obvious difference between the widths of characters e.g. 'i' and 'W', the width is specified in 'average characters'.

A grave accent format modifier (`) is used to make the control read-only. In this form no surrounding box is displayed, and the string can only be changed by the program, using window_update@ to reflect the changes. This should be used in place of %ws when it is necessary to display a string which needs to be updated. A similar effect is produced with %st. (see page 100). The caret is added to provide a call-back function that will be called on every change to the edit window.

Control option format - %co[option-list]

This format is used to modify the form of subsequent edit boxes associated with %rd, %rf, and %rs. Currently, four options are available:

check_on_focus_loss	Perform checks and call-back (and update the corresponding data item) with %rd, %rf, and %rs on loss of <i>focus</i> only. This is particularly valuable if numeric limits have been imposed by %il or %fl (see the Glossary for a definition of the <i>focus</i>).
data_border	Put a border round %rd, %rf, and %rs boxes (default).
full_check	Perform checks and call-back on %rd, %rf, and %rs for every change (default).
no_data_border	Omit border round %rd, %rf, and %rs boxes.

For example:

```
i=winio@('%co[no_data_border]%il%rd&',1,10,j)
i=winio@('%n1%co[data_border]%rd',k)
```

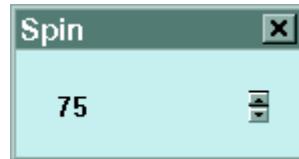
places a box around the second data area but not the first.

Integer increase/decrease format - %dd

Supplies an spin control button to go with an %rd edit box. %dd takes one integer argument that specifies the amount the value in the edit box is to be increased or decreased when the user clicks on the control. The resulting value is always a multiple of the increase. A call-back function can be supplied with the corresponding %rd format. %dd may also be used with %rs.

For example:

```
INTEGER val,i
val=0
c--- %`rd displays an integer - not user changeable ---
i=winio@('%ca[Spin]%%d%`rd',5,val)
END
```



Floating point increase/decrease format - %df

Supplies an spin control button to go with a %rf edit box. %df takes one DOUBLE PRECISION argument that specifies the amount the value in the edit box is to be increased or decreased when the user clicks on the control. The resulting value is always a multiple of the increase. A call-back function can be supplied with the corresponding %rf format. For example

```
DOUBLE PRECISION x
INTEGER i,winio@
EXTERNAL cb_func
i=winio@('%df%^rf',1D0,x,cb_func)
```

Floating point limit format - %fl

Specifies the lower and upper limits (as DOUBLE PRECISION arguments) for subsequent %rf formats. The lower and upper limits are supplied as arguments.

For example:

```
DOUBLE PRECISION x
i=winio@('%fl%rf',0.0D0,10.0D0,x)
```

The lower limit should be specified with care unless %co[CHECK_ON_FOCUS_LOSS] is used. If, for example, the lower limit were 500.0 and the user desired to enter the

value 525.0, the number would be rejected as soon as the first digit 5 is entered. If CHECK_ON_FOCUS_LOSS is not used then the lower limit should be chosen to avoid this problem.

Integer limit format - %il

Specifies the lower and upper limits (as INTEGER arguments) for subsequent %rd formats. As for %fl, the lower limit should be specified with care unless %co[CHECK_ON_FOCUS_LOSS] is used.

For example:

```
INTEGER p
p=0
i=winio@('%si*Enter a small positive integer:%il%rd',0,99,p)
END
```

Parameter box - %N.Mpb[options]

A parameter block is a control that has been designed for use in complex simulation programs where you may wish to browse a large set of parameter names and values, with the option to select a value and modify it. A parameter block is defined using %N.Mpb where *N* is the width of the block in average characters, and *M* is the number of lines in the control. The control will scroll if necessary to display all the parameters. %pb takes the option sorted to cause the parameters to be alphabetically sorted by name. After a parameter block has been defined, parameters can be attached by subsequent format codes as follows:

- %dp - Integer parameter
- %fp - Floating point parameter
- %tp - Text parameter
- %ep - Enumerated parameter (a set of named alternatives)
- %up - User parameter (just activates a call-back)

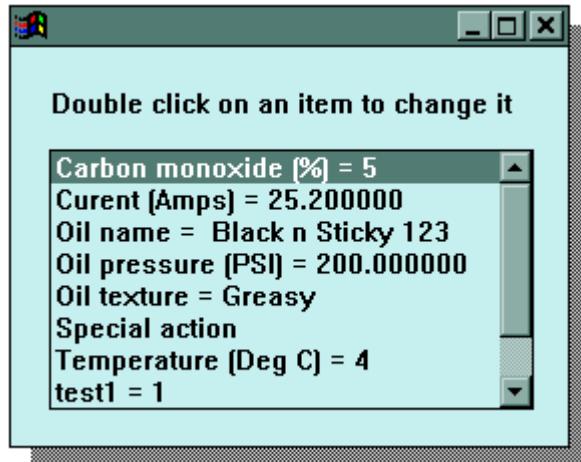
Each of these format codes is followed by a *standard character string* that gives the text to appear in the box. The question mark (?) modifier can be used with any of the additional format codes as illustrated below but %pb itself takes no modifiers. If a question mark is used with a parameter, the associated help string is displayed when the cursor is positioned above that parameter. One use for the user parameter (%up) is to activate another parameter block to achieve a hierarchical effect.

The various parameter types are illustrated by the following example:

```
WINAPP
```

```
INCLUDE <windows.ins>
EXTERNAL action
INTEGER i,texture_type
DOUBLE PRECISION v,p,a
CHARACTER*8 textures(6)
CHARACTER*20 name
DATA textures/'Sticky','Messy','Dirty','Greasy','Slimy',
+ 'Very slippery'
DATA /k1,k2,k4,k5,v,texture_type,a/1,2,4,5,4.5,4,35.4/
p=120.6
name='Black n Sticky 123'
i=winio@('%ww[casts_shaddow]%ca[Parameter setting]&')
i=winio@('Double click on the parameter you wish to'
+           //change:%2n1&')
i=winio@('%30.8pb[sorted]&')
i=winio@('%dp[test1]&',k1)
i=winio@('%dp[test2]&',k2)
i=winio@('%ep[Oil texture]&',textures,6,texture_type)
i=winio@('%dp[Temperature (Deg C)]&',k4)
i=winio@('%dp[Carbon monoxide (%)]&',k5)
i=winio@('%fp[Curent (Amps)test6]&',a)
i=winio@('%10.3fp[Voltage]&',v)
i=winio@('%fp[Oil pressure (PSI)]&',p)
i=winio@(
1 '%?tp[Oil name][What is the official name of this oil?]&,'
2 name)
i=winio@('%up[Special action]',action)
END

INTEGER FUNCTION action()
INCLUDE <windows.ins>
i=winio@('No special action available'
+           //yet%2n1%cn%bt[Continue]')
action=2
END
```

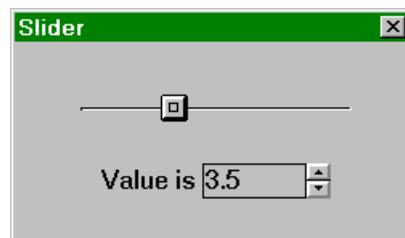


Slider format - %ns!l[options]

This format produces a vertical or horizontal (default) slider control *n* average characters wide. It takes three arguments which are all floating point values. The first is a variable to hold the current value, the second its lower limit and the third is the upper limit.

For example:

```
WINAPP
INTEGER i,wini@_
DOUBLE PRECISION max,min,value,def
DATA max,min,def,value/10.0,0.0,1.0,5.0/
i=wini@('%ca[Slider]%bg[grey]%'`cn&)
i=wini@('%20s1&',value,min,max)
i=wini@('%2n1 Value is &')
i=wini@('%df%f1%6rf',def,min,max,value)
END
```



Data output

Each of the format codes %wd, %wx, %wf, %we, %wg, %ws, and %wc, can be modified in order to control the way in which the information is presented in the dialog window. The modifiers are all optional and appear after the % sign with the following general form:

%[flags][width][.precision] code

where *code* is one of *wd*, *wf*, etc.

[width] represents a positive integer that specifies the minimum number of characters that are to be output. More characters will be output rather than truncate the result. Where fewer than the minimum are needed, by default the field is padded with spaces on the left. Note, however, that with a proportionally spaced font, padding may have little apparent effect.

[.precision] represents a full stop followed by a positive integer that specifies the precision of the output. For floating point values, this integer is usually the required number of decimal places in the output.

Alternatively the [width] and [.precision] values may be replaced with an asterisk (*).

For example:

%[flags][*.*] code

Integer values must then be supplied as arguments, one for each asterisk.

[flags] represents one or more of the characters: minus “-”, zero “0”, plus “+”, and the hash symbol “#”. These modifiers have various effects depending on the current format code. For example they are used to force left justification, to pad with zeros rather than spaces, and to force a plus sign for a non-negative result.

In the following examples ∇ is used to represent a space.

%wc

%wc is used for the output of a single character. The width modifier may be used to pad out a field with spaces and the minus flag will cause the character to be left justified in such a field.

%wd

%wd is used for the output of integer values.

format	output for i=123	comment
%wd	123	minimum width
%6wd	VVV123	minimum of 6 characters, right justified
%6.4wd	VV0123	shows 4 digits, with a leading zero
%-6wd	123VVV	left justify
%06wd	000123	pad with zeros on the left
%+6wd	VV+123	force a + sign when not negative

Use this format only for displaying formatted text output that does not need to be refreshed when a variable is modified. Otherwise use %`rd in conjunction with the window_update@ call. The grave accent makes the %rd format a read-only text display box.

%we

%we is used for the output of floating point values in exponent form. The exponent part (following the letter 'e') always contains a sign and two digits.

format	output for x=0.0126	comment
%we	1.2607000e-02	6 decimal places, otherwise minimum width
%14we	VV1.260000e-02	minimum of 14 characters, right justified
%10.2we	VV1.26e-02	10 characters, 2 decimal places
%6.0we	V1e-02	6 characters, no decimal point
%-9.1we	1.3e-02VV	9 characters, 1 decimal place, left justified
%010.3we	01.260e-02	right justified, padded with zeros
%+10.2we	V+1.26e-02	force a + sign when not negative
%#8.0we	VV1.e-02	forces a decimal point when the <i>precision</i> modifier is zero

%wf

%wf is used for the output of floating point values in decimal form.

format	output for x=1.26	comment
%wf	1.260000	6 decimal places, otherwise minimum width
%10wf	VV1.260000	minimum of 10 characters, right justified
%6.2wf	VV1.26	6 characters, 2 decimal places
%2.0wf	V1	2 characters, no decimal point
%-6.1wf	1.3VVV	6 characters, 1 decimal place, left justified
%07.3wf	001.260	right justified, padded with zeros
%+6.2wf	V+1.26	force a + sign when not negative
%#4.0wf	VV1.	forces a decimal point when the <i>precision</i> modifier is zero

%wg

%wg is used for floating point values and has essentially the same effect as %wf unless the supplied value cannot adequately be represented in %wf form; in which case %wg has the same effect as %we. To be specific, the %we form is only used if the exponent is less than -4 or greater than or equal the precision value. Note, however, that with %wg, trailing zeros after a decimal point are removed. Where appropriate, the decimal point is also removed.

format	value	output	comment
%wg	0.0126	0.0126	
%10wg	1.26e-4	VV0.000126	exponent is -4
.3wg	126.7	126.7	exponent is less than the precision
%9.2wg	1.267e-5	V1.27e-05	exponent is less than -4
%7.2wg	100.0	VV1e+02	exponent is equal to the precision

%ws

%ws is used for the output of character strings.

format	string	output	
'Mr %ws Bloggs'	'Frederick'	Mr Frederick Bloggs	(1)
'Mr %9ws Bloggs'	'Fred'	Mr Fred Bloggs	(2)
'Mr %`9ws Bloggs'	'Fred'	Mr Fred Bloggs	(3)
'Mr %.4ws Bloggs'	'Frederick'	Mr Fred Bloggs	(4)
'Mr %-9ws Bloggs'	'Fred'	Mr Fred Bloggs	(5)

- (1) minimum field width,
- (2) minimum of 9 characters, right justified,
- (3) right justified in a space the width of 9 characters
- (4) maximum of 4 characters,
- (5) minimum of 9 characters, left justified.

%wx

%wx is similar to %wd but is used to output hexadecimal values. A grave accent (`) modifier is used to specify an upper case letter hexadecimal output.

Save settings to .INI file - %ss[filename/section]

Saves the supplied variables to an .INI file that is automatically located in the Windows directory, not the current working directory (the format of an .INI file is illustrated below). When the program is rerun the variables stored in the .INI file will automatically be restored (whenever the windows containing the %rs, %rd or %rf controls are displayed). The file name string (without the extension .INI) must be supplied in square brackets followed by a (/) character and the section name that will be used in the .INI file. The item name (within the section) is formed automatically from the text that appears immediately before the (%rd, %rf, %rs) format code.

A control argument of type INTEGER must also be supplied. This is a ‘save’ control flag that prevents the automatic saving of values when set to zero. This is useful when an abort/abandon path is taken rather than a clean exit, thus preventing erroneous or incomplete data from being stored. %ss should be placed before the first (%rd, %rf, %rs) format code.

For example:

```
INTEGER num,save_control,i
CHARACTER*20 directory
save_control=1
```

```
num=0
directory=' '
i=winio@('%ss[file/section]&', save_control)
i=winio@('Users: %rd%nl&', num)
i=winio@('Path:  %rs', directory)
```

If the user enters ‘12’ and ‘c:\temp’ into the two edit boxes, this creates a file called *file.ini* in the Windows directory containing:

```
[section]
Users=12
Path=c:\temp
```


6.

Button controls

Button format - %nbt[button_text]

This format defines a button where *button_text* represents a *standard character string* consisting of the text that is displayed on the button. The text must either be enclosed in square brackets, or an @ symbol is used to indicate that the text is supplied as an argument. The buttons of a dialog box are numbered from 1. The `winio@` function returns the number of a button used to close a window or zero if it was closed in some other way (this assumes the button does not have a call-back). Button names may include the “&” character in order to provide accelerator keys.

For example:

```
i=winio@('Idiot!%ta%bt[&Sorry]')
```



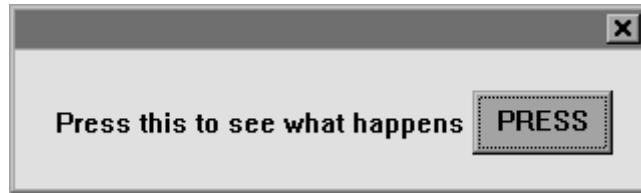
A grave accent (`) is used to indicate that the button is the default when the `Enter` key is pressed. A default button has a slightly different appearance.

The caret character (^) may be used to define a call-back function that is called when the user clicks on the button (or, if the button is the default button, when the user presses the `Enter` key).

For example:

```
INTEGER i,winio@
EXTERNAL func
i=winio@('Press this to see what happens ^')
```

```
i=winio@('%^bt[PRESS]',func)
END
c---Function to do something---
INTEGER function func()
func=1
END
```



The buttons in a *format* window (both %bt and %tt) are numbered from 1 in the order in which they are defined in the format string. When the user clicks on a button that does not have a call-back function, the *format* window closes and the corresponding winio@ call returns the number of the button pressed. For example:

```
PRINT *,winio@('%bt[OK]%ta%bt[Cancel]')
```

outputs the value 1 if ‘OK’ is pressed, 2 if ‘Cancel’ is pressed and zero if the window is closed without clicking on a button.

Alternatively, if a button has a call-back function that returns a negative value, then clicking on the button causes the *format* window to close and the corresponding winio@ call returns this value made positive. For example:

```
INTEGER winio@,i,func
EXTERNAL func
i=winio@('%^bt[OK]%ta%bt[Cancel]',func)
PRINT *,i
END
INTEGER FUNCTION func()
func=(-1)
END
```

displays the value +1 if ‘OK’ is pressed, 2 if ‘Cancel’ is pressed and zero if the window is closed without clicking on a button.

Normally a button created with %bt is permanently enabled. However, if the tilde (~) format modifier is used then an extra INTEGER argument must be supplied (before any call-back function). This argument provides an integer that controls the state of the button. When this integer is zero the button is greyed and cannot be used. When this integer is 1 the button is enabled. Typically this control integer would be altered by code responding to another control. For example, after a successful file-opening, a

number of options might be enabled. Note that there is no reason why the same control integer should not control several buttons and/or menu items.

For example:

```

WINAPP
INCLUDE <windows.ins>
INTEGER grey_cntrl
EXTERNAL open_f,save_f,save_as_f
COMMON grey_cntrl
c---Only the OPEN button is initially available
grey_cntrl=0
i=winio@(%^bt[Open] &,open_f)
i=winio@(%~^bt[Save] &,grey_cntrl,save_f)
i=winio@(%~^bt[Save as],grey_cntrl,save_as_f)
END
c---
INTEGER function open_f()
INTEGER grey_cntrl
COMMON grey_cntrl
grey_cntrl=1
open_f=2
END
.....

```



The width of the button, in average characters, can be supplied using the parameter *n* (e.g. %10bt[Save]) rather than this being determined by the width of its text. A button is created wide enough for at least *n* characters (possibly more, if the font is not mono-spaced). This enables a number of buttons of the same size to be created. %fn and %ts can be used before %bt in order to modify the button text. %bi is used to add an icon to a button whilst %bc is used to change its background colour.

The text on a button will be placed on more than one line if a line feed character (CHAR(10)) is spliced into the text at the appropriate point.

For example:

```

button_name='Press'//CHAR(10)//'here'
i=winio@('%bt@',button_name)

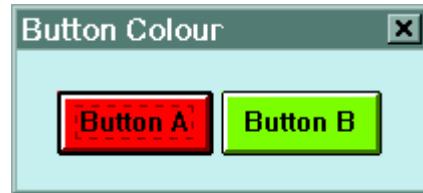
```

Button colour - %bc[*colour*]

This format is used in order to specify the background colour for the next %bt button only. The colour is specified in the same way as for %bg (see page 127). The grave accent modifier is used to allow the colour to be changed dynamically after the button has been formed (see page 217).

For example:

```
i=winio@('%ca[Button Colour]&')
i=winio@('%bc[red]&')
i=winio@('%bt[Button A] &')
i=winio@('%bc&',RGB@(128,255,0))
i=winio@('%bt[Button B]')
```



Button icon - %bi[*icon_name*]

This format supplies the name of an icon resource (as a *standard character string*) for the next %bt button. For example: %bi[*tea_icon*]%bt[] where *tea_icon* is defined in an associated resource script (see %ic on page 77). In this example no text would appear on the button. However, in most cases you will probably want to supply button text to go with the icon.

For example:

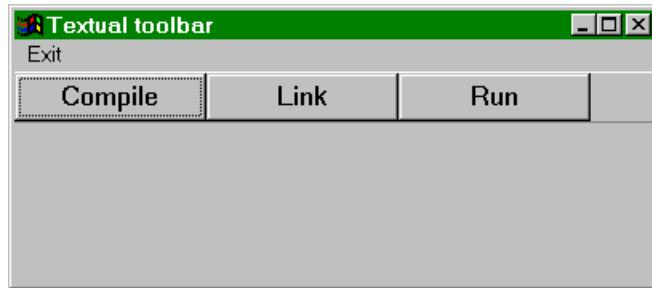
```
%bi[tea_icon]%bt[Tea!]
```

Textual toolbar - %tt[*button_text*]

This format code is much the same as %bt, except that the button is not as tall. Also the width of the button is rounded up to one of a set of standard sizes. This makes it possible to produce a textual toolbar (using a sequence of %tt formats) as used, for example, in the standard Windows Help buttons “Contents”, “Search”, “Back”, etc. When using this format you will probably want to remove the borders from the window using the no_border option of the %ww format.

For example:

```
WINAPP
INTEGER i,winio@
i=winio@('%ww[no_border]%ca[Textual toolbar]&')
i=winio@('%mn[Exit]&', 'EXIT')
i=winio@('%tt[Compile]%tt[Link]%tt[Run]&')
i=winio@('%bx&', 0.0D0)
i=winio@('%bg[grey]&')
i=winio@('%pv%fr', 400,100)
END
```



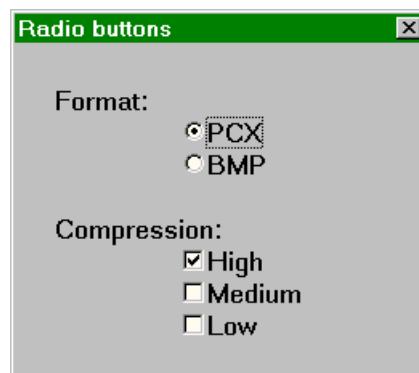
Radio button - %rb[button_text]

This format defines a radio button. The button text is a *standard character string*. It takes a single INTEGER argument that is set to 0 or 1 according to whether the corresponding control is *off* or *on*. The grave accent (`) is used to change this control to a check box. Radio buttons can be ganged together (see %ga page 59) so that only one of a set is selected at any one time. Note that it is not “standard” Windows programming practice to gang check boxes together.

For example:

```
WINAPP
INTEGER i,winio@,r(2),c(3)
DATA r/1,0/c/1,2*0/
i=winio@('%ca[Radio buttons]%bg[grey]&')
i=winio@('Format:%n1&')
i=winio@('%2`ga&',r(1),r(2))
i=winio@('%ta%rb[PCX]%n1&',r(1))
i=winio@('%ta%rb[BMP]%2n1&',r(2))
i=winio@('Compression:%n1&')
i=winio@('%3`ga&',c(1),c(2),c(3))
i=winio@('%ta%`rb[High]%n1&',c(1))
```

```
i=winio@( '%ta%`rb[Medium]%n`&',c(2))
i=winio@( '%ta%`rb[Low]',c(3))
END
```



The caret character (^) is used in association with a call-back function and this will be called whenever the item is selected.

Bitmap button and toolbar - %n.mtb

%tb has been superceded by %ib. For new code use %ib instead. See pages 58 and 247 for further details.

This format defines a bit mapped button or a whole toolbar. The names of three bitmap resources representing the *off* state, *on* state, and *depressed* state should be supplied as character string arguments, followed by an INTEGER argument that represents the state (0 = *off*, 1 = *on*). In the absence of a call-back function, pressing the button toggles the state, but does not close the *format* window. The correct order for the arguments is as follows:

```
'BMPoff', 'BMPon', 'BMPdown', ControlVariable, Callback
```

Any button on the toolbar can be set to an inactive/disabled state by using the (~) tilde format modifier. An extra bitmap will be required to represent this state.

For example:

```
'BMPoff', 'BMPon', 'BMPdown', 'BMPdisabled', ControlVariable,
GreyControlVariable, Callback
```

The caret character (^) is used in association with a call-back function and this will be called whenever the user clicks on the button.

In order to obtain the effect of a non-toggling button, simply specify the same bitmap for the *on* and *off* states, and supply a call-back function to respond to each button press. Note that bitmaps may look different on screens of different resolutions.

Where necessary the program should supply the appropriate bitmaps for the resolution in use. Bitmaps can be created using the standard Microsoft “Paint” accessory.

Here is a simple example of the use of bitmap buttons:

```
WINAPP 0,0,'d20.rc'
INCLUDE <windows.ins>
INTEGER answer
i=winio@('Hello%ta%tb','BT_OFF','BT_ON','BT_DOWN',answer)
END
```

The file *d20.rc* contains:

```
BT_OFF BITMAP "BT1.BMP"
BT_ON BITMAP "BT2.BMP"
BT_DOWN BITMAP "BT3.BMP"
```

Using the optional parameters *n* and *m* it is possible to create a whole toolbar as a rectangular array of buttons *n* items across by *m* deep. For example, a 1-deep horizontal bar of 10 items would be represented by %10.1tb (or just %10tb) together with 30 bitmap resources as the corresponding arguments.

If call-back functions are used, these follow each set of bitmaps thus: ‘off1’, ‘on1’, ‘down1’, func1, ‘off2’, ‘on2’, ‘down2’, func2,... etc. If the “?” modifier is used, then *n* × *m* help strings must be provided. It is particularly valuable to provide such help in the case of toolbar buttons, since their meaning is not always obvious. The following example creates a 3-element horizontal toolbar with help information and call-back functions. The second button is initially set to the *on* state.

```
WINAPP 0,0,'d21.rc'
INCLUDE <windows.ins>
INTEGER b1,b2,b3
EXTERNAL func1,func2,func3
DATA b1,b2,b3/0,1,0/
i=winio@('%3.1?^tb[Apples][Oranges][Pears]',
1    'OFF1','ON1','DWN1',b1,func1,
2    'OFF2','ON2','DWN2',b2,func2,
3    'OFF3','ON3','DWN3',b3,func3)
END
-----
INTEGER function func1()
func1=1
END
....
```

The file *d21.rc* contains:

```

OFF1 BITMAP "BT1.BMP"
ON1 BITMAP "BT2.BMP"
DWN1 BITMAP "BT3.BMP"
OFF2 BITMAP "BT1.BMP"
ON2 BITMAP "BT2.BMP"
DWN2 BITMAP "BT3.BMP"
OFF3 BITMAP "BT1.BMP"
ON3 BITMAP "BT2.BMP"
DWN3 BITMAP "BT3.BMP"

```

Bitmap buttons can be ganged together (see %ga below). A greyed bar can also be drawn around the tool bar (see %bx below).

Image button and image bar - %n.mib

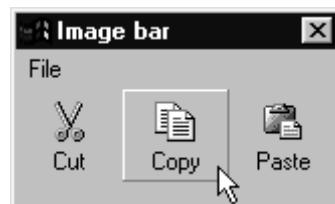
%ib provides an enhanced and simplified replacement for %tb. Like %tb, %ib can be used to create toolbars or rectangular arrays of image buttons. However, only one flat image is required for each button - all the variants are created automatically. Also, text can be automatically added to the image. Here is some sample code that illustrates the use of %ib.

```

WINAPP
INTEGER winio@,i
i=winio@('%ca[Image bar]%'sy[3d]&')
i=winio@('%bd&',0.0D0,0.0D0,0.5D0,1.0D0)
i=winio@('%mn[File[Exit]]&','EXIT')
i=winio@('%3ib[flat]', 'cut/Cut',4,'CONTINUE',
+ 'copy/Copy',4,'CONTINUE', 'paste/Paste',4,'CONTINUE')
END

RESOURCES
cut    BITMAP cut.bmp
copy   BITMAP copy.bmp
paste  BITMAP paste.bmp

```



The argument list for %ib consists of the following triplet repeated for each button in turn: *bitmap_name*, *stat_ctrl*, *cb_func*.

The first part of the string *bitmap_name* gives the name of a bitmap resource. If you want a string to be added to the bitmap then this must come next after an oblique ('/'). This string can include linefeed characters (char(10)).

The integer *stat_ctrl* is used to control the state of the button. The constant value 4 is used to describe a button that does not toggle, i.e. it is in the down state only whilst the mouse button is held down. For a button that toggles, use a variable that will take the value 1 when it is in the up state and the value 2 when in the down state. The state value 3 describes a button that is greyed and disabled. Every button has a call-back function *cb_func*.

The option *flat* gives a toolbar that is initially flat and grey. When the mouse cursor is above a button, the image becomes coloured and the button is raised or depressed depending on its state. When this option is not used the old style of button is obtained.

%ib is designed for bitmaps that have a grey background. If the image has a one-pixel border then the border colour is used as the background colour, otherwise the background is assumed to be light grey (192,192,192). In the above example, the image bitmaps are 48x24 pixels. This results in buttons of equal width when the strings are added. See page 247 for further details.

Tool bar border - %bx

This format will add a raised grey bar effect to any tool bar (created using %bt, %tt, %ib or %tb) set at the top of the window. It takes one floating point argument that specifies the additional depth (in average characters) to be inserted above and below the bar.

Gang format - %Nga

By using %ga, radio buttons (%rb) and/or toolbar buttons (%tb) can be ganged together. At any time at most one control in a gang will be switched *on*. The addition of a grave accent (`) makes sure that there is always one control switched *on*.

The parameter *N* must be present, and have a value greater than 1. *N* specifies the number of controls to be ganged. The argument list must contain *N* INTEGER arguments, that contain the *N* controlling integers. For example, in the toolbar example (see page 57), the three controls could be ganged together by modifying the code thus:

```
i=winio@('%3ga&',b1,b2,b3)
i=winio@('%3.1?^tb[Apples][Oranges][Pears]', 
2    'OFF1','ON1','DWN1',b1,func1,
3    'OFF2','ON2','DWN2',b2,func2,
4    'OFF3','ON3','DWN3',b3,func3)
```

The ganging format code may be placed before or after the formats in which the control variables are actually used. A given variable may be used in more than one control. However, it is unusual for a variable to appear in more than one gang format for a given window. If it does then there must only be one variable that is common and the effect is to merge the two gang sets. If the grave accent is used on one of the gangs then it must be used on both.

7.

Controls other than buttons

Bar format - %Nbr[*options*]

This format draws a horizontal or vertical bar which is partially filled with a given colour. The first argument is a DOUBLE PRECISION control variable that indicates the extent of the fill. This variable should have a value between 0.0D0 and 1.0D0 inclusive. The second argument is an (r,g,b) colour value given by RGB@. N is used to specify the length of the bar, in average characters, when fully filled.

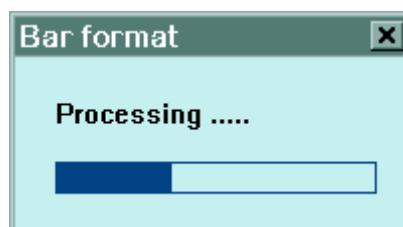
If options are supplied they should be placed in square brackets and separated by commas. The following options are available:

left_right	Bar is drawn horizontally left to right (default).
right_left	Bar is drawn horizontally right to left.
top_bottom	Bar is drawn vertically from the top.
bottom_top	Bar is drawn vertically from the bottom.
no_border	No bar border is drawn.

The bar is updated by using the window_update@ function. This should be called at suitable intervals to provide a smooth progression. This is illustrated in the following example:

```
WINAPP
INCLUDE <windows.ins>
INTEGER ctrl
DOUBLE PRECISION d
d=0.0
i=winio@('%ca[Bar format]&')
i=winio@('Processing .....%2n1&')
i=winio@('%20br&',d,RGB@(255,255,0))
```

```
i=winio@('lw',ctrl)
DO WHILE(d.LT.1.0)
    DO i=1,100000
        END DO
        d=d+0.01
        c---- The data has changed so prompt windows to ---
        c---- update the bar display ---
        CALL window_update@d)
    END DO
END
```



The above program will produce a horizontal bar which will extend to the right as the task proceeds.

Horizontal and vertical scroll bars %hx and %vx

Both %hx and %vx have similar meaning and usage. %vx generates a vertical scroll bar on the right-hand side of the next control and %hx a horizontal scroll bar along the bottom of the next control. %hx and %vx should not be used with controls (like %eb and %cw) that have their own scroll bar mechanism. Three arguments must be supplied to both format codes. The first is an INTEGER argument that determines the ‘page step’. This determines the amount the scroll box (also called the *thumb*) moves each time the left mouse button is pressed inside the scroll bar region. You would normally link this value to the effect of a ‘page change’ in the control. The second argument is the maximum value (call this *max* say) that the scroll bar may generate. The third argument holds the current value of the scroll bar. This is of type INTEGER and will be in the range 0 to (*max* - 1). A call-back function is provided when the caret modifier is used. See also pages 247 and 286.

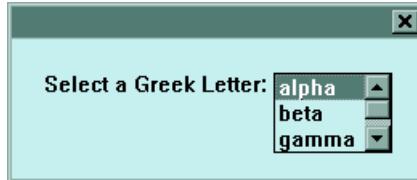
List box format - %n.mls[options]

This format supplies a simple list-box facility. The corresponding arguments are an array of CHARACTER strings, an INTEGER argument giving the size of the array and an INTEGER argument that both sets the initial selection and returns the result.

For example:

```
INTEGER k
CHARACTER greek(4)*5
DATA (greek(i),i=1,4) //'alpha','beta','gamma','delta'/
k=1
i=winio@('Select a Greek letter: %7.3ls',greek,4L,k)
END
```

The characters “7.3” before “ls” specify the width of the list box (as the number of characters) and the depth of the list box (as the number of lines). Pre-setting *k* to 1 would cause “alpha” to be already highlighted when the window is displayed. If the user selects “beta”, then *k* set to 2. If *k* were pre-set to zero, none of the items would be initially selected. If the user were to exit without making a selection, *k* would keep its pre-set value.



The grave accent (`) may be used with %ls to produce a drop-down combo box rather than a list box (for an editable combo box see %el on page 236). The caret (^) character is used in association with a call-back function. This function will be called whenever an item is selected.

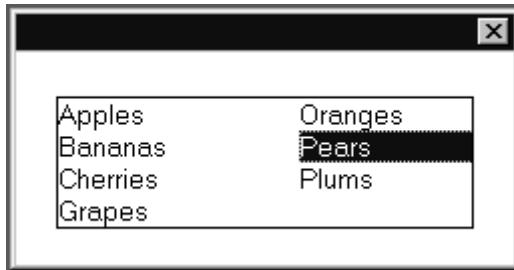
For example:

```
EXTERNAL ls_function
i=winio@('%^ls',array,6L,n,ls_function)
```

(See also the LISTBOX_ITEM_SELECTED parameter in clearwin_info@ page 337).

For a list box (i.e. when a grave accent modifier is not used), the option hscrollbar provides a horizontal scroll bar whilst the option multicolumn provides a multiple column display. For example:

```
CHARACTER*8 fruit(7)
DATA fruit/'Apples','Bananas','Cherries','Grapes',
+          'Oranges','Pears','Plums'/
i=winio@('%24.4LS[multicolumn]',fruit,7,k)
```



Combining `hscrollbar` with `multicolumn` gives a horizontal scroll bar that selects one column at a time.

Multiple selection box - %n.mms

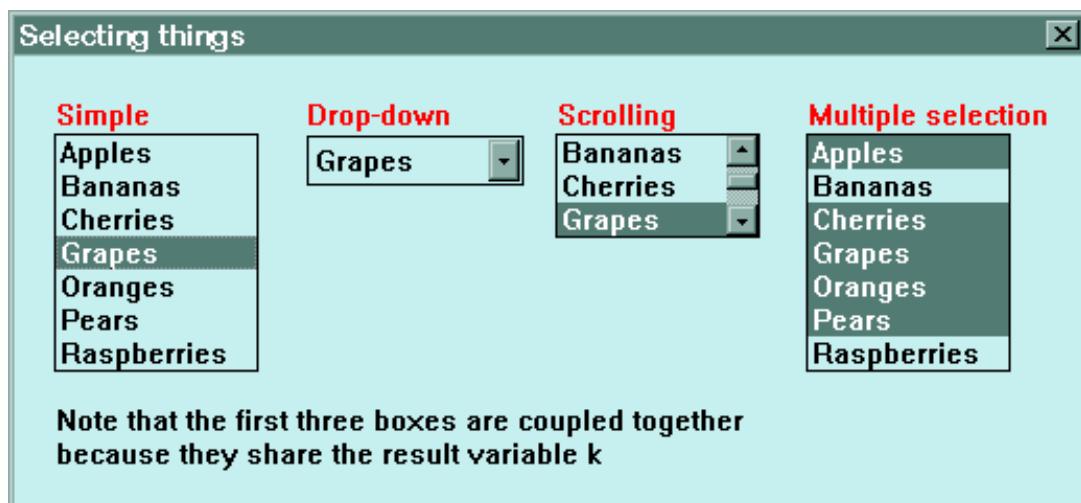
`%ms` creates a list box that allows the user to select more than one item at a time. It is similar to `%ls` but the variable returning the result is replaced by an integer array. The elements of this array should be initialised to a value of zero for initially de-selected items and to a value of 1 for selected items. The array is updated as the user changes the selection.

n and *m* are optional (*m* cannot be specified without *n*). *n* represents the width of the multiple selection box (as a number of characters) and *m* the depth (as the number of lines). `%ms` requires three arguments. The first is an array of `CHARACTER` strings; the second is the number of entries in the list; and the third is an array of type `INTEGER`. The two arrays must have the same number of elements. If the number of elements that are displayed is less than the number in the list then a vertical scroll bar is provided. If a call-back function is supplied then it is called each time the user selects an item.

The following program illustrates some typical cases:

```
WINAPP
INCLUDE <windows.ins>
INTEGER k,i
INTEGER ivec(7)
CHARACTER*10 things(7)
DATA things/'Apples','Bananas','Cherries','Grapes',
+ 'Oranges', 'Pears', 'Raspberries'/
c--- Set the initial values for the selections ---
DATA ivec/0,1,0,0,1,1,0/
i=winio@('%ca>Selecting things]&')
i=winio@('%3tl&',15,30,45)
i=winio@('%tc[red]Simple%taDrop-down%taScrolling%ta&')
i=winio@('Multiple selection%nl%tc[black]&')
```

```
i=winio@('%ls%ta&',things,7,k)
i=winio@('%`ls%ta&',things,7,k)
i=winio@('%10.3ls%ta&',things,7,k)
i=winio@('%ms&',things,7,ivec)
i=winio@('%ff%nlNote that the first three boxes are&
i=winio@('coupled together%nl&')
i=winio@('because they share the result variable k')
END
```



List-view - %lv[options]

A list view control is a window that displays a collection of items, each item consisting of an optional icon and a label. List view controls provide several ways of arranging items and displaying individual items. One of the arguments for %lv is an integer variable in the range 0..3 that represents the type of view: 0=Icon view; 1=Report view; 2=Small icon view; 3=List view. In the report view, additional information about each item can be displayed in columns to the right of the icon and label. The Microsoft Explorer uses a list view in order to display directory information. Here is a Fortran 90 example that illustrates the concept. For further details see page 252.

```
INTEGER N
PARAMETER (N=20)
CHARACTER*36 info(N+1),Colour/icons*60
INTEGER rgb(N),sel(N),view,hwnd,winio@,RGB@
COMMON info,rgb,hwnd
EXTERNAL call_back
```

```

INTEGER call_back,i
sel=0;rgb=0
rgb(1)=RGB@(255,0,0)
rgb(2)=RGB@(0,255,0)
rgb(3)=RGB@(0,0,255)
rgb(4)=RGB@(0,255,255)
rgb(5)=RGB@(255,0,255)
rgb(6)=RGB@(255,255,0)
info(1)='|Colour_90|Red|Green|Blue'
info(2)=Colour('ARed',rgb(1))
info(3)=Colour('BGreen',rgb(2))
info(4)=Colour('CBlue',rgb(3))
info(5)=Colour('DCyan',rgb(4))
info(6)=Colour('EMagenta',rgb(5))
info(7)=Colour('FYellow',rgb(6))
view=1
icons='red,green,blue,cyan,magenta,yellow,blank'
i=winio@('%ww%sy[3d]%ca[Listview sample]&')
i=winio@('%mn[Views[Icon]]&', 'SET',view,0)
i=winio@('%mn[[Report]]&', 'SET',view,1)
i=winio@('%mn[[Small icon]]&', 'SET',view,2)
i=winio@('%mn[[List]]&', 'SET',view,3)
i=winio@('%pv%`^lv[edit_labels,single_selection]&',
+ 240,120,info,N,sel,view,icons,call_back)
i=winio@('%ff%cn%10`rs%lc','Selected colour',hwnd)
END

CHARACTER*36 FUNCTION Colour(name,rgb)
CHARACTER(*) name
INTEGER rgb,r,g,b
CHARACTER*3 rc,gc,bc
r=AND(255,rgb)
g=AND(255,RS(rgb,8))
b=AND(255,RS(rgb,16))
WRITE(rc,'(i3)') r
WRITE(gc,'(i3)') g
WRITE(bc,'(i3)') b
rc=ADJUSTL(rc)
gc=ADJUSTL(gc)
bc=ADJUSTL(bc)
WRITE(Colour,'(6a)') name,rc(1:LEN_TRIM(rc)), '|',
+ gc(1:LEN_TRIM(gc)), '|',bc(1:LEN_TRIM(bc))
END

```

```

INTEGER FUNCTION call_back()
INCLUDE <windows.ins>
INTEGER N
PARAMETER (N=20)
CHARACTER*36 info(N+1)
INTEGER rgb(N),hwnd,row
COMMON info,rgb,hwnd
CHARACTER*36 Colour,name
call_back=2
SELECT CASE(clearwin_string@('CALLBACK_REASON'))
CASE('SET_SELECTION')
    row=clearwin_info@('ROW_NUMBER')
    CALL set_control_text_colour@(hwnd,rgb(row))
CASE('END_EDIT')
    name=clearwin_string@('EDITED_TEXT')
    row=clearwin_info@('ROW_NUMBER')
    info(row+1)(3:)=Colour(name(1:LEN_TRIM(name))//'|',rgb(row))
    CALL window_update@(info)
END SELECT
END

RESOURCES
red      ICON red.ico
green    ICON green.ico
blue     ICON blue.ico
cyan     ICON cyan.ico
magenta ICON magenta.ico
yellow   ICON yellow.ico
blank    ICON blank.ico

```

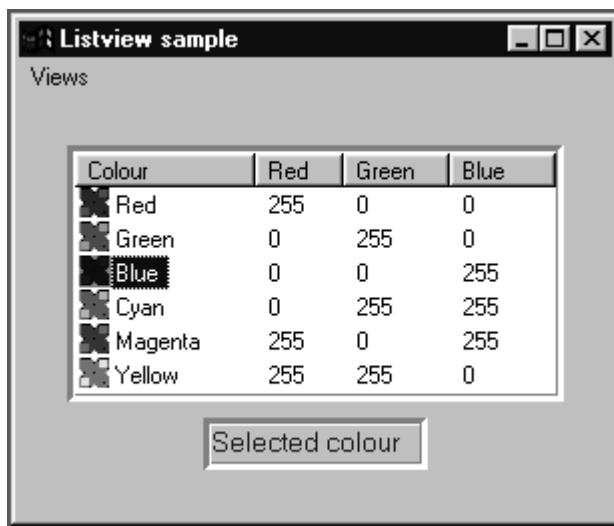
The `winio@` statement that generates the list-view is:

```
i=winio@('%`^lv[edit_labels,single_selection]&',
+ 240,120,info,N,sel,view,icons,call_back)
```

Here is an outline of the meaning of the symbols and variables used in this statement:

- | | |
|-------------------------------|--|
| grave accent | a list of icons (called <i>icons</i>) is supplied |
| caret | a call-back function (called <i>call_back</i>) is supplied |
| <code>edit_labels</code> | an option that enables the names of the colours to be edited by the user |
| <code>single_selection</code> | an option that ensures that at most one colour can be selected at a time |
| 240 | the initial width of the list-view in pixels |

120	the initial height of the list-view in pixels
info	an array of strings that defines each row; the first row defines the column headings; the first character on subsequent rows is an index (A, B, C,...) that selects an icon from the list of icons; different fields are separated by a pipe ‘ ’ symbol.
N	a parameter defining the maximum number of rows
sel	an integer array to mark the current selection
icons	a string of icon resources
call_back	a call-back function that is used when a selection is made and when the end of label editing is detected.



Tree-view - %N.Mtv[options]

The %tv format is used to construct a hierarchical list box that is analogous to the type of control sometimes used to display a directory structure. %bv is a newer format that is similar to %tv but has more options (see below). %tv takes a CHARACTER array followed by an INTEGER argument giving the size of the array. The final argument is an INTEGER argument (call it *sel*) that is used to receive the index of the chosen item. The format modifiers *N* and *M* set the width and depth of the tree-view in ‘average characters’. The control operates in a manner that is similar to a listbox except that the first two characters of each string have a special meaning. The first character must be a capital letter (‘A’ - ‘Z’) indicating the level of the item within the hierarchy. The second letter should be a ‘C’ if the item is to be displayed compactly (i.e. not

displaying any children) and ‘E’ if the item is to be displayed expanded. Consider the following list of things:

```
AFFood  
BEFruit  
CCApples  
CCPears  
BCNuts  
CCHazel nuts  
CCBrazil nuts  
AEDrink  
BCAlcoholic  
BCNon-alcoholic
```

The children of a node (together with any of their children, etc.) are placed immediately beneath the parent node. In the example shown the types of nuts would not immediately be shown because of the ‘C’ in ‘BCNuts’. Any node is expanded and contracted by simply clicking on it. Usually you would code ‘C’ for the second character of each string. This would produce an initial display showing only the top level. As nodes are expanded and contracted, the information is stored in the strings. This means that, if the strings are used to display the hierarchy a second time, the display will start from where it left off. You could even store the strings in a file so that the expanded state would be the same from one run to another.

Items with a lower case letter or a blank in the first position are ignored and never displayed. This provides a means to hide elements, or to provide space for the display to grow dynamically. If a node has no children you can use ‘E’ or ‘C’ - there is no difference.

Notice that it makes no sense for a level 2 item (say) to be followed immediately by a level 4 item. An item may not be followed immediately by its ‘grandchildren’. This condition will be detected as an error. Conversely, a level may decrease by any amount.

%tv can take a call-back function. The call-back function may use the index of the chosen item (i.e. the argument *sel*) and can also get the value of `clearwin_info@('TREEVIEW_ITEM_SELECTED')`. This value will be equal to 1 if the call-back is responding to a double click mouse event.

By default, the tree-view displays bullet marks to the left of the nodes. By using a grave accent with %tv, the bullet marks are replaced by icons. When the grave accent is used, the third character in each string is interpreted as an index (‘A’ - ‘Z’) into a list of icons. This icon list is supplied as an extra argument that consists of icon resources separated by commas (e.g. ‘ICON1,ICON2,ICON3’). You should construct the icons in such a way that the image is confined to the top left 16 x 16 corner. The remainder of the icon should be filled with the transparent colour. The Salford icon editor **SCION** can be used for this purpose.

By providing a call-back that examines the expansion indicator (character two of each string) it is possible to change the icon index to reflect whether or not the icon is expanded. If you perform a change of this sort you should pass the main array to `window_update@` to force the control to be updated.

The following example illustrates the use of tree-view controls:

```

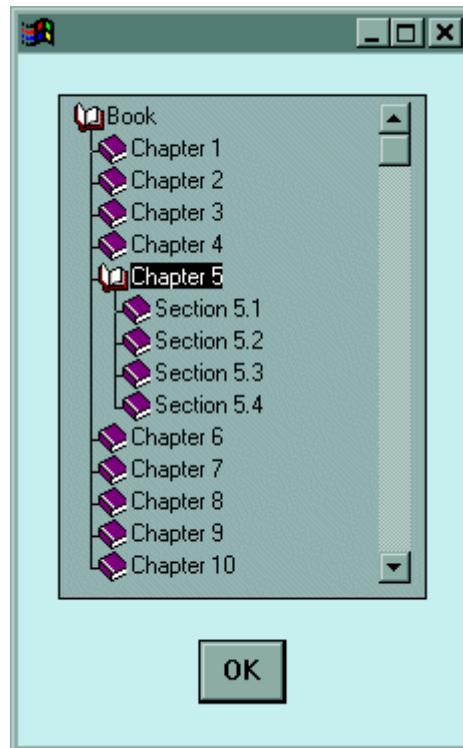
WINAPP
BLOCK DATA
CHARACTER*30 contents(56)
INTEGER item
COMMON/C/contents
COMMON/I/item
DATA contents/
+ 'ACABook',           'BCAChapter 1',      'CCASEction 1.1',
+ 'CCASEction 1.2',    'CCASEction 1.3',      'CCASEction 1.4',
...
+ 'CCASEction 11.3',   'CCASEction 11.4'/
END
C-----
      INTEGER FUNCTION test()
C-----.
c Call-back function sets the icon for
c each object according to whether it is expanded or not
C-----.
      CHARACTER*30 contents(56)
      INTEGER item
      COMMON/C/contents
      COMMON/I/item
      CHARACTER*30 string
      string=contents(item)
      IF(string(2:2).EQ.'E')THEN
          string(3:3)='B'
      ELSE
          string(3:3)='A'
      ENDIF
      CALL window_update@(contents)
      test=2
      END
c--- Main program - just call WINIO@
      EXTERNAL test

```

```
CHARACTER*30 contents(56)
INTEGER item,winio@a
COMMON/C/contents
COMMON/I/item
a=winio@(%ww%ob%pv&')
a=winio@(%`^20.15tv&,
+      contents,56,item,'CLOSED_BOOK,OPEN_BOOK',test)
a=winio@(%cb%ff%nl%cn%`bt[OK]')
END
```

Link this with a resource file containing the following:

```
CLOSED_BOOK ICON BOOK1 ICO
OPEN_BOOK   ICON BOOK2 ICO
```



Branch-view - %bv[*options*]

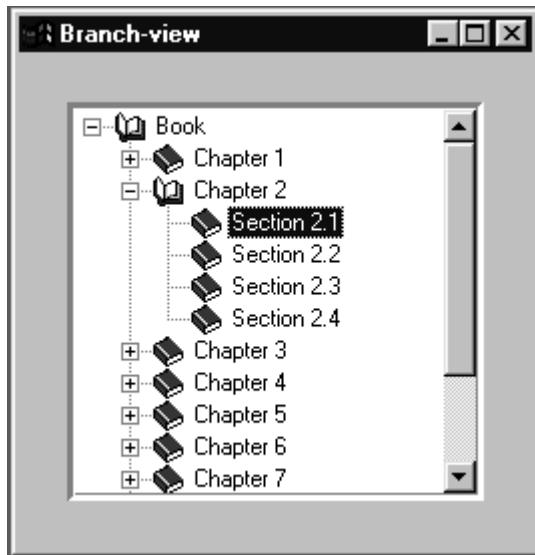
The %bv format is used to construct a hierarchical tree-view that is similar to %tv. %bv gives a control with a slightly different appearance. %bv has more options and is

simpler to use because a call-back function is no longer needed in order to change the image when expanding a node. Here is some sample code like that for %tv:

```

WINAPP
CHARACTER*30 contents(56),fmt*120
INTEGER item,i,winio@
DATA contents/
& 'AEBBook', 'BCAChapter 1', 'CCASEction 1.1',
.....
item=1
fmt='%'`bv[has_buttons,has_lines,show_selection_always,
& //edit_labels,lines_at_root,paired_bitmaps]'
i=winio@('%ww%sy[3d]%pv&')
i=winio@('%ca[Branch-view]&')
i=winio@(fmt,200,160,contents,56,item,'CLOSED,OPEN')
END
RESOURCES
CLOSED BITMAP closeb.bmp
OPEN BITMAP openb.bmp

```



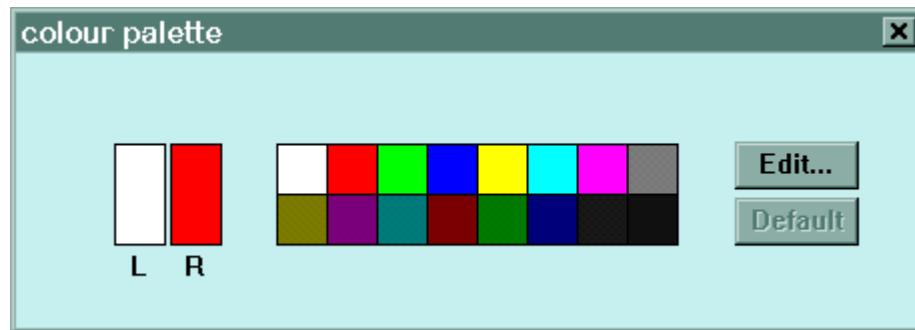
%bv uses 16x16 bitmaps. The character array called *contents* is constructed in the same way as for %tv. The option *paired_bitmaps* indicates that the list of bitmaps 'CLOSED, OPEN' is ordered such that the bitmap for an expanded node (OPEN) follows immediately after that for the corresponding collapsed node (CLOSE). For further details see page 222.

Colour palette - %cl[*options*]

The %cl format code allows the user to select from a set colours presented in a palette. The user can also change the palette. Note that the range of colours available depends on the video display and currently selected display mode.

This format takes one array argument of type INTEGER that holds the values of up to three selected colours (for, respectively, the mouse left button, the right button and the middle button - values that are not used return a shade of grey). These values are in the same format as the result of the RGB@ function.

```
WINAPP
INCLUDE <windows.ins>
INTEGER i,v(3)
i=winio@('%ca[colour palette] %cl',v)
i=winio@('%ca[colour mix result] The colour was %wd.',v(1))
END
```



The options are:

-
- | | |
|--------|--|
| b1 | Right mouse button only |
| b2 | Left and right buttons (the default) |
| b3 | Left , right and middle buttons |
| title1 | Title above button boxes |
| title2 | Title above palette |
| iconic | <i>Transparent and inverse transparent selection</i> |
-

See page 227 for further details.

8.

Bitmaps, Cursors, Icons

Bitmap - %bm[*bitmap_name*]

This is similar to the icon format %ic (see page 77) but the picture is supplied as a simple bitmap. For example we could write:

```
i=winio@('This is a bitmap: %bm[tea_bitmap]')
```

and the resource script would include a line of the form:

```
tea_bitmap BITMAP "BT1.BMP"
```

The grave accent is used to specify that the bitmap is supplied via a handle rather than a name. A suitable handle is returned by `make_bitmap@` or by the Windows API function `LoadBitmap`. If you want to use `LoadBitmap` to access one of the pre-defined bitmaps then you will need to create your own binding to `LoadBitmap`. For example:

```
INTEGER h_bitmap,OBM_DNARROW
PARAMETER (OBM_DNARROW=Z'7FF0'L)
STDCALL MyLoadBitmap 'LoadBitmapA' (VAL,VAL):INTEGER
h_bitmap=MyLoadBitmap(NULL,OBM_DNARROW)
i=winio@('%`bm',h_bitmap)
```

A call-back function can be added to a bitmap by using the caret (^) format modifier.

The %gi format is also available and works in the same way as %bm except that it uses a GIF resource created from an image file stored in GIF format. GIF files are compressed and may be animated and/or partially transparent. They therefore make ideal eye-catching features. See page 243 for further details.

Cursor - %cu[*cursor_name*]

By default the mouse cursor is represented by an arrow except in graphics regions (such as %gr) where by default it is represented by a cross. Two formats are supplied to change the mouse cursor representation. %dc sets the default cursor for the window as a whole, and %cu sets the cursor for the next control in the window. Each is followed by a *standard character string* (i.e. a string in square brackets or an @ character indicating that the required string is supplied as an argument). The string should be the name of a CURSOR resource. By using a grave accent (`), you can use constants representing the standard Windows cursors. These constants are defined in *windows.ins*.

For example:

```
a=winio@('%`dc&',CURSOR_IBeam)
a=winio@('%ob%`cu&',CURSOR_WAIT)
a=winio@('%gr&',100,100)
a=winio@('%cb%ff%nl%cn%`bt[OK]')
```

For further information see %dc below.

%cu and %dc also have a more complex form in which several alternative cursors are supplied, together with an integer used to select the one to use. The integer should be in the range from 1 to the number of different cursors supplied.

For example:

```
INTEGER k
k=1
i=winio@('%3dc[CURSOR_1][CURSOR_2][CURSOR_3]',k)
```

Whenever the cursor is in the main window, its shape will be governed by the current value of *k*. The %cu format is used in a similar way.

Default cursor - %dc[*cursor_name*]

By default, *winio@* supplies an arrow for the mouse cursor. The %dc format takes a *standard character string* giving the name of the cursor resource or, if a grave accent modifier is used, it takes an argument that is one of the following constants representing built-in Windows cursors:

CURSOR_ARROW	Standard arrow cursor
CURSOR_IBEAM	Text I-beam cursor
CURSOR_WAIT	Hourglass cursor
CURSOR_CROSS	Cross hair cursor
CURSOR_UPARROW	Vertical arrow cursor

CURSOR_SIZE	A square with a smaller square inside its lower-right corner
CURSOR_ICON	Empty icon
CURSOR_SIZENWSE	Double-pointed cursor with arrows pointing Northwest and Southeast
CURSOR_SIZENESW	Double-pointed cursor with arrows pointing Northeast and Southwest
CURSOR_SIZEWE	Double-pointed cursor with arrows pointing west and east
CURSOR_SIZENS	Double-pointed cursor with arrows pointing North and South

For example:

```
i=winio@('%`dc',CURSOR_IBeam)
```

whilst for a user defined cursor:

```
i=winio@('%dc[my_cursor]')
```

combined with a resource script that contains the line:

```
my_cursor CURSOR cursor.cur
```

The file *cursor.cur* is created using an image editor.

See the description of %cu (page 76) for a further example of how %dc is used and how several different cursors can be used for different parts of the format window.

Icon - %ic[icon_name]

This format supplies the name of an icon resource (as a *standard character string*). It is drawn at the current position in the format window.

For example:

```
WINAPP 0,0,'d28.rc'
INCLUDE <windows.ins>
i=winio@('This is an icon: %ic[win_icon]')
END
```



In a separate file called *d28.rc* we have:

```
win_icon ICON window.ico
```

The caret (^) character is used to attach a call-back function to an icon. The function is called each time the user clicks on the icon.

Icons differ in one respect from bitmaps in that they may have transparent regions that make the image appear to sit on the background in the same way that text does.

If a grave accent (`) is used, the icon is supplied via an icon handle. A suitable handle is returned by `make_icon@` (see page 377) or by the Windows API function `LoadIcon`. If you want to use `LoadIcon` to access one of the pre-defined icons then you will need to create your own binding to `LoadIcon`. For example:

```
STDCALL MyLoadIcon 'LoadIconA' (val,val):INTEGER
  INTEGER h_icon
  h_icon=MyLoadIcon(NULL,IDI_QUESTION)
  i=winio@('%`ic',h_icon)
```

Minimise icon - %mi[icon_name]

This format supplies the name of an icon resource (as a *standard character string*) to be used if the window is minimised. The %ww format must be used in order for it to be possible to minimise the window.

For example:

```
WINAPP 0,0,'d10.rc'
INCLUDE <windows.ins>
i=winio@('%wwDon''t be late for tea!%mi[tea_icon]')
END
```

The file *d10.rc* contains:

```
tea_icon ICON tea.ico
```

Standard icon - %nsisymbol

Defines a standard icon that is to be placed to the left of the block of text that follows the format code. *symbol* is one of "?", "*", "!", and "#". The icons given by these symbols are illustrated below.

For example:

```
i=winio@('%2si?%2si*%2si!%2si#&,cb_icon)
i=winio@('%2n!These are all the standard icons.&')
i=winio@('%2n!%cn%7bt[OK]')
```



The symbol is vertically centred to the left of *n* lines of text (the parameter *n* defaults to 1), and a half icon width is left blank to the right so as to leave an appropriate gap between the icon and subsequent text. Sound is added by attaching the “BEEP” callback (see page 206). Alternatively, you can call the API function **MessageBeep** before creating the format window.

The caret character (^) is used with %si in order to attach a call-back function. This function is called each time the user clicks on the icon. In the above example the ‘!’ icon uses the call-back function *cb_icon*.

Using the handle of a resource

ClearWin+ controls that take the name of a bitmap, icon, or cursor resource, can be given a string consisting of a decimal integer enclosed in curly brackets (with no embedded spaces), for example {21542}. The number should be the decimal value of a Windows handle to an object of the appropriate type. If the handle is invalid the results will be unpredictable.

For example:

```
CHARACTER*32 icon(32),handle  
icon(1)=...  
...  
icon(32)=...  
WRITE(handle,'(i10.10)') make_icon@(icon)  
i=winio@('%ic[{}//handle//{}]')
```

For %bm, %ic, %mi, %cu, and %dc it is easier to use a grave accent, supplying the handle as an integer argument . However, the above technique is useful for other format codes (e.g. %bi, %ti, %tb, %tv) and when format strings must be constructed dynamically under program control.

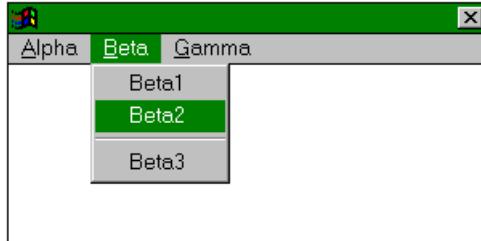
9.

Menus and accelerator keys

Menu format - %mn[*menu_specification*]

%mn is used to attach a menu to the window. This is best illustrated by an example:

```
i=winio@(%mn[&Alpha,&Beta[Beta1,Beta2,|,Beta3],&Gamma],  
+           cb1,cb2,cb3,cb4,cb5)
```



This format specifies five selectable items, three of which, Beta1, Beta2, and Beta3 are contained in a drop-down menu. A bar symbol (|) between Beta2 and Beta3 causes a separator to be drawn in the drop-down menu. The three top-level menu items (Alpha, Beta and Gamma) can be selected using accelerator keys Alt-A, Alt-B and Alt-G that are specified by the inclusion of the ampersand character (&) placed to the left of the character chosen to be the (unique) accelerator for this menu (a double ampersand (&&) is required if you want an ampersand to appear in the text). Five call-back functions are required as arguments corresponding to each selectable menu item. A call-back function is called when the corresponding menu item is selected.

Individual menu items may also be greyed (so that they are visible, but not usable). In order to grey an item, prefix its name with a tilde (~) and insert an INTEGER argument to provide a control integer. When the value of this integer is zero, the item is greyed and disabled, otherwise it is enabled. Note that it is not necessary to use a different control integer for every item. You can use one integer to control several

menu items and/or buttons. The following example illustrates a 3-item menu, in which the middle item is greyed:

```
WINAPP
INCLUDE <windows.ins>
INTEGER drinker
EXTERNAL squash_func,beer_func,tea_func
drinker=0
i=winio@(%mn[Squash]&,squash_func)
i=winio@(%mn[~Beer]&,drinker,beer_func)
i=winio@(%mn[Tea]',tea_func)
END
c---- call back function ---
INTEGER FUNCTION squash_func()
squash_func=1
END
.....
```



The hash character (#) is used to control the placement of a check mark (tick) at the front of a menu item. The corresponding control variable is set to the value 1 in order to display a check mark otherwise it is set to zero. Top level menu items cannot be checked.

As an alternative to using the ampersand (&) before a character, an accelerator key may also be associated with a menu item and its corresponding call-back function as illustrated in the following example.

```
i=winio@(%mn[File[Open¶Ctrl+F12]],open_func)
```

The paragraph mark ¶ (CHAR(20)) is used here as an alias for the tab character (CHAR(9)) and is followed by the key name. Valid key names are illustrated with the accelerator key format %ac below. The use of an accelerator key is only permitted in sub-menus and should not be used in top level menus.

In order to continue a menu in a new winio@ statement, it is necessary to match opening square brackets with closing brackets on the line to be continued. Then use an appropriate number of opening square brackets on the next line. For example:

```
i=winio@(%mn[Test,File[Open]]&,f1,f2)
i=winio@(%mn[[Save],Help]',f3,f4)
```

is equivalent to

```
i=winio@('%mn[Test,File[Open,Save],Help]',f1,f2,f3,f4)
```

Help strings are provided by inserting a question mark (%?mn). All the strings are placed in order at the end of the format description, one for each item that has a callback function. For example,

```
i=winio@('%?mn[File[New]][Start a new file]&',newfn)
i=winio@('%?mn[[Save]][Save this file]',savefn)
```

Separators also require a help string but this is not used.

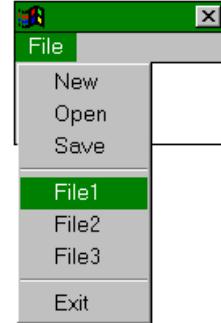
Dynamic Menus

If you wish to add or remove menu items whilst the program is running then this is possible with the aid of a handle:

```
i=winio@('%mn[&Window[*]] &', handle)
```

The asterisk (*) causes %mn to provide a handle that is passed to either `add_menu_item@` (see page 333) or `remove_menu_item@` (see page 394). Note that items are added to the end of the pop-down menu that contains the handle. You may have a number of handles for various pop-down menus.

Only one handle is required to create a menu with multiple entries. To detect which menu entry has been selected use `clearwin_string@('CURRENT_MENU_ITEM')` to obtain the text of the menu item selected.



Here is some sample code that illustrates how to add a list of files to a file menu.

```
WINAPP
INCLUDE <windows.ins>
INTEGER i,handle,ctrl,cb,cb0
CHARACTER*129 fname
EXTERNAL cb,cb0
COMMON ctrl
i=winio@('%mn[File[New]]&',cb)
i=winio@('%mn[[Open]]&', cb)
i=winio@('%mn[[Save,|]]&', cb)
i=winio@('%mn[/*]]&',handle)
i=winio@('%lw',ctrl)
DO i=1,3
    WRITE(fname,'(a,I1)') 'File',i
    CALL add_menu_item@(handle,fname,1,0,cb)
ENDDO
CALL add_menu_item@(handle,CHAR(0),0,0,cb)
```

```

        CALL add_menu_item@(handle,'Exit', 1,0,cb0)
        END
c-----
        INTEGER FUNCTION cb()
        cb=1
        END
c-----
        INTEGER FUNCTION cb0()
INCLUDE <windows.ins>
        INTEGER ctrl
        COMMON ctrl
        ctrl=0
        CALL window_update@(ctrl)
        cb0=1
        END

```

Popup menu - %pm[menu_specification]

This format code is almost identical to %mn but a popup menu is activated when the right mouse button is pressed in the format window. Please refer to %mn above for general information on how to construct a menu.

In the example below a popup menu controls the mathematical operation carried out on two values that are provided by slider controls. The divide option is greyed out whenever the slider for the denominator gives zero. An attempt to divide by zero would cause a processor exception. This has been included to show how to use the tilde (~) character.

```

WINAPP
c-----Copy to pm.ins
INCLUDE <windows.ins>
INTEGER enable,mode,c(5)
DOUBLE PRECISION r,v1,v2
COMMON enable,mode,c,r,v1,v2
c-----End of pm.ins
DOUBLE PRECISION min,max,step
EXTERNAL calc,setmode
INTEGER i,calc,setmode
DO i=1,5
    c(i)=0
ENDDO
min=0.0D0
max=100.0D0
step=1.0D0
enable=1
mode=3

```

```

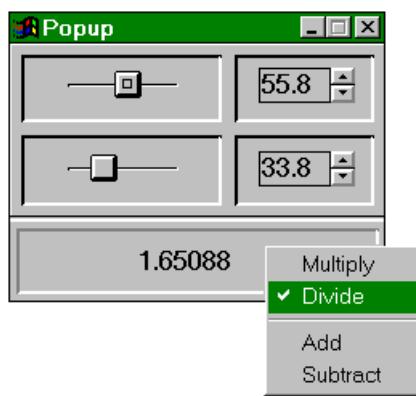
c(mode)=1
v1=0.0D0
v2=1.0D0
i=winio@('%ww[no_border,no_maxbox]%'ca[Popup]&')
i=winio@('%^2.2ob[thin_panelled]&')
i=winio@('%^10s1%cb&',v1,min,max,calc)
i=winio@(' %dy&',0.3D0)
i=winio@('%df%^4rf%cb&',step,v1,calc)
i=winio@('%^10s1%cb&',v2,min,max,calc)
i=winio@(' %dy&',0.3D0)
i=winio@('%df%^4rf%cb&',step,v2,calc)
i=winio@('%ff%ob[status,thin_panelled]&')
i=winio@('%ta%dy%`7rf%cb&',0.3D0,r)
i=winio@('%pm[#Multiply]&',           c(1),setmode)
i=winio@('%pm[~#Divide,|]&',enable,c(2),setmode)
i=winio@('%pm[#Add]&',                c(3),setmode)
i=winio@('%pm[#Subtract]',             c(4),setmode)
END
c -----
INTEGER FUNCTION setmode()
INCLUDE 'pm.ins'
CHARACTER*10 item
EXTERNAL calc
INTEGER i,calc
c(mode)=0
item=clearwin_string('CURRENT_MENU_ITEM')
IF(item.EQ.'Multiply') mode=1
IF(item.EQ.'Divide')   mode=2
IF(item.EQ.'Add')      mode=3
IF(item.EQ.'Subtract') mode=4
c(mode)=1
i=calc()
setmode=2
END
c -----
INTEGER FUNCTION calc()
INCLUDE 'pm.ins'
enable=1
IF(v2.LT.1D-6)THEN
    enable=0
    IF(mode.EQ.2) mode=5
ENDIF
IF(mode.EQ.1) r=v1*v2
IF(mode.EQ.2) r=v1/v2

```

```

IF(mode.EQ.3) r=v1+v2
IF(mode.EQ.4) r=v1-v2
IF(mode.EQ.5) r=0.0D0
CALL window_update@(r)
calc=2
END

```



System menu - %sm[menu_specification]

This format code is used to augment the system menu and is constructed in a manner similar to %mn. A grave accent modifier (`) is used in order to replace the menu rather than to add to it.

For example:

```
i=winfo@('%sm[Alpha,~Beta]',Alpha_func,Beta_ctrl,Beta_func)
```

will add the menu items Alpha and Beta to the end of the system menu. Beta will be greyed when the control variable *Beta_ctrl* has a zero value. *Alpha_func* and *Beta_func* are the associated call-back functions. If a grave accent is supplied in the form %`sm, then the two menu items will replace the system menu.

The tab character CHAR(9) (or CHAR(20)) attaches an accelerator key to a particular menu item and its associated call-back function. The syntax is the same as that for %mn above.

Enhanced menus

A ClearWin+ window can have an enhanced standard menu (%`mn) as an alternative to a standard (%mn) menu. Enhanced popup menus (%`pm) are also available. An enhanced menu has more possibilities such as bitmap entries and is more flexible for

programs that update menus on the fly or read them from configuration files. Any window can be given either a normal menu or an enhanced menu.

%`mn and %`pm each take one character argument, that describes the entire menu, and one call-back function that is used for all menu events.

Example

```

INTEGER i,winio@,cbfunc
EXTERNAL cbfunc
CHARACTER*256 fmt
fmt='[File[-bm[openbmp],-se,E&xit],Help[About]]'
i=winio@('%`mn',fmt,cbfunc)
END
!-----
INTEGER FUNCTION cbfunc()
INCLUDE <windows.ins>
CHARACTER*16 menuitem
menuitem=clearwin_string('CURRENT_MENU_ITEM')
cbfunc=1
IF(menuitem.EQ.'File~Exit') cbfunc=0
END
!-----
RESOURCES
openbmp BITMAP open.bmp

```



The character string fmt illustrates a typical enhanced menu definition:

```
fmt='[File[-bm[openbmp],-se,E&xit],Help[About]]'
```

Here the ‘&’ character performs the same accelerator key function as for standard menus. Special options preceded by a minus sign are available as follows:

- bm[xxx] Represents a bitmap (loaded from a resource named xxx). In the example, the text ‘Open’ is part of the bitmap (the font is 8pt MS Sans Serif).
- se Menu separator
- gr Subsequent item is greyed out
- ch Subsequent item is checked

- he[...] Subsequent item has help information given between square brackets.
This information can include newline characters.
- rc Subsequent item has a radio button placed in front of it.

Notice that sub-menus are specified using nested square brackets - just as they are for standard menus.

The call-back function is invoked when the user selects a menu event. This function uses `CLEARWIN_STRING@('CURRENT_MENU_ITEM')` to determine which event is required. In the given example, when the user selects 'Exit' the string 'File~Exit' is returned. Strings are separated by a tilde '~' character and any '&' characters in the menu names are left in place. In the given example, selecting 'Open' generates 'File~openbmp'. (Menu items often represent recently opened files. File names might contain tilde '~' characters, however no ambiguity will occur because such items will always appear in terminal positions in the menu tree structure.)

Newline and carriage return characters are ignored between menu items. This means that a menu structure can conveniently be defined in a text file, read in and presented as an enhanced menu.

If a menu is to be altered (whether it be to alter the grey/check status of an item, or to add/remove entries etc.) the menu is simply replaced using one of the calls:

```
CALL REPLACE_ENHANCED_MENU@(HANDLE, MENU)
CALL REPLACE_ENHANCED_POPUP_MENU@(HANDLE, MENU)
```

where `HANDLE` is an integer that is the (%hw) window handle of the parent window and `MENU` is the replacement character argument. `HANDLE` can be zero, in which case the most recently created window is updated. It is an error to call these routines for a window that does not use %`mn or %`pm.

Accelerator key format - %ac[key]

%ac is used to attach an accelerator key to a specified call-back function. The key name is provided as a *standard character string* and the call-back function is an argument associated with %ac.

For example:

```
EXTERNAL call_back_func
i=winio@('%ac[Ctrl+Alt+P]',call_back_func)
```

Other examples of valid key names are Alt+Esc, Ctrl+Shift+Del, Esc, Alt+Enter, and Ctrl+F9. See page 215 for further details. See also `add_accelerator@`.

10.

Layout and positioning

Format windows, created using `winio@`, contain text and child windows called *controls*. Text and controls are presented in the order in which they appear in the format string. ClearWin+ automatically positions the text and controls in a window and adjusts the size of the window to suit its contents. This automatic positioning provides for a standard blank border within the window. This border surrounds all the text and controls. It can be removed by using `%ww` together with its `no_border` option.

Text and controls in a format string can be separated by one or more of the following format codes.

- `%nl` gives a new line
- `%ta` advances to the next tab position
- `%ff` advances to the left margin below any existing controls
- `%cn` centres text and controls on the current line
- `%rj` right justifies text and controls on the current line

`%ta` and `%nl` can be used with an integer modifier. For example, `%2nl` gives two new lines. Further information about `%cn` and `%rj` is given below.

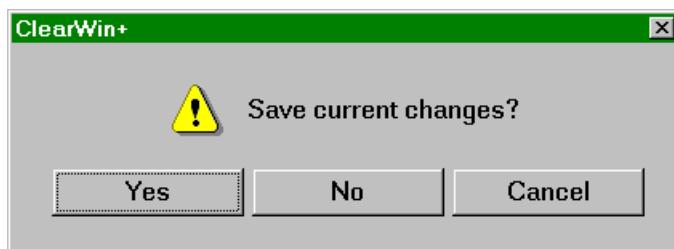
It is important to note that, if these codes appear within a `%ob` box, then the effect relates to the box rather than the window as a whole. Moreover, `%ob` boxes can be nested. As a result it is possible to exercise a fine control over the positioning of items within a window. In extreme cases, there are other mechanisms for adjusting the position of a given item. Details are given below.

Centre - `%cn`

`%cn` forces the text and controls that follow (up to the next `%nl` or `%ff`) to be centred in the window (or box). It is often used in conjunction with buttons.

For example:

```
WINAPP
INTEGER i,winio@
i=winio@(%ca[ClearWin+]%bg[grey]&)
i=winio@(%cn%si!Save current changes?%2nl&)
i=winio@(%tt[Yes] %tt[No] %tt[Cancel]')
END
```



The %cn format code can take a grave accent format modifier. This causes centring to apply to the remainder of the format window or box. Thus a whole window or a box full of centred items may be created very easily.

For example:

```
i=winio@(%`cnProgram to test prime numbers&)
i=winio@(%2nlWritten by Joe Bloggs%2nl%`bt[OK]')
```

If %cn is used within a box then it centres objects within that box.

Right justify - %rj

%rj forces text and controls that follow (up to the next %nl or %ff) to be right justified.

Tab location - %Ntl

This format code allows you to change the default tab locations. Distances are measured from the left-hand edge of the window or box using multiples of the width of an average character. N is the number of tabs to set and an argument is supplied for each location. For example, %4tl would take four integer arguments specifying the first four tab locations. A complex window layout might contain several %tl formats. Each %tl format takes effect for subsequent text and controls and cancels any previous one. If you use the grave accent modifier with %tl, then the tab locations are specified as DOUBLE PRECISION values.

Box format - %n.mob[options][title]

%ob is used to create boxes, to create a status bar, or to create a grid. Any of these can be filled with text and controls. %ob defines the top left hand corner of a rectangular box. Subsequent items are placed to the right and beneath this point until the box is closed with %cb. Every open box must be closed before the format string (plus any continuations) is complete. Boxes can be nested, one within another, in order to provide a fine control over the positioning of items in a window. A grave accent is used to indicate that the box and all items within it should have a darker (shaded) background.

%ob can be followed by options enclosed in square brackets. The options are:

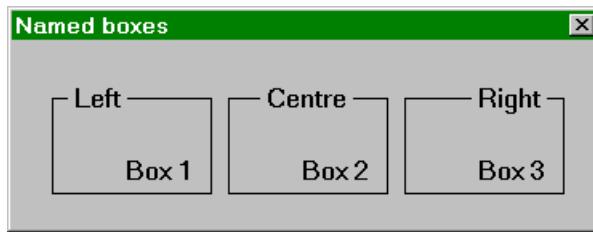
named_c	Places a given title in the top line of the box (in the centre)
named_l	Places a given title in the top line of the box (on the left).
named_r	Places a given title in the top line of the box (on the right)
no_border	Creates an invisible box which is useful for grouping controls.
panelled	Creates a three-dimensional panel.
shaded	Equivalent to %`ob.
status	The box takes the form of a status bar.
thin_panelled	An alternative to panelled.
scored	Double line, three-dimensional shade effect.
depressed	Single line, three-dimensional shade effect.
raised	Single line, three-dimensional shade effect

A status bar is a strip (with the same colour as the face of a standard button) that is placed at the bottom of a window. A status bar is typically used to display help information. The status bar format should be used at the beginning of the first (normally the only) line of status information. For example, %ob[status]%he%cb would create a status bar and place help information in it. Although status bars can be made to cover more than one line (for example by using %he in a window in which some of the help strings extend over more than one line), the result tends to look ugly. No further controls should be specified after the final %cb that closes a status bar, unless they are positioned via a %ap format code (see page 95).

A name that is to be placed in the top line of the box is provided as a *standard character string*. For example:

```
WINAPP
INTEGER i,winio@
i=winio@('%ca[Named boxes]%bg[grey]&')
i=winio@('%ob[named_l][Left]&')
```

```
i=winio@('%%nl          Box 1%cb&')
i=winio@('%%ob[named_c][Centre]&')
i=winio@('%%nl          Box 2%cb&')
i=winio@('%%ob[named_r][Right]&')
i=winio@('%%nl          Box 3%cb')
END
```



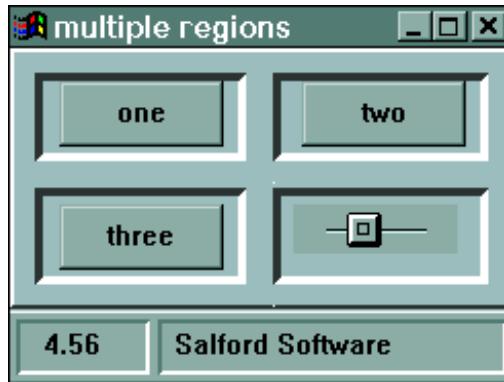
`%ob` can take integer modifiers in order to create a grid. For example, `%2.3ob` would create a grid two items across and three down. The contents of each grid component are terminated with `%cb`. Thus the above case would require six `%cb` format codes following it to close each component of the grid.

Note that, if you place `%rd` (say) in the grid, you might also choose to use `%co[no_data_border]` in order to remove the border from the `%rd` control.

Most of the `%ob` options may be used when `%ob` is used to create a grid. However, a grid box cannot be named.

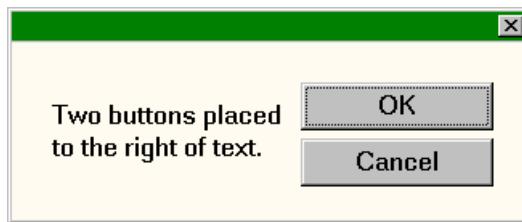
For example:

```
WINAPP
CHARACTER*17 str
DOUBLE PRECISION val,min,max
INTEGER i,winio@
val=1.0
min=1.0
max=10.0
str='Salford Software'
i=winio@('%ww[no_frame]%ca[multiple regions]&')
c---Define a 2x2 box---
i=winio@('%2.2ob[panelled] %7bt[one] %cb&')
i=winio@(' %7bt[two] %cb&')
i=winio@(' %7bt[three] %cb&')
c---Ddefine a slider ---
i=winio@('%10sl[horizontal] %cb%nl&',val,min,max)
i=winio@('%2.1ob[status,thin_panelled]&')
i=winio@('%`4rf%cb&',val)
i=winio@('%`rs%cb',str)
END
```



`%ob` can be used with the option `no_border` to divide a window into a rectangular array of sub regions that are used to position text and controls within each region. As a simple example, suppose we wish to place two buttons (one below the other) to the right of some text. Here is some code that has the desired effect.

```
i=winio@('%2.1ob[no_border]%dy&',0.5D0)
i=winio@('Two buttons placed    %nlto the right of text.%cb&')
i=winio@('%tt[OK]%nl%dy%tt[Cancel]%cb',0.3D0)
```



Notice that `%dy` has been used to provide vertical displacement. In this context, if the single argument for `%dy` were given the value `1.0D0` then the use of `%dy` would be equivalent to `%nl`.

Pivot - `%pv`

In the absence of a pivot (`%pv`) a window will not re-size (except to minimise). A pivot may only be placed before a control that will re-size in response. It is an error to attempt to pivot any other type of control. The following controls will accept the pivot (more may be added later, but many controls are intrinsically fixed in size):

<code>%bv</code>	Branch-view
<code>%cw</code>	Embedded <i>ClearWin</i> windows

%dw	Owner draw box
%eb	Edit boxes
%fr	Frame for MDI child windows
%gr	Graphics box, must have <code>metafile_resize</code> or <code>user_resize</code> property.
%ht	Hypertext
%ls	List box controls (but not dropdown boxes using the grave accent)
%lv	List-view
%ms	Multiple selection boxes
%og	OpenGL graphics box.
%tv	Tree-view
%tx	Text array
%uw	User defined window

An alternative %pv mechanism is available but not recommended. The alternative is obtained by calling `set_old_resize_mechanism@`. In this case, the %pv format code marks the position of a pivot point. Items to the right of this point move to the right as the window is widened, whilst items below the pivot point move downwards as the window is lengthened. %nr and %nd can be used in conjunction with this mechanism.

Get and set window position formats - %gp, %sp

%gp takes two integers as arguments. When the window is created these integers are set to the screen x, y co-ordinates of the point where the %gp format is used. Typically these values are used to position other windows over appropriate parts of a main window by using the %sp format. It is also used for child windows. Each time the window is moved or re-sized the x, y pair is updated.

For example:

```

WINAPP
INTEGER x,y,winio@
COMMON x,y
EXTERNAL myfunc
i=winio@('Press this button to conceal it! &')
i=winio@('%gp&',x,y)
i=winio@('%^bt[Press]',myfunc)
END
C ---
INTEGER FUNCTION myfunc()
INTEGER x,y,winio@

```

```
COMMON x,y
c--- Window will be positioned relative to the button
c--- control in the main window
i=winio@('%spHidden!',x-5,y-5)
myfunc=1
END
```



Absolute position format - %ap

This format code takes two integer arguments and positions the next control at the given (x, y) point. The units are the same as those used in the %gd format code described below. This format code is not recommended for general use. The following points should be noted if %ap is used:

- Try to place all the absolute positioned objects at the end of a format, or follow a %ap format with a %ff to enable the automatic control placement mechanism to recover.
- It is possible to place controls on top of each other using this format. This is usually undesirable unless all but one of the overlapping controls are given the 'hidden' attribute.

See also %rp and %dy.

Programmer grid format - %gd

%gd is for program development only. It overlays a grid over the window and any items within it. This enables controls to be positioned using the %ap format. The grid is marked in small intervals corresponding to the average character size of the default font, with major divisions every 10 such units. The grid starts at the left and top margins, but extends over the right and bottom margins, since this may help in the placement of further controls.

11.

Displaying text

Displaying text

Font name - %fn[*font_name*]

%fn is used to select a font for subsequent text and controls.

For example:

```
winio@('This is an example of %fn[Times]Times%sf font.')
```

This causes the word “Times” to appear in Times font with the remainder of the sentence in system font. %sf is used to reset all text attributes to the system font.

Get font - %gf

%gf takes one INTEGER variable that receives the current font handle. This handle can then be used in any Windows API function that requires a font handle.

For example:

```
INTEGER h_font  
i=winio@('%ts%gf',2.0D0,h_font)
```

Font attributes - %it, %bf, %ul

These format codes are used to begin italic, bold face, and underlined text respectively. Each code takes the grave accent modifier (`) in order to terminate the corresponding attribute.

For example:

```
i=winio@('Uses %bf bold face%`bf for emphasis')
```

%sf (system font) is used after any combination of these format codes in order to restore the system font.

System font - %sf

%sf resets the text attributes and is used after any combination of %bf, %it, %ul, %fn, and %ts.

%sf can also take a grave accent (`) but this only has effect if the program is running under Windows 95. For Windows 95, %`sf selects the scalable font that is used in menus and controls. %ts (see below) can be used after %`sf in order to set the size of the scalable font. %`sf followed by %ts with a scale factor of about 1.15 will produce readable text (see also use_windows95_font@).

The value returned by the function windows_95_functionality@ will indicate if Windows 95 is running. This function returns the value 1 if the program is running under Windows 95, otherwise it returns zero.

Font default - %fd

%fd sets the font to the default Windows 95/98 font (Windows 95/98 only).

Subscript - %sd

A font must be selected using %fn, before %sd is used. All text following %sd will be in subscript style. A grave accent (%`sd) is used to return to the previous font setting.

For example:

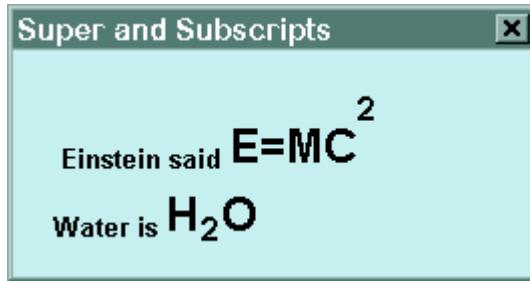
```
i=winio@('%fn[arial] Water is H%sd2%`sd0')
```

Superscript - %su

A font must be selected using %fn, before %su is used. All text following %su will be in superscript style. A grave accent (%`su) is used to return to the previous font setting.

For example:

```
i=winio@('%fn[arial]Einstein said %tsE=MC%su2%`su.',1.8D0)
```



This is the result of the examples above for %su and %sd used together. In general, scaled down fonts are undesirable as they may become unreadable in lower screen resolutions.

Text size - %ts

%ts takes one DOUBLE PRECISION argument that is used to scale the size of any text that follows until the next %ts or %sf format code. The default value of this argument is 1.0. Values of the argument in the range from 0 to 1.0 are used to scale down, whilst values greater than 1.0 are used to scale up. Note that the system font cannot be scaled, so %fn must always be used before %ts (see page 97).

For example:

```
i=winio@('%fn[Courier New]%'tsThis is larger',1.5D0)
```

Text colour - %tc

By default, the text in a format window is coloured using the ‘window text’ colour that the user can select from the Control Panel in the Program Manager. This colour is returned by `GetSysColor(COLOR_WINDOWTEXT)`. %tc is used to specify the text colour for subsequent text and controls (until another %tc overrides it). %tc takes one integer argument which is typically created using `RGB@`.

For example:

```
INTEGER RGB@
i=winio@( '%tcRed&', RGB@(255,0,0))
i=winio@( '%tcGreen&', RGB@(0,255,0))
i=winio@( '%tcBlack', RGB@(0,0,0))
```

It is often desirable to base the new colour on the default set by calling `GetSysColor` (see the description of %bg on page 127 for details of how to do this). An argument value of -1 resets the colour to the ‘window text’ colour.

Alternatively one of the standard colours (black, white, grey, red, green, blue, and yellow) can be specified by enclosing the colour name in square brackets (for example: %tc[red]).

Note that the text colour for a given control can be changed dynamically (after the window has been created) by calling the subroutine SET_CONTROL_TEXT_COLOUR@.

Variable string - %N.mst

This format code takes one character string argument and lays out the string in a field of N characters. The string is redrawn each time the window is renewed. For this reason, the size N must be specified and must be the maximum number of characters required. The number of lines m is optional and defaults to 1. For example, %20.2st provides for 20 characters on each of two lines. Use the window_update@ function (see page 26) in order to force a change to become visible. %st is similar to %`rs.

Sterling pound symbol - %pd

%pd will ensure that a pound ‘£’ symbol appears correctly on the chosen output device. Note that the ‘£’ symbol is represented differently in the OEM and ANSI character tables.

Mathematical equation - %eq[equation]

%eq is provided to enable mathematical equations to be drawn. It is followed by a *standard character string* (i.e. a string in square brackets, or an ‘@’ symbol with a string as an argument). It takes two integer arguments which are the width and depth of the equation in pixels. If the contents of the equation are not going to change, then the width and depth are set to zero, forcing the size of the region to be defined by the equation. Equations may also be included in hypertext (see Chapter 19).

Equations are strings with the three characters {} reserved. The following symbols can be included in strings:

{alpha}	α	{beta}	β
{gamma}	γ	{delta}	δ
{epsilon}	ϵ	{phi}	ϕ
{pi}	π	{zeta}	ζ
{psi}	ψ	{sigma}	σ
{theta}	θ	{kappa}	κ
{lambda}	λ	{iota}	ι

{rho}	ρ	{mu}	μ
{nu}	ν	{omega}	ω
{chi}	χ	{tau}	τ
{infinity}	∞	{plus_minus}	\pm
{ge}	\geq	{proportional}	\propto
{curly_d}	∂	{le}	\leq
{ne}	\neq	{identically_eq}	\equiv
{approximately_eq}	\approx	{such_that}	$ $
{vector_sum}	\oplus	{empty_set}	\emptyset
{set_intersection}	\cap	{set_union}	\cup
{proper_subset}	\subset	{subset}	\subseteq
{belongs_to}	\in	{not_belongs_to}	\notin
{for_all}	\forall	{there_exists}	\exists
{del}	∇	{sqrt}	$\sqrt{}$
{semicolon}	$:$		

The names of these special symbols are case sensitive, and the Greek characters also exist as upper case characters. For example, {Pi} will produce Π . In this context only the first character is upper case.

In addition, a number of more complex constructions are supplied. In the list below , the asterisk “*” stands for any equation string:

{sup *}	Superscript or exponent
{sub *}	Subscript
{divide *;*}	Division written with a horizontal bar
{sum *}	Summation without upper limit
{sum *;*}	Summation with both limits
{integral }	Indefinite integral
{integral *}	Integral with lower limit only
(integral *;*)	Full definite integral.
{symbol nnn}	Character number <i>nnn</i> (decimal) from the symbol font.

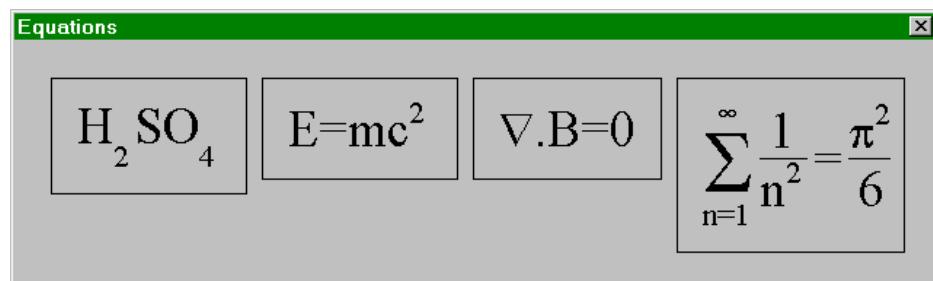
Note that the body of the sum or integral will be placed after the closing curly brace. Spaces are important in the above constructions.

Badly formed equations will normally cause a Clearwin+ error. However, if a grave accent modifier is specified (%`eq), the error will be suppressed.

Excessive nesting of constructs such as ‘sup’ or ‘sub’, that reduce the font size, may result in unreadable equations.

Here are a few well known equations and chemical formulae as examples:

```
WINAPP
INTEGER i,w,winio@
CHARACTER*80 s(4)
s(1)='H{sub 2}SO{sub 4}'
s(2)='E=mc{sup 2}'
s(3)='{del}.B=0'
s(4)=
+'{sum n=1;{infinity}}{divide 1;n{sup 2}}={divide {pi}{sup 2};6}'
i=winio@('%ca[Equations]%bg[grey]&')
DO i=1,4
    w=winio@('%ob%eq@%cb&',s(i),0,0)
ENDDO
w=winio@(' ')
END
```



Equations are constructed using symbols from the Times New Roman and Symbol fonts.

These same equation strings may be used in hypertext (%ht) embedded between <equation> and </equation>. The string between these delimiters is treated as a pure equation - no mark-up codes are recognised. So, for example, the ‘<’ symbol should be written as it is.

Edit boxes

Edit box - %eb

See Chapter 12.

Text array - %N.Mtx[options]

Text arrays provide a convenient way to display a grid of characters with ‘attributes’ to achieve an effect analogous to that achieved under DOS by writing to the screen buffer. A text array can respond to keyboard and mouse input via its call-back function by using `clearwin_info@` (see below), updating the array as necessary. The text array format takes two character strings and two integer modifiers *N*, and *M*.

For example:

```
WINAPP
CHARACTER*800 text,attr
INTEGER k,winio@
text='Test colour'
CALL char_fill@(attr,char(0))
CALL char_fill@(attr(1:4),char(1))
CALL char_fill@(attr(6:11),char(2))
k=winio@('%60.8tx&',text,attr,80L,10L)
k=winio@('%tc[blue]%ty[red]%tc[yellow]%ty[green]')
END
```

In the above example, the text array *text* is displayed in the box as an 80 x 10 array of which initially only 60 x 8 is visible. The size of the control can be varied if it is preceded by a pivot (%pv) in a variable sized window. The *attr* array contains attribute numbers stored using `CHAR(i)` where $0 \leq i \leq 255$. Zero is the default attribute (usually black on white), others are defined by subsequent `%tc[fgcolour]%ty[bgcolour]` format pairs. The first `%tc%ty` pair defines colour index 1 and so on. *fc colour* is the foreground colour of the text, and *bg colour* is the background. In the above example, the word ‘Test’ appears as blue on a red background whilst the word ‘colour’ appears as yellow on a green background. The colour can be specified as an `RGB@` value in the argument list if the colour in brackets is omitted. Changes in font attributes (underlining, etc.) are not possible with `%tx`.

`%tx` may have a call-back function that can use the following `clearwin_info@` strings:

TEXT_ARRAY_CHAR	value of key pressed
-----------------	----------------------

TEXT_ARRAY_DEPTH	holds the new value on resizing
TEXT_ARRAY_MOUSE_FLAGS	mouse button press information
TEXT_ARRAY_RESIZING	text array window dimension change
TEXT_ARRAY_WIDTH	holds the new value on resizing
TEXT_ARRAY_HEIGHT	holds the new value on resizing
TEXT_ARRAY_X	location <i>x</i> when the mouse is pressed in %tx
TEXT_ARRAY_Y	location <i>y</i> when the mouse is pressed in %tx

The TEXT_ARRAY_RESIZING parameter is set to one only in a call-back responding to a re-sizing event. The next two parameters in the table only have meaning in this context. They supply the new size of the control in average characters.

The TEXT_ARRAY_X and TEXT_ARRAY_Y parameters give the mouse position in characters from the top left corner, and are zero based. The mouse flags, that contain information about which mouse button (if any) is depressed, can be obtained by a call to `get_mouse_info@`. The option FULL_MOUSE_INPUT is specified in order to force the call-back function to be called whenever the mouse moves over the control.

TEXT_ARRAY_MOUSE_FLAGS uses bitwise flags as follows:

MK_LBUTTON	1	Left mouse button depressed
MK_LBUTTON	2	Right mouse button depressed
MK_SHIFT	4	Keyboard shift key depressed
MK_CONTROL	8	Keyboard control key depressed
MK_MBUTTON	16	Middle mouse button depressed

By default, the call-back function is called when a keyboard character is received while the control has the focus. The character is placed in the parameter TEXT_ARRAY_CHAR. If the option FULL_CHAR_INPUT is specified, the call-back function receives each keystroke using the Microsoft VK_ parameters (defined in the file *windows.ins*). The call-back is invoked for each key press and each key release. In the latter case TEXT_ARRAY_CHAR parameter has the top bit set. The parameter TEXT_ARRAY_CHAR will be zero when not in use. To respond to ALT key combinations you should use the %ac facility rather than tracking the ALT key with FULL_CHAR_INPUT. This is because messages associated with the release of the ALT key might be delayed until the next key is pressed. The option USE_TABS may be used to cause the tab key to be passed to the call-back rather than performing its normal Windows function of moving between controls.

The system font is used in a %tx control, unless %fn appears before %tx. If %fn is used then a mono-spaced font (such as Courier New) should be selected because the text array must be aligned both vertically and horizontally.

Note that the text array is not re-dimensioned if the control is re-sized. Re-sizing simply changes the region that is visible.

File selection and filter

File selection - %fs[*path*]

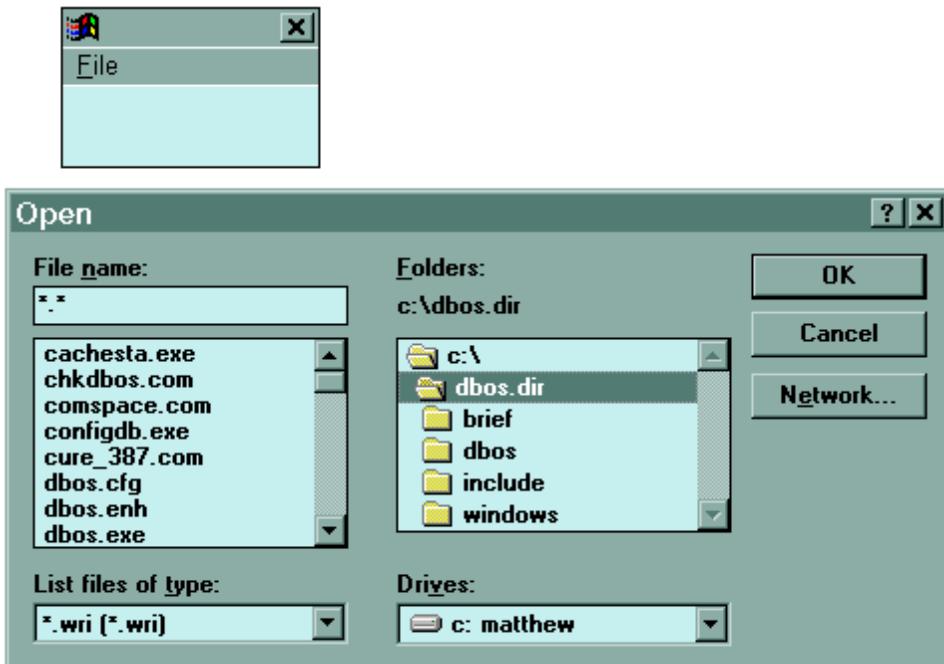
%fs together with the related %ft format code (see below), changes the default file selection used by a subsequent FILE_OPENR or FILE_OPENW call-back function (see page 205). The information is used in the standard Windows dialog displayed by these call-back functions. %fs is followed by a standard character string, that is either:

- the path name of a directory to be used rather than the current working directory (e.g. C:\TEST), or
- a complete file selection specification (e.g. C:\TEST*.EXE).

In the latter case the file filter information, displayed at the bottom of the standard 'file open' dialog, is replaced by the specified filter.

```
WINAPP
CHARACTER*129 filenm
INTEGER i,winio@
COMMON filenm
EXTERNAL disp_name
filenm=' '
i=winio@('%fs[c:\dbos.dir\*.*]&')
i=winio@('%mn[&File[&Open]]&','FILE_OPENR',filenm,disp_name)
i=winio@('%mn[[E&xit]]','EXIT')
END

INTEGER FUNCTION disp_name()
CHARACTER*129 filenm
INTEGER i,winio@
COMMON filenm
i=winio@('%ws %`bt[OK]',filenm)
disp_name=1
END
```



File filter - %ft[*name*][*filter*]

%ft is used to change the default filter for subsequent use of the FILE_OPENR and FILE_OPENW call-back functions (see page 205). The information is used in the standard Windows dialog displayed by these call-back functions. Two *standard character strings* are required after %ft. The first is the name of the filter (e.g. 'Executable files'). The second is the specification (e.g. '*.EXE').

For example:

```
i=winio@( '%ft@@', 'Executable files', '*.EXE' )
```

Alternatively,

```
i=winio@( '%ft[Executable files][*.EXE]' )
```

If the grave accent modifier is used with %ft, the filter information replaces the existing information, otherwise it is added at the front of any existing filter information. The default filter selects all files.

You cannot change the name and filters dynamically by using %fs and %ft with variable character strings. See `get_filtered_file@`.

12.

Edit box (%eb)

%N.Meb[options]

This format code provides a text editor and is by far the most complex format available. However, many applications will not require any of the embellishments detailed below. Note that %rs and %re are simpler to use but are limited to 32768 characters.

Two arguments are required, a pointer to the buffer containing the text to be edited, and an integer specifying the maximum size of the buffer. The text should be terminated by the null (CHAR(0)) character, and new lines should consist of carriage return/line feed pairs. This does not use the standard Windows edit control and as a result there is no limit (other than the total available memory) to the size of the text area. If the second argument (the length of the buffer) is set to zero, the edit box allocates its own memory for the buffer, re-sizing it as required.

%60eb denotes an edit box that displays 60 characters on one line (which is the same as %60.1eb). %60.20eb denotes an edit box that displays 60 characters by 20 lines.

The text itself has the format of a normal DOS text file (lines of text separated by a carriage return/line feed character pair) with a closing a null terminator. The edit box can also handle text lines separated by carriage return characters only, providing this is used consistently.

In order to read/write data to a file in this format the file should be opened in binary mode.

For example:

```
WINAPP  
CHARACTER*1000 buffer  
CHARACTER*80 fname  
INTEGER handle,err_code,bytes_read,nbytes,winio@  
fname='myfile'
```

```

CALL openr@(fname,handle,err_code)
CALL readf@(buffer,handle,1000,bytes_read,err_code)
CALL closef@(handle,err_code)
i=winio@('%60.20eb',buffer,1000)
CALL openw@(fname,handle,err_code)
nbytes=INDEX(buffer,CHAR(0))-1
CALL writef@(buffer,handle,nbytes,err_code)
END

```

The only null character in the string must be at the end. You must include the null character when calculating the length of a string. In the above example, the Fortran INDEX intrinsic function has been used to obtain the size of the edit buffer. An alternative method is given below. Note that openr@ etc. are Salford Fortran library routines.

<option list> is a list of items surrounded by square brackets and separated by commas. The entire (square-bracketed) list may be omitted if no options are required.

hscrollbar	Supply and control a horizontal scroll bar.
limited_hscrollbar	Like hscrollbar but the user is prevented from scrolling horizontally until all characters have disappeared.
vscrollbar	Supply and control a vertical scroll bar.
no_hscroll	Inhibit all horizontal scrolling.
no_vscroll	Inhibit all vertical scrolling.
alt_edit	Changes the action of Del, Backspace, and Enter keys so that they never split or join lines.
fixed_font	Uses a fixed font for the edit box, rather than the default variable font. (However, it is preferable to use %fn, with a monospaced font such as Courier New, before %eb in the format string.)
hook_focus	Causes the call-back to be called every time the box gains or loses the focus. Note that the over use of this option can cause annoying effects for the user and more importantly an infinite loop may be started if the result of an object gaining a focus is the shift of that focus to another object!
use_tabs	Uses tabs in the edit box in the normal way for an editor. Otherwise tabs are used to cycle through the various controls in the parent window.
no_frame	Removes the line around the edit box without removing the blank border (no_border removes both). This

no_border	option is particularly useful for edit boxes which fill a window.
read_only	Omit the blank border that by default is placed between the text and the surrounding box.
user_colours	Prevents the user from changing the text. The cursor can still be used to scroll through the text.
extended	The call-back function is invoked in such a way that the colours used for text and its background can be modified. This facility could be used, for example, to produce an editor which highlighted certain keywords. The use of this option is described further below.
undo	Allows CLEARWIN_STRING@('CALLBACK_REASON') to be used within a call-back function. For %eb the possible returns include 'DATA_ALTERATION'.
no_cursor_snap	Causes the edit box to save information to enable changes to be 'undone'. By default all changes are recorded and can be undone in reverse order. The program can also restrict the amount of undo information by calling a function. The standard call-back function 'EDIT_UNDO' is used to perform one level of undo each time it is invoked. Menus or buttons attached to this call-back will automatically become grey when no more undo information is available. The undo mechanism can reverse changes made using the keyboard/ mouse or by calling any of the edit-box related functions. However changes made by directly altering the buffer and calling WINDOW_UPDATE@ will not be recorded and cannot be reversed.
report_double_click	By default if, when you move the cursor up or down, you end up off the end of the line, the cursor will snap back to the end of the line. For some purposes, such as when you are using block marks (see below), this is inconvenient. Use the no_cursor_snap option to leave the cursor in free space on the end of the line.
report_right_click	Responds to a mouse (left) double click by calling the call-back function rather than by marking a word.
	Responds to a mouse right click by calling the call-back function rather than by looking for a pop-up menu.

The edit box maintains a set of variables in a structure (called an EDIT_INFO structure) that controls the operation of the box. Normally these variables are hidden. However, much more detailed control of the edit box can be obtained by using the grave accent edit modifier and by supplying an array of the form:

```
INTEGER info(24)
```

If more than one edit box is open, each one must have its own EDIT_INFO structure.

The following example illustrates the use of this structure, making use of the following identifiers:

Offset	Label	Description
1	h_position	Cursor horizontal character position.
2	v_position	Cursor vertical character position.
3	last_line	Total number of lines in the buffer.
4	buffer	Address of edit buffer.
5	size	Size of buffer contents (excluding null terminator).
6	max_buffer_size	Size of memory block.
7	current_position	Address of the position corresponding to eb_h_position/eb_v_position.
8	selection	Address of selected text if any.
9	n_selected	Number of selected characters.
10	vk_key	Set to VK_TAB etc. if this handles a key press.
11	vk_shift	Shift state corresponding to key.
12	active	Set to the number of the edit box (1 if only one edit box is open) when a call-back is invoked, reset afterwards. This value is used to identify which edit box is associated with the current use of a call-back function.
13	modified	Set to 1 each time the buffer is modified.
14	closing	Set when buffer is about to be closed.
15	n_chars_to_colour	Count of characters to colour.
16	text_to_colour	Address within buffer for region to colour.

17	text_colours	Address of foreground colours.
18	background_colours	Address of background colours.
19-24		Reserved for future enhancements.

Users of **FTN95** could define a TYPE corresponding to the **EDIT_INFO** array.

Although this structure is passed to most of the functions described below, you will not normally need to know its contents nor should you normally need to manipulate the buffer yourself. (You can do so if you wish, but you must remember to update the contents consistently and to make use of **window_update@** at appropriate intervals.)

The following example takes the last example above and adapts it so that a) the edit box allocates its own buffer with address *eb_buffer* and b) the *eb_modified* flag is checked before a prompt to save the edited file.

```

WINAPP
INTEGER handle,err_code,i,winio@
INTEGER info(24),
1 eb_h_position, eb_v_position, eb_last_line,
2 eb_buffer, eb_buffer_size, eb_max_buffer_size,
3 eb_current_position,eb_selection, eb_n_selected,
4 eb_vk_key, eb_vk_shift, eb_active,
5 eb_modified, eb_closing, eb_reserved(10)
EQUIVALENCE
1 (eb_h_position, info(1)), (eb_v_position, info(2)),
2 (eb_last_line, info(3)), (eb_buffer, info(4)),
3 (eb_buffer_size, info(5)), (eb_max_buffer_size,info(6)),
4 (eb_current_position,info(7)), (eb_selection, info(8)),
5 (eb_n_selected, info(9)), (eb_vk_key, info(10)),
6 (eb_vk_shift, info(11)),(eb_active, info(12)),
7 (eb_modified, info(13)),(eb_closing, info(14)),
8 (eb_reserved, info(15))
i=winio@('%60.20`eb','*',0,info)
IF(eb_modified.EQ.1)THEN
  i=winio@('%cnSave changes?%2n1%6bt[Yes] %6bt[No]')
  IF(i.EQ.1)THEN
    CALL openw@('myfile',handle,err_code)
    CALL writef@(CCORE1(eb_buffer),handle,eb_buffer_size,
+          err_code)
  ENDIF
ENDIF
ENDIF
END

```

Note the use of the Salford-supplied intrinsic function CCORE1 to return the character string at the given address.

The Salford-supplied intrinsic function LOC can be used when the address of a variable is required. For example, if an edit buffer is provided, the following fragment of code would give the position of any selected text in the window relative to the beginning of the file.

```
CHARACTER*1000 buffer
INTEGER position,info(24)
....
i=winio@('%60.20`eb',buffer,1000,info)
position=eb_selection-LOC(buffer)
```

Note also that, if an edit buffer is provided, a call-back function can be used to monitor the size of the text in the buffer and to make a call to get_storage@ if it is getting full. If the buffer overflows, the system will generate a fatal error.

%eb can take the caret (^) format modifier to indicate that a call-back function is supplied (after the address of the EDIT_INFO structure if any). This function is called whenever the contents of the box change, a key is pressed, or the cursor is moved. If the call-back function does not handle the response to control keys such as Delete, BackSpace, etc., then it should return a zero value.

This format code can also use the tilde (~) format modifier to indicate that a grey control variable is supplied. This variable should precede the call-back function (if any) in the argument list.

An edit box will almost certainly contain information that the user will not wish to lose, so it is a good idea to use the %cc format to monitor the closure of the main window.

Standard call-back functions

The following standard call-back functions (described in Chapter 20) can be used with a %eb edit box.

```
CONFIRM_EXIT
COPY, CUT, PASTE
EDIT_FILE
EDIT_FILE_OPEN
EDIT_FILE_SAVE
EDIT_FILE_SAVE_AS
EXIT
FILE_OPENR
FILE_OPENW
```

```
SELECT_ALL
```

Colouring the text and background

The grave accent is used with %tc in order to provide the text colour for a %eb edit box. %`tc then takes an integer argument which is the RGB value for the text colour. This colour can be changed under program control. Here is some sample code that illustrates the concept:

```
INTEGER clr
clr=RGB@( 255 , 255 , 0 )
...
i=winio@('%`tc&', clr)
i=winio@('%20.20eb', buffer, 10000)
...
```

In order to update the colour of the whole text in the edit box, the following code would appear in a call-back (obviously, you could use RGB@ instead of **GetSysColour**):

```
...
clr=GetSysColor(COLOR_HIGHLIGHT)
CALL window_update@(clr)
```

In order to change the colours of selected parts of the text dynamically, you should supply the `user_colours` option to %eb (use %^eb[user_colours] in a winio@ call) together with an EDIT_INFO array and a call-back function.

Extra invocations of your call-back function will be made for each line of text when it is about to be displayed. This call-back will be indicated by a non-zero value of the `n_chars_to_colour` component of the EDIT_INFO array. In this case `text_colours` and `background_colours` are the addresses of arrays of 4-byte colour values, such as those created by using RGB@. These values will have already been initialised to the default values including colour changes associated with text selection, but you are able to modify the values as required. Note that the colours may be adjusted by display drivers using a limited number of colours.

It is very important to ensure that the function which alters the display colours does not itself perform any screen I/O. At the point when the colouring request is made, Windows has issued a WM_PAINT message, and Windows will become unstable if this rule is not followed.

Since the colours are stored as 4-byte integers, this leaves the top byte of each colour otherwise unused (it is set to zero by RGB@). With %eb, you can set the following bits in the foreground colour:

Z'40000000'	Force underline.
Z'20000000'	Force italic.

```
Z'10000000'      Force bold.  
Z'08000000'      Hide the character.
```

These bits only add attributes, so if the basic font for the edit box has been set bold (say) the bold bit will have no effect. The ‘hide character’ bit removes the corresponding character (and the space it occupies) from the screen. This is useful for implementing magic text sequences to achieve special effects. For example, suppose you were writing a program to display some text containing URL’s. If you surrounded the URL’s with some special character sequences to identify them:

```
$$$http://www.salford.ac.uk/ssl$$$
```

your program could recognise these sequences and make them invisible while changing the colour/attributes of the characters comprising the URL.

The following functions have been added to help manipulate edit boxes:

Text search:

```
INTEGER FIND_EDIT_STRING@(E,STR,BACKWARDS,WHOLE_WORD, &  
                         CASE_SENSITIVE)  
INTEGER E(24)           ! EDIT_INFO block  
CHARACTER*(*) STR      ! String to seek  
LOGICAL BACKWARDS      ! True to perform backward search  
LOGICAL WHOLE_WORD     ! Seek a whole word  
LOGICAL CASE_SENSITIVE ! Case must match  
LOGICAL SELECT_RESULT   ! Leave result selected
```

This function searches for the specified string and returns 1 if successful and zero otherwise. Typically you would supply a search menu item on a window containing an edit box. In the call-back function associated with that menu you would display a dialog box to obtain the string (%rs) and some of the various flags (%rb) and then call find_edit_string@. Although designed for %eb, this function may also be useful when searching for a sub-string in a string obtained from some other source.

Text search/replace:

```
INTEGER FIND_REPLACE_EDIT_STRING@(E,STR,REPLACEMENT, &  
                                 BACKWARDS,WHOLE_WORD,CASE_SENSITIVE,REPLACE_ALL)  
INTEGER E(24)           ! EDIT_INFO block  
CHARACTER*(*) STR      ! String to search for  
CHARACTER*(*) REPLACE   ! Replacement string  
LOGICAL BACKWARDS      ! True to perform backward search  
LOGICAL WHOLE_WORD     ! Seek a whole word  
LOGICAL CASE_SENSITIVE ! Case must match
```

```
LOGICAL REPLACE_ALL
```

```
    ! Continue until the end or beginning of file
```

This function searches for the specified string and replaces it with the replacement string. The function returns a count of the number of replacements made.

Buffer positioning:

```
INTEGER SCROLL_EDIT_BUFFER@(E,HPOS,VPOS)
INTEGER E(24) ! EDIT_INFO block
INTEGER HPOS
INTEGER VPOS
```

This function positions the buffer at a given point and returns 1 if successful and zero otherwise.

```
INTEGER FUNCTION ADVANCE_EDIT_BUFFER@(E,N)
INTEGER E(24) ! EDIT_INFO block
INTEGER N ! Count of characters to move
```

This routine moves the pointer through an edit buffer character by character. The carriage return/line feed character pair are treated as one. N can be set to a negative value in order to move backwards through the buffer. The number returned is the number of character positions moved. Thus unless the start or end of the buffer was encountered, the value of N will be returned unchanged.

Undo operations:

```
SUBROUTINE UNDO_EDIT_CHANGE@(E)
INTEGER E(24) ! EDIT_INFO block
```

Performs one level of undo operation. A single undo may reverse the effects of several function calls. Direct user interactions with the edit box are tagged so that one user command is reversed at each step (even though it may be implemented in several steps). Other operations should be tagged as required using next_edit_operation@ described next.

```
SUBROUTINE NEXT_EDIT_OPERATION@(E)
INTEGER E(24) ! EDIT_INFO block
```

Calling this function indicates to the edit box that future changes are to be considered to belong to a new user operation. Typically you would call this routine at the head of a menu call-back function that was about to change the edit buffer in some way. If you perform a long series of changes without calling this function, they will all be reversed as a block if the user performs an undo operation.

Opening a file:

```
INTEGER FUNCTION OPEN_EDIT_FILE@(E,FILE)
INTEGER E(24)      ! EDIT_INFO block
CHARACTER*(*) FILE ! Name of file to open
```

This function discards any existing information in the edit buffer and reads the contents of the specified text file into the buffer. It returns 1 on success and zero if it was unable to open the file. To use this function successfully you are advised to specify a zero buffer size with %eb to obtain a dynamically allocated buffer.

Inserting text:

```
SUBROUTINE INSERT_EDIT_STRING@(E,STR)
INTEGER E(24)      ! EDIT_INFO block
CHARACTER*(*) STR ! String to insert
```

Inserts the given string (including any trailing blanks) into the buffer at the current point and advances the current point to beyond the insertion. To insert a new line into data containing carriage return-line feeds (i.e. the binary image of a file) you should insert carriage return (decimal 13) followed by line feed (decimal 10).

Moving around the display:

```
SUBROUTINE EDIT_MOVE_HOME@(E)
INTEGER E(24) ! EDIT_INFO block
```

Moves the cursor to the start of the line (as if the HOME key had been pressed).

```
SUBROUTINE EDIT_MOVE_END@(E)
INTEGER E(24) ! EDIT_INFO block
```

Moves the cursor to the end of the line (as if the END key had been pressed).

```
SUBROUTINE EDIT_MOVE_TOF@(E)
INTEGER E(24) ! EDIT_INFO block
```

Moves the cursor to the top of the file (as if CTRL-HOME had been pressed).

```
SUBROUTINE EDIT_MOVE_BOF@(E)
INTEGER E(24) ! EDIT_INFO block
```

Moves the cursor to the bottom of the file (as if CTRL-END had been pressed).

Deleting text:

```
INTEGER FUNCTION EDIT_DELETE_LINES@(E,N)
INTEGER E(24)      ! EDIT_INFO block
INTEGER N          ! No of lines to delete
```

This function deletes N lines starting at the current position. It returns the number of lines actually deleted, which may be less than N if the end of the buffer is encountered.

Marking blocks:

```
SUBROUTINE EDIT_OPEN_LINE_MARK@(E)
INTEGER E(24) ! EDIT_INFO block
```

This starts a line-mode selection. Initially the selection covers the current line, but it can be extended upwards or downwards by moving the cursor key. The selected area can be cut/paste in the same way as an ordinary character selection.

```
SUBROUTINE EDIT_OPEN_BLOCK_MARK@(E)
INTEGER E(24) ! EDIT_INFO block
```

This starts a block (or column) selection. To use this you probably want to use the no_cursor_snap option (see above) since the user will be creating and manipulating rectangular regions on the screen.

```
SUBROUTINE EDIT_CLOSE_MARK@(E)
INTEGER E(24) ! EDIT_INFO block
```

This will close a selection of either type.

New CLEARWIN_INFO@ parameters

An extra clearwin_info@ parameter has been added to help programs with several edit boxes.

'ACTIVE_EDIT_BOX' returns the address of the EDIT_INFO array of the currently active edit box. Notice that this edit box might not have focus if, for example, the user had just moved to a menu item.

'EDIT_KEY' returns the ASCII value of the last key pressed. This information can be obtained in response to a return of 'DATA_ALTERATION' from clearwin_string@('CALLBACK_REASON'). In order to activate this feature you must use the %eb[extended]. The call-back is called after the change has been made so you will need to use one of the above editing functions if you want to undo the key press.

Clipboard cut/paste

The address of an EDIT_INFO structure associated with an edit box can also be passed to the following functions:

```
INTEGER FUNCTION EDIT_CLIPBOARD_COPY@(E)
INTEGER FUNCTION EDIT_CLIPBOARD_CUT@(E)
INTEGER FUNCTION EDIT_CLIPBOARD_PASTE@(E)
INTEGER FUNCTION EDIT_REFRESH@(E)
INTEGER E
```

The first three functions perform the indicated clipboard operation using the current cursor position and current selected text in the edit box. They would typically be called from an edit menu call-back. The last function informs the edit control that the contents of the buffer have been replaced. In each case the function returns one if successful, and zero otherwise.

Scroll position

The address of an EDIT_INFO structure associated with an edit box can also be used to get the cursor position of the top left corner of the window and to scroll vertically to make a given line appear as the first line.

```
SUBROUTINE EDIT_WINDOW_TOP_LEFT@(E,HPOS,VPOS)
SUBROUTINE SET_EDIT_FIRST_LINE@(E,VPOS)
INTEGER E(24) ! EDIT_INFO block
INTEGER HPOS,VPOS
```

In the first of these two routines, a return value of (-1) indicates an error condition.

13.

Help

The question mark “?” is used as a format modifier in order to signify that a help string is supplied. By default, the help string is displayed at the bottom of the window. The help string is either surrounded by square brackets, or an “@” character is placed in the format string to indicate that the help string is supplied as an extra argument. In addition to “?”, %bh, %th and %he are used to change the manner in which a help string is presented.

Bubble and tooltip help - %bh and %th

%bh takes an INTEGER argument that is a control variable. When this variable is non-zero, help text is supplied in the form of a “bubble” whenever the mouse cursor lies over the control in question. To enable users to switch such help *on* and *off*, simply supply a button with a call-back function that changes the integer control variable.

For example.

```
INTEGER switch,i
COMMON switch
EXTERNAL toggle
CHARACTER*60 hlp
hlp='Toggle bubble help On/Off'
c--- Define a button with help attach the help ---
c--- string with an @ symbol ---
switch=0
i=winio@(%^?bt[Start]@%bh',toggle,hlp,switch)
END

INTEGER FUNCTION toggle()
INTEGER switch
COMMON switch
switch=1-switch
```

```
toggle=2
END
```



A grave accent can be added to %bh. This has the effect of generating a short delay before the help bubble appears.

%th is used as an alternative to %`bh. %th provides a help bubble in the form of a “tooltip”.

Help format - %n.mhe

By default, help information (associated with the “?” format modifier) is displayed at the bottom of the window. When %he is used, the point at which %he appears in a format string specifies the location of the help information. The information could (for example) be placed inside a box.

In the form %he (without the integer modifiers), the area for the help string will be sized to fit the largest help string so far encountered. If further help strings are supplied after %he, their lengths should not exceed this size. In most cases help boxes are best placed at the bottom of a window. However, if necessary, an earlier help string could be padded with spaces in order to increase its size.

%20he fixes the size of the area to 20 standard characters on one line, whilst %20.3he has the same width but occupies 3 lines.

In the following example the help information is placed in a box below the %rd edit box.

```
INTEGER n_ch
CHARACTER*32 str,hlp
n_ch=0
str='No of children:'
hlp='How many children have you got?'
i=winio@('%21st%?rd@%2nl&',str,n_ch)
i=winio@('%ob%he%cb',hlp)
```

14.

Window attributes

Caption - %ca[*title*]

%ca defines the title for a format window. The title is supplied as a *standard character string*.

For example:

```
i=winio@('%ca[Error]')
```



The caption will be re-drawn whenever the window is refreshed. This means that if the title is supplied in the argument list (signified by the "@" character in the format string) then the window title can be changed dynamically by changing the contents of the string and calling `window_update@`.

Window handle - %hw

%hw is used to return the window handle of the format window being created. This will be the same as the handle returned by a call to `clearwin_info@('LATEST_FORMATTED_WINDOW')`. %hw takes one integer argument for the return value.

Control handle - %lc

%lc is similar to %hw and provides the window handle of the control immediately preceding %lc in the format string.

Leave window open - %lw[option]

By default, `winio@` does not return until the window that it creates is closed. Occasionally it is useful to create a window that remains open while the program takes other actions. `%lw` causes `winio@` to return as soon as it has created the window (the return value of `winio@` is not used in this case). The format takes one INTEGER argument that can be used to close the window. This integer, known as the control variable, is set to -1 while the window is still active. It is very important that the control variable used to receive the result does not cease to exist before the window closes. Hence it is necessary to use a variable that is either global (appears in the main program, a COMMON block or MODULE) or static (appears in a SAVE or DATA statement).

Note: Although the control variable is initialised by the call to `winio@`, you may choose to give it a value before the call in order to avoid a compiler warning.

`%lw` is also used to create a child window that is to be inserted within another window that includes `%ch`. The grave accent is used in this context. In this case the window is not displayed until the control variable is passed to the window containing `%ch`. See `%ch` below for an example of this procedure. A child window may be used only once. If it is not used it will be destroyed when the program terminates.

A window that has been left open by means of `%lw` can be closed by setting its control variable to a non-negative value and by passing the address of the control variable to `window_update@`.

For example:

```
INTEGER control
i=winio@('Processing - please wait%lw',control)
CALL do_something_complex()
control=0
CALL window_update@(control)
```

It is possible to use `%lw` in conjunction with graphics drawing using `%gr`, `%og` or `%dw`. This provides a means of drawing to a region after it has been created. However, it is recommended that code to draw to a graphics region be placed in a call-back function attached to `%sc`.

`%lw` can take the option owned (`%lw[owned]`). This option defines certain properties of the window in relation to its parent. The “owned” properties are those of a window that does not use `%lw`, namely that the parent cannot overlay the child and, if the parent is closed, the child automatically closes. For these properties to be meaningful the parent must be created using `%ww`. Also `%lw[owned]` must be used in the definition of a child window that appears in a call-back function of the parent.

Start-up call-back - %sc

%sc takes one call-back function which is called only once when the window is first displayed. It has several uses, one of which could be to display a start-up message.

Child window - %ch

%ch inserts a child window created by %`lw in a previous call to `winio@`. The format takes a single INTEGER argument that is the control variable of the child window.

Child windows are automatically restored on top of their parent and are constrained to lie within it. If they have a caption (as they will by default) they can be moved by the user. Child windows created in this way are fixed in size and location (relative to the parent window). Here is a simple example:

```
INTEGER control_var  
i=winio@('%ww[no_frame]%'`lw',control_var)  
i=winio@('This is a child window%2n1%ch',control_var)
```

Control variable - %cv

%cv is used to establish a control variable for a window that does not use %lw. It takes an INTEGER control variable as an argument. This is used to enable another window to attach itself as a child using the %aw format.

Create MDI frame - %fr

%fr is used in a parent window to define a frame into which child windows are placed. These windows can be moved and re-sized within this MDI (multi-document interface) frame. The format takes two integer arguments that define the width and height of the frame in pixels. A MDI frame can be preceded by the pivot (%pv), in which case it will expand as the window expands. An example using %fr appears with %aw below.

A grave accent (`) can be added to %fr to cause it to maintain a handle identifying the currently selected child window. An extra integer argument is supplied to hold the returned handle. The integer will be zero if there is no child window. When a child window is created, the handle is obtained by using %hw (see page 121). A comparison may then be made between the handle updated by %fr and those previously returned by %hw.

Attach window - %aw

%aw attaches the window as an MDI child of a parent. The parent must contain an MDI frame (%fr). %aw takes an INTEGER argument that is a control variable attached to the parent window (the parent must already exist). If the parent window

does not use %lw, and therefore does not have a control variable, use %cv (see above) in the parent window. In general the child window should have a caption so that it can be moved, maximised, minimised, etc. within the frame. This means that you should not use the %ww no_caption option in this context.

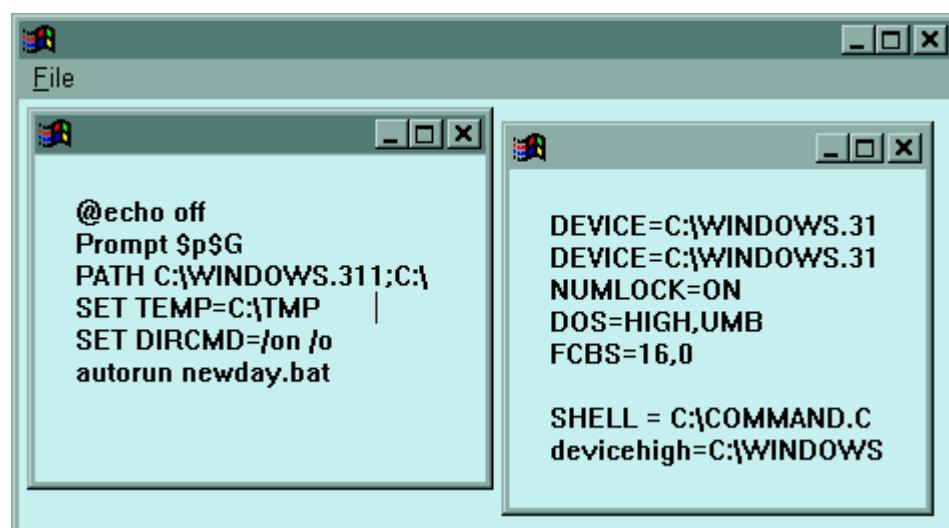
The following code provides the basis of a simple “multipad” editor using %fr and %aw.

```

WINAPP
INTEGER ctrl,wini@
EXTERNAL open_func
CHARACTER*128 fname
COMMON ctrl
fname='*.for'
i=wini@('%ww[no_border]&')
i=wini@('%mn[&File[&Open]]&',
+ 'FILE_OPENR[Open]',fname,'+',open_func,'EDIT_FILE',fname)
i=wini@('%mn[[E&xit]]&','EXIT')
i=wini@('%pv%fr&',400L,300L)
i=wini@('%lw',ctrl)
END

INTEGER FUNCTION open_func()
COMMON ctrl
INTEGER ctrl,wini@
i=wini@('%pv%aw&',ctrl)
i=wini@('%20.8eb[no_border]','*',0L)
open_func=1
END

```



Details of the standard call-back functions FILE_OPENR, +, EDIT_FILE, and EXIT are given in Chapter 20.

Drag and drop - %dr

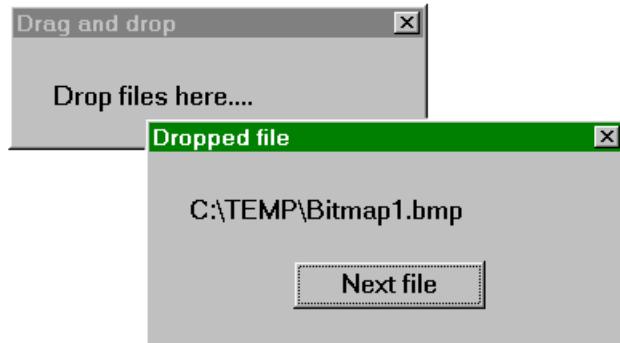
%dr takes a call-back function that is called when a file is dropped on to a window. If a collection of files is dropped, the call-back is called for each file in turn. This can be achieved, for example, by the following actions:

- Open the File Manager or Explorer and select one or more files.
- Drag the files to the format window containing %dr.
- Use clearwin_string@('DROPPED_FILE') in the call-back associated with %dr.

For example, a text editor could use this feature by opening the dropped file for editing.

The following code illustrates how %dr works:

```
WINAPP
INTEGER i,winio@
EXTERNAL drop_func
i=winio@('%ca[Drag and drop]%bg[grey]&')
i=winio@('Drop files here....&')
i=winio@('%dr',drop_func)
END
c -----
INTEGER FUNCTION drop_func()
INCLUDE <clearwin.ins>
CHARACTER*129 filename
INTEGER i
filename=clearwin_string@('DROPPED_FILE')
i=winio@('%ca[Dropped file]%bg[grey]&')
i=winio@('%`20rs&',filename)
i=winio@('%2n1%cn%tt[Next file]')
drop_func=2
END
```



Set window position - %sp

This format code is used to initialise the position of the window on the screen. Integer arguments *x* and *y* are used to specify the position of the top left-hand corner of the window in screen co-ordinates.

For example:

```
i=winio@('%si?%spWhy is this stuck in the corner',0L,0L)
```

If more than one position format is used, the positions are added vectorially. This is particularly useful in conjunction with the get window position format %gp.

Set window size - %sz

%sz takes two INTEGER variables (*w,d*) in the form:

```
INTEGER w,h  
i=winio@('%sz',w,h)
```

w and *h* are usually given zero values in the program before the first call to winio@. Zero values are ignored by winio@. (*w,h*) subsequently hold the current pixel width and height of the window. These values will change when the window is sized. When the window is closed, the latest values can be stored (in a configuration file for example) and used the next time the same window is opened. In this way you can recreate the window size that was last used. A maximised window is configured in the same way because specific values are used to represent this state.

Window style - %ww[options]

This format code causes the resultant window to look like an application window, rather than a dialog window. The window can be maximised, etc.. %ww can be followed by an option list enclosed in square brackets (use an empty list if options are

not used and the next character in the format string is an open square bracket '['). The following options are available:

<code>casts_shadow</code>	This option produces a window that casts a shadow. It is most effective when used in conjunction with <code>no_frame</code> and <code>no_caption</code> .
<code>no_caption</code>	Omit the caption from the window.
<code>no_maxminbox</code>	Omit the maximise and minimise boxes from the window.
<code>no_maxbox</code>	Omit the maximise box.
<code>no_minbox</code>	Omit the minimise box.
<code>no_sysmenu</code>	Omit the box at the top left that is used to produce the system menu.
<code>no_border</code>	Omit the blank border that normally surrounds the window contents, but leave the frame itself. Using this option you can, for example, force a tool bar (textual or bit-mapped) to be placed immediately beneath the menu.
<code>no_frame</code>	Omit the window frame and the blank border that normally surrounds the window contents. This option is particularly useful with certain types of child window.
<code>no_edge</code>	This option is like <code>no_frame</code> except that it leaves the default border (an empty space around the edge of the window).
<code>topmost</code>	Forces the window to stay on top, even if another application gets control. This is sometimes useful when it is necessary to execute another application without the user losing sight of the current window.
<code>maximise</code> (or <code>maximize</code>)	Displays the window as a maximised window.
<code>minimise</code> (or <code>minimize</code>)	Displays the window as a icon.

Background colour - `%bg[colour]`

`%bg` is used with or without the grave accent modifier. Without the modifier, `%bg` changes the background colour of the main window. With the modifier, `%`bg` changes the background colour of the next (and only the next) control (used for example with `%rd`, `%rf`, `%rs`, `%ed`, `%ls`). `%`bg` is not used for buttons. For buttons use `%bc`.

By default a main window uses the background colour that the user can select from the Control Panel in the Program Manager. This colour is returned by `GetSysColor(COLOR_WINDOW)`.

The new colour can be supplied as one of the standard colours: black, white, grey, red, green, blue, and yellow (for example: `%bg[yellow]`). Alternatively `%bg` takes one integer argument that has a value given by `RGB@`.

For example:

```
i=winio@('%bg',RGB@(0,255,255))
```

It is often desirable to base the new colour on the default by calling `GetSysColor`. In other words, you could create an alternative shade of the default background colour by unpacking the integer returned by `GetSysColor` and by modifying one or more of the (red, green, blue) components. For example:

```
INTEGER color,red,green,blue
color = GetSysColor(COLOR_WINDOW)
red   = AND(color,255)
green = AND(RS(color,8), 255)
blue  = AND(RS(color,16),255)
color = RGB@(red,green,blue+50)
i     = winio@('%bg',color)
```

This will deepen the blue component of the background colour (the new value is assumed to be less than 256). RS and AND are Salford-supplied intrinsic functions (in Fortran 90/95 you can use the standard intrinsics ISHFT and IAND).

ClearWin window - %N.Mcw[options]

Using `%cw` you can embed a *ClearWin* window (as described in Chapter 23) in a format window. The window is created *N* characters wide and *M* deep but this can be changed by the user if the pivot format (`%pv`) is included. `%cw` takes one argument that is the Fortran unit number that you wish to associate with the window. Alternatively, if you pass zero, the *ClearWin* window is created as the default stream. By default the window has no caption and is not scrollable. The options `vscroll` and/or `hscroll` are used to provide vertical and horizontal scroll bars. If a caption, etc. is required (for example to make the window movable), then `%cw` must be embedded in a child window which is itself embedded in the main window.

A grave accent (`) is used with `%cw` to obtain a handle. This handle is returned via an integer argument. The handle is used in calls to associated routines e.g. `set_max_lines@`.

Although a *ClearWin* window can be used for both input and output, it is normally wise to perform input using other format codes and to use *ClearWin* windows for scrolling results. *ClearWin* windows are also very useful while debugging a program.

The call-back functions CUT, COPY and PASTE will work from within a window defined by %cw.

Property sheet - %Nps

Property sheets represent a way of presenting two or more ‘sheets’ of data in a form that resembles a card index. Each sheet is set up as a separate window using %sh. %sh produces a child window that is hidden from view until connected to a property sheet control using %ps. The property sheet can also have a call-back function that is called each time the visible sheet is changed. A SHEET_N0 parameter is provided (see clearwin_info@) so that it is possible to determine the sheet that is topmost. When the sheet is first displayed, the SHEET_N0 is set to 1 and the call-back is also called. A grave accent modifier is supplied to %ps in order provide a handle. The handle is used set the initial sheet (at a value other than one) and to change the sheet number under program (rather than user) control (see page 265 for further details). This handle must be given the SAVE attribute or be a global variable (e.g. stored in a module or common block).

If %ca is used to provide a caption for the property sheet, then the character ‘&’ will have the effect of generating an accelerator key (this will only work for the %ps format).

If an individual sheet is closed (e.g. by placing a button without a callback function in the sheet) then the parent window will close.

```

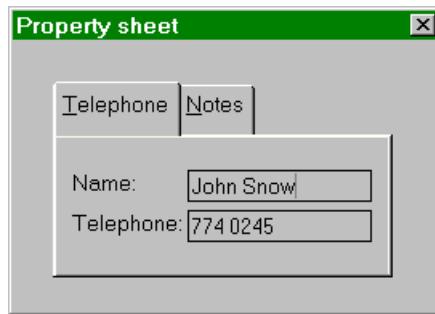
WINAPP
INTEGER h1,h2,i,winio@
CHARACTER*120 notes
CHARACTER*20 name,telephone
notes=' '
name=' '
telephone=' '
--First property sheet
i=winio@('%sh&',h1)
i=winio@('%fn[MS Sans Serif]%ts&',0.90D0)
i=winio@('%ca[&Telephone]%dy&',0.6D0)
i=winio@('%2.2ob[no_border]&')
i=winio@('Name:%cb&')
i=winio@('%12rs%cb&',name)
i=winio@('Telephone: %cb&')
i=winio@('%12rs%cb',telephone)
--Second property sheet

```

```

i=winio@( '%sh&', h2)
i=winio@( '%ca[&Notes]&' )
i=winio@( '%fn[MS Sans Serif]%ts&', 0.90D0)
i=winio@( '%20.3re', notes)
c---Display the combined sheets
i=winio@( '%ca[Property sheet]%bg[grey]&' )
i=winio@( '%2ps', h1, h2)
END

```



Property sheet child - %sh

This format code is used with %ps. %sh is placed in a format string for a child window and an argument is provided so that a handle to the window is returned. This handle is an input parameter for %ps. See %ps above for an example.

Closure control format - %cc

Sometimes, especially when an edit box is displayed, it is desirable to control the closure of a window. %cc takes a call-back function as its argument. This function is called before the window closes (for whatever reason). If it returns a positive value the window is not closed. If a grave is used (%`cc) then the associated call-back function is called after the window is closed. A single window could use both %cc and %`cc.

Delayed auto recall - %dl

%dl allows a given call-back function to be called at regular intervals. It takes a DOUBLE PRECISION argument that is the number of seconds to wait between calls followed by the given call-back.

The following code produces a window that displays its message for two seconds before closing:

```
i=winio@( '%dl%si!Testing', 2.0D0, 'EXIT' )
```

Note: Timing messages are only received while the program is idle.

Wallpaper - %wp[*bitmap*]

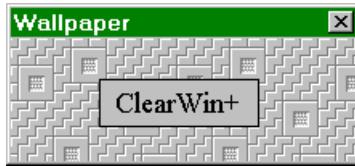
%wp takes the name of a bitmap resource. *bitmap* is a *standard character string*. %wp “wallpapers” the window with the bitmap before the text and controls are drawn.

For example:

```
WINAPP 0,0,'chitz.rc'
INTEGER i,winio@
i=winio@('%ca[Wallpaper]%wp[chitz]&')
i=winio@('%bg[grey]%fn[Times New Roman]&')
i=winio@('%cn%obClearWin+%cb')
END
```

where the resource script *chitz.rc* contains:

```
chitz BITMAP c:\windows\chitz.bmp
```



%wp is most effective in windows that contain only graphical objects. If the bitmap is smaller than the dimensions of the window, it will be repeated horizontally and vertically to fill the space. Therefore, your bitmap should either be sufficiently large, or it should contain a pattern that is designed to repeat, like the standard Windows wallpaper bitmaps. A grave accent (`) is added to %wp in order to ensure that the window will be expanded to hold at least one copy of the bitmap.

User window - %uw[*class_name*]

%uw enables existing Windows code to be interfaced with ClearWin+. Given a registered Windows class (that will have its own Windows procedure), a window can be embedded in a *format* window by using %uw. %uw is followed by a *standard character string* giving the name of the class. It also takes the following arguments:

- 1) the INTEGER width of the child in pixels,
- 2) the INTEGER height of the child in pixels,
- 3) the Window style (This will be OR-ed with WS_CHILD. In general, no border or caption will be specified, as these will normally be applied to the main window e.g. using %ca.)
- 4) the extended window style (often zero),
- 5) an INTEGER to receive the handle for the resultant child window.

Screen saver - %sv

By using %sv it is possible to create a screen saver. A format window that includes %sv will close as soon as it receives mouse or keyboard input. The executable should be renamed to a <filename>.SCR file, placed in the Windows directory and then selected as the current screen saver from the Control Panel. Apart from its obvious use, a screen saver can be used to perform a lengthy calculation. To ensure the calculation is continued from where it was interrupted, the %cc (window closure format) should be used to trap the closure and perform file I/O.

For example:

```

WINAPP
INCLUDE <windows.ins>
INTEGER xres,yres,g_handle,r,g,b,i
COMMON xres,yres,g_handle,r,g,b
EXTERNAL redraw_screen
DOUBLE PRECISION secds
secds=1.0
r=20
g=64
b=11
xres=clearwin_info@('screen_width')
yres=clearwin_info@('screen_depth')
c --- %bg background colour
i=winio@('%sv%bg[black]&')
c --- Define a window that takes the Whole display area
i=winio@('%ww[no_caption,no_maxminbox,no_sysmenu,'
+ // 'no_frame,no_border,topmost]&')
i=winio@('%`gr[rgb_colours,black]&',xres,yres,g_handle)
c --- Use a periodic delayed callback to initiate updates
i=winio@('%d1',secds,redraw_screen)
END
c -----
INTEGER FUNCTION redraw_screen()
INCLUDE <windows.ins>
INTEGER xres,yres,g_handle,r,g,b,loop
COMMON xres,yres,g_handle,r,g,b
CALL select_graphics_object@g_handle)
DO loop=1,yres
    CALL draw_line_between@(0,loop,xres,loop,RGB@(r,g,b))
    r=r+1
    g=g+3
    b=b+5
    IF(r.GT.255) r=0
    IF(g.GT.255) g=g-255

```

```
IF(b.GT.255) b=b-255
ENDDO
redraw_screen=1
END
```

Disable screen saver - %ns

%ns is designed to cause screen saver activity to be inhibited while the window is open. If your application opens a window and then performs a long calculation, there is a danger that a screen saver will cut in when your program yields, even temporarily. This is because the operating system considers the system to be idle if there is no input for a set period of time. Many screen savers are very CPU intensive and will starve your program of CPU resources. Use %ns to eliminate this problem.

15.

Graphics

Introduction

ClearWin+ includes a library of drawing routines for drawing lines and text, for filling areas, for copying images, etc.. These routines do not have an explicit argument referring to the ‘surface’ on which the drawing is to take place. Instead, drawing routines are applied to what is called the current *drawing surface*. This *drawing surface* can be:

- 1) one of a number of rectangular areas of the screen generated by %gr,
- 2) one of a number of printer bitmaps (created for example by open_printer@),
- 3) an internal *drawing surface* created by create_graphics_region@ (that is, a bitmap that is created for copying to another *drawing surface*).

When it is necessary to distinguish one surface from another, the programmer provides a unique integer identifier (called a *handle*) and the routine select_graphics_object@ is used to provide the switch. This chapter describes *drawing surfaces* created using %gr and create_graphics_region@. Printer surfaces are described in Chapter 18.

%gr is one of four graphics format codes provided by ClearWin+. The others are %og (used to create a region in which OpenGL routines are used; described in the next chapter), %pl (used to create a region in which SIMPLEPLOT routines are used; described in Chapter 17) and %dw (used to create a region in which Windows API functions are used; described at the end of this chapter). Using these formats it is possible to embed your graphics in a window together with other controls. The ClearWin+ library of drawing routines is used only with %gr. Thus, only %gr (not %og, %pl and %dw) generates a *drawing surface*.

Graphics format %gr[*options*]

There are two colour modes for ClearWin+ drawing routines: (a) *VGA colour* mode, designed to emulate the EGA and VGA palette registers and used to port legacy code to Windows and (b) *RGB colour* mode, the native Windows colour mode.

To assist in the porting of legacy code, the default colour mode is currently set as *VGA colour* mode. However, it is recommended that all new programs should use *RGB colour* mode. There are three ways to change the colour mode.

1. The default is changed to *RGB colour* mode by calling `set_rgb_colours_default@(1)` at the beginning of the program. This is recommended for new programs.
2. Alternatively, when creating a *drawing surface* using `%gr`, you can include the option `RGB_COLOURS` (see below).
3. In either case you can change the colour mode for a particular *drawing surface* by calling `use_rgb_colours@`.

`%gr` can be followed by a list of options that is any valid combination of the following:

BLACK, WHITE, RED, BLUE, YELLOW, GREY (or GRAY)	Specifies the initial background colour of the drawing surface. Otherwise this defaults to the background colour of the surrounding window.
METAFILE_RESIZE	Options specifying the behaviour of the graphics area as it is re-sized. See later in this chapter.
USER_RESIZE	
RGB_COLOURS	Selects <i>RGB colour</i> mode.
FULL_MOUSE_INPUT	Specifies that mouse movements and mouse button releases should be notified to the call-back function, not just clicks and double clicks.
BOX_SELECTION	Specifies the initial selection mode, that controls the behaviour when the user presses the left mouse button and drags the mouse over the graphics area. These options are useful for interactive programs.
LINE_SELECTION	
USER_SURFACE	A Win32-only option to enable software to access the in-memory copy of the graphics area so as to modify it at the highest possible speed. This option is particularly useful for image processing software.

`%gr` creates a ‘control’ that is a rectangular graphics area (*a drawing surface*) that will be filled by subsequent calls to ClearWin+ drawing routines. In the absence of `%gr`, using drawing surface routines causes ClearWin+ to create a separate window

containing a drawing surface together with a button to close the window. This is analogous to the automatic creation of a *ClearWin* window for standard Fortran I/O. The various options to the %gr format are explained in this and subsequent sections of the chapter.

%gr takes two arguments that provide the pixel width and height of a rectangular area. You may want to vary these values depending on the resolution of the user's screen. The pixel width and depth of the screen is obtained by calling clearwin_info@ using SCREEN_WIDTH and SCREEN_DEPTH.

The %gr options consisting of simple colour names (such as BLACK) supply an initial background colour that would otherwise default to the background colour of the surrounding window. The option RGB_COLOURS can be used if the default has not been changed to RGB colour mode by calling set_rgb_colours_default@(1).

For example:

```
INTEGER ctrl,i
i=winio@('%ww%gr[black,rgb_colours]&',400L,300L)
i=winio@('bt[OK]%'lw',ctrl)
CALL draw_line_between@(100,100,300,200,RGB@(255,0,0))
```

This will draw a red line on a black background. In this example, %lw has been used to leave the window open so that code to draw to the *drawing surface* can follow. An alternative approach is to provide a startup call-back function using %sc and to place the graphics code in the call-back. This approach is usually preferred unless you need to use %lw for other reasons.

If more than one *drawing surface* is to be drawn (either within one window, or on different windows) then a grave accent modifier is supplied to %gr and a third argument is required to input an integer (called a handle) that you provide. This handle is used in the subroutine select_graphics_object@ in order to select the current *drawing surface*.

The following example creates two *drawing surfaces* and draws to each in turn.

```
INTEGER ctrl,hnd1,hnd2
DATA hnd1,hnd2/1,2/
c ----Graphics handles are input values to %gr ---
c
CALL set_rgb_colours_default@(1)
i=winio@('%`gr&',400L,300L,hnd1)
i=winio@('%`gr&',400L,300L,hnd2)
i=winio@('%ww%'lw',ctrl)
i=select_graphics_object@(hnd1)
CALL draw_line_between@(100,100,300,200,RGB@(0,0,255))
i=select_graphics_object@(hnd2)
CALL draw_filled_ellipse@(200,150,75,50,RGB@(255,0,0))
```

Other *drawing surfaces* can be set up to write an image to memory or to write output to the printer. The graphics calls in the above program can also be used on such surfaces. Details appear later in this chapter and in Chapter 18.

Basic graphics primitives

Once a %gr *drawing surface* has been created, the following functions are used to draw in the space:

DRAW_LINE_BETWEEN@	Draws a straight line between two points.
DRAW_CHARACTERS@	Draws text.
DRAW_ELLIPSE@	Draws an ellipse.
DRAW_FILLED_ELLIPSE@	Fills an ellipse.
DRAW_FILLED_POLYGON@	Fills a polygon.
DRAW_FILLED_RECTANGLE@	Fills a rectangle.
DRAW_POLYLINE@	Draws a number of connected straight lines.
DRAW_RECTANGLE@	Draws a rectangle.
DRAW_POINT@	Sets a pixel colour.
SET_LINE_STYLE@	Sets attributes for line drawing.
SET_LINE_WIDTH@	Sets the width for line drawing.

Drawing text

Text may be added to a %gr *drawing surface* by using `draw_characters@`, for example,

```
CALL draw_characters@('Time (Secs)',350,380,RGB@(255,255,255))
```

The initial font is the current font in the format window, but it can be changed by using the following routines. These operate on the currently selected *drawing surface*.

Note that generally only TrueType fonts can have their appearance varied in all of the ways shown below.

BOLD_FONT@	Selects/deselects bold font.
CHOOSE_FONT@	Displays a font selection dialog box.

GET_FONT_NAME@	Can be repeatedly called to obtain the name of all installed fonts on the system.
GET_TEXT_SIZE@	Gets the dimensions of a given string.
ITALIC_FONT@	Selects/deselects italic the font.
ROTATE_FONT@	Rotates the font anticlockwise by a given angle in degrees.
SCALE_FONT@	Multiplies the font size.
SELECT_FONT@	Selects a font by name.
SIZE_IN_PIXELS@	Specifies the font size in pixels.
SIZE_IN_POINTS@	Specifies the font size in points.
UNDERLINE_FONT@	Selects/deselects underlined font.

The following program demonstrates the use of these routines

```

WINAPP
INCLUDE <clearwin.ins>
INTEGER i,ictrl,s_width,s_depth,g_width,g_depth,colour
c--- Get width and height of screen
    s_width = clearwin_info@('SCREEN_WIDTH')
    s_depth = clearwin_info@('SCREEN_DEPTH')
c--- Size of graphics area
    g_width = 2.0*s_width/3.0
    g_depth = 2.0*s_depth/3.0
c--- Graphics window
    i=winio@('%w%ca[WINDOWS FONTS]&')
    i=winio@('%mn[&Exit]&','EXIT')
    i=winio@('%gr[black,rgb_colours]&',g_width,g_depth)
    i=winio@('%lw',ictrl)
c--- Select Windows font 'Modern' (default attributes)
c--- Draw in intense white
    CALL select_font@('Modern')
    CALL draw_characters@('1: select_font:modern',
+                                350,380,RGB@(255,255,255))
c--- Multiply size of font by 2
c--- Draw in yellow
    CALL scale_font@(2.0D0)
    CALL draw_characters@('2: This is scale_font',
+                                300,300,RGB@(255,255,0))
c--- Underline font
c--- Draw in intense cyan
    CALL underline_font@(1)

```

```

        CALL draw_characters('3: This is underline_font',
+                           280,200,RGB@(0,255,255))
c--- Italic font
c--- Draw in intense green
    CALL italic_font@(1)
    CALL draw_characters('4: This is italic_font',
+                           300,100,RGB@(0,255,0))
c--- Specify the size of the font in pixels
c--- Draw in cyan
    CALL size_in_pixels@(20,10)
    CALL draw_characters('5: This is size_in_pixels',
+                           350,50,RGB@(127,0,127))
c--- Specify the size of the font in points
c--- Draw in intense blue
    CALL size_in_points@(20,10)
    CALL draw_characters('6: This is size_in_points',
+                           10,50,RGB@(0,0,255))
c--- Rotate the font by 70.0 degrees
c--- Draw in intense magenta
    CALL rotate_font@(70.0D0)
    CALL draw_characters('7: This is rotate_font:70',
+                           80,450,RGB@(255,0,255))
c--- Undo underline font
c--- Draw in intense cyan
    CALL underline_font@(0)
    CALL draw_characters('8: This is undo underline_font',
+                           150,450,RGB@(0,255,255))
c--- Embolden the font
c--- Draw in intense white
    CALL bold_font@(1)
    CALL draw_characters('9: This is bold_font',
+                           220,350,RGB@(255,255,255))
c--- User selected font, default colour black (0)
c--- Draw in intense white
    colour=0
    CALL choose_font@(colour)
    CALL draw_characters('10: This is choose_font',
+                           300,450,colour)
END

```

Note how `choose_font@` allows user selection of font style, underline, overstrike, size and colour.

Re-sizing a graphics surface

It is often desirable to allow a graphics area to be re-sized by the user by altering the size of the window in which it is contained. Like all ClearWin+ controls, by default the *drawing surface* created by %gr will have a fixed size. However, %gr can take a pivot, enabling it to be re-sized. Unlike other simple controls, the desired effect of re-sizing a graphics area is not obvious. ClearWin+ offers the choice of either re-drawing the image from scratch (with different contents if you wish) or of letting the system re-scale the existing graphics.

A %gr region will change size when the enclosing window is re-sized if it is preceded by a pivot format (%pv) and provision has been made to re-draw the graphics by one of the following two methods:

- 1) If the USER_RESIZE option is given to %gr and a call-back function is supplied (i.e. %^gr[user_resize]) then, when the window is re-sized, the whole area will be blanked out and the call-back function will be called. It is assumed that this function will re-draw the contents of the box. The function can determine that a re-draw is required by calling clearwin_string@ using CALLBACK_REASON. This will return the string RESIZE if a re-size is occurring. The clearwin_info@ parameter GRAPHICS_RESIZING will also return 1 if a re-size is occurring, but this has been superseded by the general purpose CALLBACK_REASON mechanism. The clearwin_info@ parameters GRAPHICS_WIDTH and GRAPHICS_DEPTH are used to return the new size of the area. For convenience the box is assumed to be re-sized immediately it is created, so it is only necessary to draw your graphics in one place in your code. Here is some sample code:

```
WINAPP
  INTEGER i,winio@
  EXTERNAL draw
  i=winio('@%ww%pv%^gr[user_resize,rgb_colours]',80L,25L,draw)
END

INTEGER FUNCTION draw()
  INCLUDE <windows.ins>
  INTEGER x,y,resize
  resize=clearwin_info@('GRAPHICS_RESIZING')
  IF(resize.EQ.1)THEN
    x=clearwin_info@('GRAPHICS_WIDTH')
    y=clearwin_info@('GRAPHICS_DEPTH')
    CALL draw_line_between@(0,0,x,y,RGB@(255,0,0))
  ENDIF
```

```
draw=2
END
```

- 2) An alternative method for re-sizing a %gr region is to supply the option metafile_resize to the %gr format code. In this case, a record (stored in a Windows object known as a meta-file) of all graphics operations is made and the result is re-played at the new box size. This approach is clearly the easiest to program, however there are two factors to consider before using this technique.
- a) The number of graphics operations being sent to the region must not be unreasonably large since each graphics call will be recorded for re-play. For example, a fractal drawing program (that continued to draw more detail for as long as it ran) could not use this technique because it would ultimately run out of memory.
 - b) Also some images may not respond well to such automatic re-scaling. This is because floating point co-ordinates are truncated to integers when the original image is drawn. When scaled, they are truncated a second time. In general, line drawing will still look good, but images drawn pixel by pixel are probably best re-sized by method 1.

The %gr call-back function

Simple graphics operations can be performed without the use of a call-back function. However, you will need a call-back function if you want to respond to mouse changes, or to icon movements, or if you want to redraw in response to a re-size event.

If a call-back function is supplied with %gr, it will be called for a variety of reasons, conveniently distinguished using clearwin_string@('CALLBACK_REASON'). Here is the full set of reasons for calling the call-back function:

MOUSE_LEFT_CLICK	These reasons are given after the corresponding mouse button has been pressed and released, except in FULL_MOUSE_INPUT mode, when they occur when the key is pressed.
MOUSE_MIDDLE_CLICK	
MOUSE_RIGHT_CLICK	
MOUSE_LEFT_RELEASE	These reasons only occur if the option FULL_MOUSE_INPUT is used with %gr. They occur when the corresponding mouse button is released.
MOUSE_MIDDLE_RELEASE	
MOUSE_RIGHT_RELEASE	
MOUSE_DOUBLE_CLICK	This reason is given when two successive left mouse clicks have occurred within the interval defined as a double click. Individual mouse LEFT_MOUSE_CLICK events will

also be generated.

RESIZE

Given when the USER_RESIZE option is used and the window has just been created or re-sized. Usually the call-back function will draw or re-draw the image.

DRAG_AND_DROP

Given when an icon has been dragged and dropped on to the graphics area.

Tip

Many systems are configured so that the middle mouse button is inactive, even though it is physically present on the mouse. For this reason it is better to avoid relying on this button.

Once the reason for the call-back has been established, additional information can be obtained by calling `clearwin_info@`. You can use `GRAPHICS_MOUSE_X` and `GRAPHICS_MOUSE_Y` to obtain the pixel co-ordinates of the mouse event that caused the call-back. Also `GRAPHICS_MOUSE_FLAGS` gives full information about the state of the mouse keys (this is most useful when the `FULL_MOUSE_INPUT %gr` option is used). The mouse flags are bitwise significant as follows:

-
- | | |
|----|-----------------------|
| 1 | Left button pressed |
| 2 | Right button pressed |
| 4 | Shift key pressed |
| 8 | Control key pressed |
| 16 | Middle button pressed |
-

Tip

Using the `FULL_MOUSE_INPUT` option it is possible to respond to a mouse press by altering parts of your image, reversing these changes when the key is released again. This can give a very dynamic feel to your program. Remember, however, that any operations you perform at this point should not take too long, otherwise the mouse response will become sluggish.

Graphics selections

Consider the task of writing a simple paint program. Such programs often allow the user to hold down the left mouse key in order to open up a box that is used to select an area. Alternatively, the effect of dragging the mouse could be to create a line rather than a box.

`%gr drawing surfaces` have an associated ‘selection mode’. This mode tells ClearWin+ what to do if the mouse is dragged (by the left button) over the area. The

mode can be set using the BOX_SELECTION or LINE_SELECTION options on the %gr format code. The mode can also be changed dynamically using the set_graphics_selection@ function. This operates on the current drawing surface. The argument to this function takes the following values:

0	default - no selection
1	box selection
2	line selection

When a selection is enabled, dragging the mouse with the left button depressed opens an XOR drawn box or line. A program obtains the co-ordinates using get_graphics_selected_area@. Typically you would call this function on the release of the left mouse (or possibly in response to a button press) so that the program would respond to the final position of the line or box.

The following example uses BOX_SELECTION. When the left mouse button is released, the program draws a diagonal line using the box position and dimensions. If the shift key is held down, a copy of the box is produced instead of a line. Alternatively, if the control key is held down, an ellipse is drawn to fill the box.

```

WINAPP
INTEGER i,wini@
CHARACTER*80 fmt
EXTERNAL gr_func

i=wini@( '%ww[no_border]%ca[Box Selection]%sy[3d_depressed]&')
fmt='%^gr[black,box_selection,full_mouse_input,rgb_colours]'
i=wini@(fmt,200,200,gr_func)
END

INTEGER FUNCTION gr_func()
INCLUDE <windows.ins>
INTEGER x1,y1,x2,y2,a,b,flags,grey
LOGICAL IsUp,WasDown
DATA WasDown/.FALSE./

flags=clearwin_info@('graphics_mouse_flags')
IsUp=AND(flags,MK_LBUTTON).EQ.0

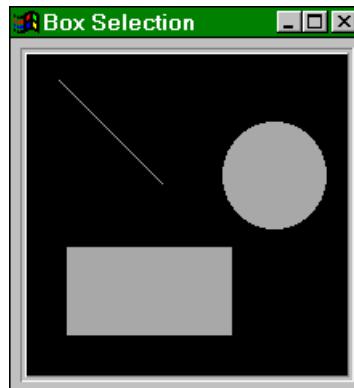
IF(IsUp.AND.WasDown)THEN
  grey=RGB@(170,170,170)
  CALL get_graphics_selected_area@(x1,y1,x2,y2)
  CALL set_graphics_selection@(0)
  IF(AND(flags,MK_SHIFT).EQ.MK_SHIFT)THEN
    CALL draw_filled_rectangle@(x1,y1,x2,y2,grey)

```

```

ELSEIF(AND(flags,MK_CONTROL).EQ.MK_CONTROL)THEN
    a=0.5*(x2-x1)
    b=0.5*(y2-y1)
    CALL draw_filled_ellipse@(x1+a,y1+b,a,b,grey)
ELSE
    CALL draw_line_between@(x1,y1,x2,y2,grey)
ENDIF
CALL set_graphics_selection@(1)
ENDIF
WasDown=.NOT.IsUp
gr_func=1
END

```



Drawing device independent bitmaps

When using %gr and its associated *drawing surfaces* (including internal and printer *drawing surfaces*) there are two sets of routines for processing device independent bitmaps (DIBs). The first set uses a CHARACTER array supplied by the programmer. The second uses a handle supplied by ClearWin+.

Using a 24 bit per pixel array

The following routines are used to handle device independent bitmaps:

GET_DIB_SIZE@	Gets parameters from a bitmap file.	p362
GET_DIB_BLOCK@	Extracts a rectangular block from a bitmap file.	p361
PUT_DIB_BLOCK@	Saves a rectangular block to a bitmap file.	p391
DISPLAY_DIB_BLOCK@	Displays a block on the current <i>drawing surface</i> .	p353

`GET_DIB_SIZE@` obtains information about an existing .BMP file. The file name is an input argument that should have the .BMP suffix. The file name may be a full path name. If the routine succeeds, an error code is returned with the value zero whilst other arguments return the width and height (in pixels) of the image contained in the file. The number of bits per pixel is also returned as one of the following values:

- 1 Monochrome black and white image
- 4 16-colour image
- 8 256-colour image
- 24 Full colour image in which each pixel is represented by eight bits per red/green/blue value.

`get_dib_block@` extracts a rectangular block (that could be the whole image) from a given file into an array. The first index of the array is the colour index in the order (red, green, blue). Displacements AX and AY into the array are typically zero but they can be set to other values if, for example, the array is to be composed of a montage of several blocks. Arguments W and H represent the width and height of the block to be transferred, whilst DX, DY represent the displacement (again typically zero) into the image in the file.

`display_dib_block@` transfers a rectangular block of given size ($W \times H$) from the array at a position (AX, AY) to a position (X, Y) on the current *drawing surface* (e.g. a %gr region or a printer). Two arguments named FUNCTION and MODE are normally set to zero. An argument for the error status is returned as zero if the process is successful.

`put_dib_block@` transfers a rectangular block from an array to given file, destroying any previous contents. Displacements AX and AY into the array are typically zero, whilst arguments W and H are the width and height of the image to be created. An argument for the number of bits per pixel is currently ignored and should set to 24. This routine always writes BMP files with 24 bits per pixel.

The use of these routines is best illustrated by a simple example. The following example reads a BMP file, alters the array to draw a red line through the image, displays the result, and writes a new BMP file with the modified image.

```

WINAPP
INCLUDE <windows.ins>
CHARACTER*50 file
CHARACTER a(3,1024,1024)
INTEGER hres,vres,nb_colours,ier,i,k,control
c--- Note - you may need to alter this path
file='c:\windows\setup.bmp'
CALL get_dib_size@(file,hres,vres(nb_colours,ier)

```

```

IF(ier.NE.0.OR.hres.GT.1024.OR.vres.GT.1024) STOP 'TROUBLE'
CALL get_dib_block@(file,a,1024,1024,0,0,hres,vres,0,0,ier)
IF(ier.NE.0) STOP 'TROUBLE'
c-- Draw a diagonal red stripe over the image
k=MIN(hres,vres)
DO i=1,k
  a(1,i,i)=char(255)
  a(2,i,i)=char(0)
  a(3,i,i)=char(0)
ENDDO
c-- Display the image
i=winio@('%gr%lw',hres,vres,control)
CALL display_dib_block@(0,0,a,1024,1024,0,0,hres,vres,0,0,ier)
c-- Write the image away to another file
file='c:\temp\junk.bmp'
CALL put_dib_block@(file,a,1024,1024,0,0,hres,vres,24,ier)
END

```

The example is written in Fortran 77 with the array A declared to be ‘sufficiently large’. With Fortran 90 you can use an allocatable array instead. Note that the resulting BMP file will typically be larger than the original because the colours will have been expanded to 24 bits per pixel.

Using a DIB handle

An alternative approach is to use routines that transmit a handle for a device independent bitmap.

The routines `import_bmp@`, `import_pcx@`, `get_screen_dib@` and `clipboard_to_screen_block@` each return a handle to a DIB. This handle is then used in `dib_paint@`, `print_dib@`, `export_bmp@` and `export_pcx@`. When it is no longer needed the associated memory can be released using `release_screen_dib@`.

You can also use `write_graphics_to_bmp@` and `write_graphics_to_pcx@` in order to write the current *drawing surface* to a file.

Here is a simple program that uses some of these routines:

```

WINAPP
INCLUDE <windows.ins>
INTEGER hres,vres,nbbp,ercode,hdib,ctrl,i,width,height
CHARACTER*50 file
file='c:\windows\setup.bmp'
CALL get_dib_size@(file,hres,vres,nbbp,ercode)
IF(ercode.NE.0) STOP 'Size error'

```

```

hdib=import_bmp@(file,ercode)
IF(ercode.NE.0) STOP 'Import error'
i=winio@('%gr[rgb_colours]%'lw',hres,vres,ctrl)
ercode=dib_paint@(0,0,hdib,0,0)
IF(ercode.EQ.0) STOP 'Paint error'
call draw_line_between@(0,0,hres,vres,RGB@(255,0,0))
IF(open_printer@(7).EQ.1)THEN
    CALL get_graphical_resolution@(width,height)
    ercode=print_dib@(7,0,0,width,height,hdib,0,0,hres,vres)
    IF(ercode.EQ.0) STOP 'Print error'
    ercode=close_printer@(7)
    IF(ercode.EQ.0) STOP 'Close printer error'
ENDIF
CALL release_screen_dib@(hdib)
END

```

This program imports a bitmap and then creates a *drawing surface* on the screen using %gr. The bitmap is drawn to the screen and a red line is draw across it. The standard printer dialog is then displayed via a call to open_printer@. If the user clicks on the OK button, the routine returns the value 1 and the current graphics device is now the printer with device handle 7. The bitmap is drawn to this device. Calling close_printer@ sends the image to the printer.

Direct manipulation of the graphics surface

Every call to a drawing surface routine involves one or more calls the Windows API in order to perform the operation. This is usually not a problem when high level operations such as line drawing are being performed. However, it might be desirable to bypass this mechanism and construct the surface directly, particularly in applications that manipulate images and that must therefore operate at the pixel level. This is possible under Win32 (not Win32S or Windows 3.1) by specifying the USER_SURFACE option with %gr.

When this option is used, an INTEGER*4 variable is placed immediately after the two size parameters. This variable will receive the address of a buffer containing the graphics contents stored using 3 bytes per pixel. By manipulating this array you can change the graphics surface directly. The INTEGER variable should be passed to window_update@ after such a manipulation to cause the screen to be updated.

Later versions of ClearWin+ may accelerate some function calls (such as draw_line_between@) by using this direct access method if USER_SURFACE is coded, but at present this is not the case.

The USER_SURFACE option implies the RGB_COLOURS option and must not be mixed with METAFILE_RESIZE, although it can be combined with USER_RESIZE. A trade-off is involved here. The user surface will usually require more memory than the device dependent bitmap that ClearWin+ normally uses to implement %gr. Also each re-draw of the window will be slower. However, if you implement pixel-by-pixel graphics by directly manipulating the buffer, you will see a substantial speed improvement over making calls to draw_point@. If USER_SURFACE is combined with USER_RESIZE then the surface pointer variable will be updated each time the window is re-sized. It is important not to copy this variable and so use an obsolete version of the pointer. The row of the buffer has a layout corresponding to the following array:

```
INTEGER*1 buffer(3,width)
```

The first index covers the colours as follows:

1	Blue
2	Green
3	Red

After each row a few bytes of padding are added if necessary to make the row a multiple of four bytes. The next row then follows. This layout is imposed by the Windows operating system and not by ClearWin+.

Here is a simple Fortran example:

```
WINAPP
INCLUDE <windows.ins>
EXTERNAL draw
INTEGER ptr
COMMON ptr
ia=winio@('%ww%pv%^gr[black,user_resize,'//
+     'user_surface]',256,256,ptr,draw)
END

INTEGER FUNCTION draw()
INCLUDE <windows.ins>
INTEGER ptr
COMMON ptr
INTEGER width,depth,m,i,q,x,y,full_on,pad_width
IF(clearwin_info@('graphics_resizing').eq.1)THEN
    width=clearwin_info@('graphics_width')
    depth=clearwin_info@('graphics_depth')
    m=MIN(width,depth)
    c--- Calculate width in bites of one row
    c--- remembering to round up to nearest 4 bytes
```

```

          pad_width=ls(rs(3*width+3,2),2)
C---   Draw a diagonal yellow line
      D0 i=1,m
          x=i-1
          y=i-1
C---   Compute index into array
          q=3*x+pad_width*y
          full_on=255
C---   Red component
          CORE1(ptr+q+2)=full_on
C---   Green component
          CORE1(ptr+q+1)=full_on
      ENDDO
      ENDIF
      draw=2
END

```

Note that it is possible (and often very useful) to mix calls to ClearWin+ graphics routines and direct manipulation of the pixel buffer. If you do this under Windows NT you must call the API function **GdiFlush()** to ensure that the effect of any previous function calls have ‘flushed through’ before you manipulate the buffer directly. This call is harmless but unnecessary under Windows 95. If you intend to perform all your graphics by direct buffer manipulation this need not concern you.

If you want to display a large image but do not have the screen space to do so or if you are producing a graphics and text output window, the routine `scroll_graphics@` will be of great use. This routine will allow you to scroll graphics in any direction by any displacement (see page 396).

Off-screen graphics

Sometimes, it is useful to build up graphics information in memory, rather than on the screen. For example, such information might be subsequently copied to the screen at several locations, or written to a file. A complex drawing could be built up off screen whilst displaying other information. When complete, the drawing can be displayed.

In ClearWin+ there are two approaches to this issue each with its own set of routines. The first creates an internal *drawing surface* using `create_graphics_region@` and is described below. The other approach uses what is called a *virtual screen* and is described in the *ClearWin+ User's Supplement*. Mixing routines from the two sets may not produce the desired result.

An internal *drawing surface* uses the following set of routines together with the DIB routines listed in the section on page 145.

COPY_GRAPHICS_REGION@	Copies from one designated surface to another.
CREATE_GRAPHICS_REGION@	Creates a <i>drawing surface</i> of a given size.
DELETE_GRAPHICS_REGION@	Releases the memory for a <i>drawing surface</i> .
GRAPHICS_TO_CLIPBOARD@	Copies a surface to the clipboard.
SCROLL_GRAPHICS@	Scrolls a <i>drawing surface</i> .
SELECT_GRAPHICS_OBJECT@	Selects an existing <i>drawing surface</i> .

Here is a short program that creates a *drawing surface* using `create_graphics_region@`, selects the surface by using `select_graphics_object@` copies a bitmap and writes a message into it. The current screen area is then selected by another call to `select_graphics_object@` allowing subsequent graphics drawing to be directed to the screen. When a button is clicked the internal surface is copied to the screen using `copy_graphics_region@`.

```

WINAPP
IMPLICIT NONE
INCLUDE <windows.ins>
INTEGER i,ictrl,width,depth,ierr,hdib
INTEGER g_handle,r_handle,white
PARAMETER (g_handle=7,r_handle=8)
CALL set_rgb_colours_default@(1)
width=2.0*clearwin_info@('SCREEN_WIDTH')/3.0
depth=2.0*clearwin_info@('SCREEN_DEPTH')/3.0
white=RGB@(255,255,255)
i=winio@('%ww%ca[Off-Screen Regions]&')
i=winio@('%mn[&Exit]&','EXIT')
i=winio@('%lw&',ictrl)
i=winio@('%`gr[black]',width,depth,g_handle)
ierr=create_graphics_region@(r_handle,width,depth)
ierr=select_graphics_object@(r_handle)
hdib=import_bmp@('c:\windows\setup.bmp',ierr)
ierr=dib_paint@(0,0,hdib,0,0)
CALL release_screen_dib@(hdib)
CALL draw_characters@('This was drawn off screen',
+                      100,100,white)
ierr=select_graphics_object@(g_handle)
CALL draw_characters@('This was drawn to a window',
+                      100,100,white)
i=winio@('%bt[Show Off-Screen Block]')
ierr=copy_graphics_region@(g_handle,0,0,width,depth,
+                           r_handle,0,0,width,depth,SRCCOPY)
END

```

Graphics icons

It is possible to supply the user with more interesting ways to interact with your graphics area than merely clicking with the mouse. A %gr *drawing surface* can have an icon resource ‘attached’ to it. The icon can be moved freely around the graphics window with the mouse. Control of the icon is maintained by the call-back function attached to %gr.

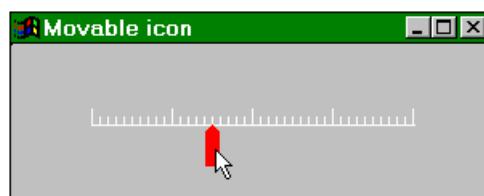
`add_graphics_icon@` is used to attach an icon resource (defined in a resource script) to the current screen drawing surface. A NAME argument is used to stipulate the resource. X and Y variables are used as arguments that hold the initial location of the image. These variables also hold the new position when the image is moved. The call-back function for %gr can modify the X and Y values if required. In this way it is possible to ‘lock’ the image on to a horizontal or vertical plane by repeatedly setting either X or Y to a constant value.

The WIDTH and HEIGHT arguments of `add_graphics_icon@` should be set to zero except when the icon image is less than 32x32 pixels. A smaller icon is constructed by starting at the top-left corner and by filling the remaining unused area with the ‘transparent’ colour. In this case, if you do not specify the correct size of the icon then the unused area will be detected by the mouse.

`clearwin_info@('DRAGGED_ICON')` returns the handle of the icon that has moved. This handle is initially obtained as the value returned by `add_graphics_icon@`.

When the icon is dropped, `clearwin_info@('DROPPED_ICON')` returns the handle of the associated icon. A graphics icon can be removed with a call to `remove_graphics_icon@`.

The following program draws a horizontal scale with an icon that can be dragged across it. The resource script attaches an icon file to the ‘arrow_r’ resource in the program. The Windows API function **ClipCursor** is used to provide a clipping rectangle for the mouse cursor when the left mouse button is held down over the icon. The Salford intrinsic CORE4 is used to pass a NULL pointer to **ClipCursor**.



```
WINAPP 0,0,'gr15x.rc'
INCLUDE <windows.ins>
```

```
INTEGER i,x,y
COMMON x,y
EXTERNAL start_cb,gr_cb
x=46
y=50
i=winio@('%ww[no_border]%ca[Movable icon]&')
i=winio@('%`cu&',CURSOR_ARROW)
i=winio@('%^gr[grey,rgb_colours,full_mouse_input]&',
+           300,100,gr_cb)
i=winio@('%sc',start_cb)
END

INTEGER FUNCTION start_cb()
INCLUDE <clearwin.ins>
INTEGER i,x,y,xx,green
COMMON x,y
i=add_graphics_icon@('arrow_r',x,y,10,26)
green=RGB@(0,255,0)
CALL draw_line_between@(50,y,251,y,green)
DO xx=50,245,5
    CALL draw_line_between@(xx,y-5,xx,y,green)
ENDDO
DO xx=50,250,50
    CALL draw_line_between@(xx,y-10,xx,y,green)
ENDDO
start_cb=1
END

INTEGER FUNCTION gr_cb()
INCLUDE <windows.ins>
INTEGER mx,my,mf,x0,y0,w0,h0,x,y,hwnd,rc(4)
COMMON x,y
LOGICAL L,WasDrag,IsDrag
DATA WasDrag/.FALSE./
CALL get_mouse_info@(mx,my,mf)
IsDrag=clearwin_info@('dropped_icon').NE.0
IF(.NOT.WasDrag.AND.IsDrag)THEN
    IF(mx.GE.x.AND.mx.LE.x+10.AND.my.GE.50.AND.my.LE.75)THEN
        hwnd=clearwin_info@('latest_formatted_window')
        CALL get_window_location@(hwnd,x0,y0,w0,h0)
        rc(1)=x0+48
        rc(2)=y0+70
        rc(3)=x0+256
        rc(4)=y0+95
```

```
L=ClipCursor(rc)
WasDrag=.TRUE.
ENDIF
ENDIF
IF(WasDrag.AND..NOT.IsDrag)THEN
L=ClipCursor(CORE4(0))
WasDrag=.FALSE.
ENDIF
y=50
IF(x.LT.46) x=46
IF(x.GT.246)x=246
gr_cb=1
END
```

Windows graphic modes

Between any Windows application and the display hardware there exists a program called a display driver. For every different make of video card there is a different display driver that is either from the Windows installation disks or from the video hardware manufacturer.

From the programmers perspective all drivers should appear the same as all Windows programs must communicate to the video display hardware via the Windows GDI interface. Drivers, on the other hand, will differ dramatically from display card to display card because the structure and types of messages required to control the video display hardware differ for each card. A video driver therefore insulates the programmer from the complexities of hardware.

Windows supports several common display mode resolutions:

640 x 480 x 16 or 256 colours

This is the most compatible mode as all Windows compatible PCs are capable of displaying at this resolution. However, it is very low quality and is now hardly ever used.

800 x 600 x 16 or 256 colours

This is a popular mode as text is readable and there is a larger display area to work with.

1024 x 768 x 16 or 256 colours

This normally used with monitors that are more than 14 inches wide. It is by far the clearest resolution but may suffer from slow updates if the display card is not of the accelerated type.

In 16 colour mode Windows uses a set palette. It holds ‘general’ colours that are of most use. If the program requires shades or colours that do not exist in the palette, Windows will attempt to dither that colour. Dithering gives a quick effect that often matches the desired colour. It is done by placing pixels of different colours next to each other so that when the user is a suitable distance away from the screen the pixels will appear to be a third colour.

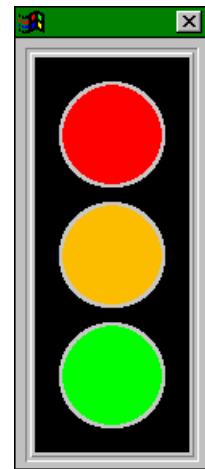
In 256 colour modes there are 256 unique palette entries. Each palette entry is made by setting a red, a green and a blue component. In this mode, when a pixel is written, a value between 0 .. 255 is placed in the display memory. When the display is refreshed the red, green and blue values are ‘looked up’ and sent to the screen.

There have recently been several new display colour ‘depths’ added as new hardware has been designed. These *high colour* modes store the colours directly in the display memory thus avoiding the need for a palette at all. The two most common colour depths are 65535 and 16.7 million. These depths allow each pixel to be set to any one of the possible colours.

If you want to use a wide range of colours whilst avoiding the complexities of managing a palette, one solution is to restrict the usage of your application to desktops operating in high colour mode. For this purpose, you can use the function `high_colour_mode@` to test if the desktop is set to operate in high colour mode.

A call to `clearwin_info@('SCREEN_NUM_COLOURS')` will give the number of colours for the desktop unless this number is greater than 256 (the value 8192 represents all values greater than 256, i.e. a high colour device). A list of palette functions is given on page 322. These are documented in the *ClearWin+ User’s Supplement*.

If the desktop graphics device is not a high colour device, when you draw to the screen, the colour that you get will not always be exactly the colour you specify in the program. This may cause difficulties if you read a pixel from the screen and try to match the colour with one that you have used earlier to draw to the screen. The following program illustrates the problem together with one method of solution. The program draws a set of ‘traffic lights’. When the user clicks on any part of the picture, the colour for that part cycles through the sequence: red, amber, green, black.



```

WINAPP
INTEGER i,winio@
EXTERNAL fill_cb,start_cb
i=winio@('%ww[no_border,no_maxminbox]&')
i=winio@('%sy[3d_depressed]&')
i=winio@('%^gr[black,rgb_colours]&',100,250,fill_cb)
i=winio@('%sc',start_cb)

```

```

        END
C-----
        INTEGER FUNCTION start_cb()
INCLUDE <clearwin.ins>
        INTEGER i,white
        CALL set_line_width@(2)
        white=RGB@(200,200,200)
        CALL draw_rectangle@(1,1,99,249,white)
        DO i=50,200,75
            CALL draw_ellipse@(50,i,32,32,white)
        ENDDO
        start_cb=1
        END
C-----
        INTEGER FUNCTION fill_cb()
INCLUDE <clearwin.ins>
        INTEGER x,y,d,f,red,green,black,amber,white,colour,a
        CALL get_mouse_info@(x,y,d)
        CALL get_rgb_value@(x,y,colour)
        red= get_matched_colour@(RGB@(255,0,0))
        green=get_matched_colour@(RGB@(0,255,0))
        amber=get_matched_colour@(RGB@(255,168,0))
        white=get_matched_colour@(RGB@(200,200,200))
        black=0

        IF(colour.EQ.black)THEN
            f=red
        ELSEIF(colour.EQ.red)THEN
            f=amber
        ELSEIF(colour.eq.amber)THEN
            f=green
        ELSE
            f=black
        ENDIF
        CALL fill_surface@(x,y,colour,f)
        fill_cb=1
        END

```

An alternative solution is to use `get_nearest_screen_colour@` in place of `get_matched_colour@`. The effect is the same but you will probably get a different amber colour on the screen for the following reasons.

When you draw to a screen *drawing surface*, ClearWin+ filters the colour through its logical palette in order to find an optimum match (we are assuming the device is not a high colour device). When ClearWin+ calls a Windows API drawing function, the result is then filtered through the system palette. Now suppose we call one of the

drawing surface functions (such as `fill_surface()`) but instead of using a simple RGB value, we use the return from `get_nearest_screen_colour()`. The effect is to apply the system palette filtering first and this over-rides the ClearWin+ colour optimisation.

The automatic ClearWin+ colour filtering described above may have the effect of slowing the output produced on a drawing surface. You can switch this filtering off by calling `use_approximate_colours@(1)`. When the filtering is switch off, use `get_matched_colour()` in order to get the filtered colour for a particular RGB triplet. This colour is then used in the drawing routines.

Owner draw box format - %dw[*options*]

The %dw format allows you to access the Windows GDI directly to prepare a bitmap for display. This is a low level method that has been provided primarily for the integration of certain existing packages. It does not use ClearWin+ drawing routines (that is, it does not generate a *drawing surface*).

The format takes one argument that is a handle to a device context acquired by calling the ClearWin+ function `get_bitmap_dc()`. This handle accesses a bitmap that is drawn to using standard Windows API functions. It is never necessary to re-paint the area because the bitmap remains in existence even if the window is obscured.

The device context will continue to exist (together with its associated bitmap) until released with the ClearWin+ function `release_bitmap_dc()`. At normal program termination the system will automatically release device contexts acquired by `get_bitmap_dc()`, so it is only necessary to use `release_bitmap_dc()` in complex programs that manipulate many graphics bitmaps.

For example:

```
WINAPP
INCLUDE <windows.ins>
INTEGER my_dc
my_dc=get_bitmap_dc@(50,50)
CALL MoveTo(my_dc,0,0)
CALL LineTo(my_dc,50,50)
i=winio@('%dw %bt[OK]',my_dc)
--- The next line is not necessary unless ---
--- many device contexts will be acquired ---
CALL release_bitmap_dc@(my_dc)
END
```



Note that you can continue to change the graphics in the bitmap after the window has been displayed (from call-back functions, or using the %lw format to leave the window in place) provided that `window_update@(HDC)` is used to update the screen, where `HDC` is the handle of the device context. Other routines that operate in terms of a handle to a device context are listed on page 319.

When it is created, the bitmap will be filled with the default background window colour (i.e. it will be invisible unless displayed inside a shaded box). The colour can be changed with a call to the standard Windows API `FillRect` function.

It is possible to change the pen associated with a device context acquired by calling the ClearWin+ function `get_bitmap_dc@` (see page 361), without worrying about deleting used pens. This is performed by calling the ClearWin+ `change_pen@` function (see page 335). The first argument in this function is a device context acquired by calling `get_bitmap_dc@` followed by three other arguments similar those supplied to the Windows API `CreatePen` function.

Currently the option list, if present, may contain:

<code>user_resize</code>	%dw can take the option <code>user_resize</code> , which allows it to be re-sized as the pivot item of a window. In this form, %dw does not take a handle from <code>get_bitmap_dc@</code> as an argument, but two integers specifying the initial width and depth of the desired owner-draw region. A call-back function must be supplied in this case. When the call-back function is called, it can check the <code>clearwin_info@</code> parameter <code>GRAPHICS_RESIZING</code> to determine that it is being called to re-draw the image (the image is deemed to be re-sized once as it is created, so all the drawing code can be placed in the call-back).
--------------------------	---

The call-back function should use `clearwin_info@` to obtain `GRAPHICS_DC`, `GRAPHICS_WIDTH`, and `GRAPHICS_DEPTH`. Traditional Windows programmers should note that although your function will be called to re-draw the image each time the size changes, you will not be called each time a `WM_PAINT` message is processed. For example, if the image is obscured and then exposed again without change, your function will not be called.

<code>full_mouse_input</code>	With this option the %dw call-back function is called each time the mouse is moved, or a button is pressed or released within the area of the control. Otherwise the call-back function only responds to mouse input when the user clicks
-------------------------------	---

on the control.

You can use `get_mouse_info@` to obtain mouse data from within a `%dw` call-back function.

Updating the screen

By default a call to a graphics function will not produce an instantaneous update of the screen. Instead **ClearWin+** waits until you have finished your current sequence of graphics calls from within your call-back (i.e. it waits until the application is idle). This mechanism is suitable for most circumstances. However, a call to the routine `perform_graphics_update@` will cause an immediate update should this be required.

When your application is in an idle state or a call to `perform_graphics_update@` is made, changes to the graphics buffer are copied to the screen. **ClearWin+** keeps track of the smallest rectangle to enclose the changed areas and updates only this region. As a result, if your program updates information at opposite corners of a large graphics region (as an extreme example) it may be faster to insert calls to `perform_graphics_update@` so that each small region is updated separately.

If you use `%dw`, **ClearWin+** cannot automatically detect the smallest rectangle to be updated and so by default it updates the whole of the graphics region whenever a change is made. A call to the routine `set_update_rectangle@(x1,y1,x2,y2)` sets the update rectangle making the program run faster. Once used, this routine must be called whenever the update rectangle changes.

16.

OpenGL

Introduction

OpenGL provides a relatively simple, platform independent, way to produce complex (possibly animated) 2 or 3-dimensional images. Lighting and perspective effects are easily introduced, although in order to achieve high performance, full ray tracing (which produces shadows and effects caused by multiply scattered light rays) is not available. The performance of OpenGL is sufficient that it is often used for 3-D animation. The following two books explain the use of OpenGL without going into platform-specific details:

OpenGL Programming Guide

OpenGL Reference Manual

See the end of this chapter for further details. OpenGL is available via two DLLs that may already be installed on your PC.

This chapter does not present a tutorial on OpenGL nor does it describe the OpenGL API. You are advised read the official “Red Book” learning guide to OpenGL, and to visit the OpenGL web site at <http://www.OpenGL.org>

Examples in the *Red Book* are simplified by the use of a specially designed and written auxiliary OpenGL library (called GLAUX). GLAUX provides an easy way to open a window with the OpenGL attributes that you require. However, it is limited in its use of mouse, keyboard and message handling and has the overwhelming disadvantage you cannot attach menus or buttons. Also, it only allows one window to be opened. *ClearWin+* is very similar to GLAUX with regard to the ease with which OpenGL windows can be opened. However, *ClearWin+* has none of the above restrictions. Nevertheless, the *Red Book* examples provide a convenient tutorial approach to learning how to call the OpenGL library from *ClearWin+*. Much of this

chapter is devoted to discussing how to convert the *Red Book* examples that use GLAUX to ClearWin+.

OpenGL - %og[*options*]

In order to open an OpenGL window with ClearWin+, you must use the %og format code:

```
i=wini0('%og[static]')
```

%og takes an list of options consisting of any valid combination of the following:

<code>double</code>	This option is used for animation and causes OpenGL output to be sent to an in-memory buffer until the <code>swap_opengl_buffers@</code> subroutine is called. This helps to reduce flicker at the expense of requiring more memory.
<code>depth16</code>	Specifies 16 bits of depth buffering.
<code>depth32</code>	Specifies 32 bits of depth buffering.
<code>stencil</code>	Stencil buffer. The stencil buffer is used to define masks used to prevent part of image being drawn. For example, OpenGL only allows convex polygons to be filled. More complex polygons require the use of the stencil buffer.
<code>accum</code>	The accumulation buffer is used to accumulate images for effects such as anti-aliasing motion blur, depth of field etc.
<code>user_resize</code>	Same as %gr usage.
<code>static</code>	Used to redraw window after it has been partially revealed.

Any combination of these attributes is permitted. Depth buffering is an option that can be omitted. If both `depth16` and `depth32` are specified, `depth32` takes effect.

These attributes, with the exception of `static`, are directly analogous to the GLAUX equivalent. For example, the GLAUX attribute `AUX_ACCUM` is the same as the ClearWin+ attribute `ACCUM`.

Converting examples written using GLAUX to ClearWin+ is, therefore, relatively straight forward. Most of the differences arise through language choice rather than by the use of ClearWin+. A Fortran program will inevitably have a slightly different structure from a C program.

The conversion of a Red Book example

The example in listing 1.2 of the *Red Book*, *simple.c* is reproduced below. Each line in the source is numbered to aid description.

```

1 /* simple.c */
2
3 #include <GL/gl.h>
4 #include <GL/glaux.h>
5 #include <stdlib.h>
6
7
8 int main(int argc, char** argv)
9 {
10 auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
11 auxInitPosition (0, 0, 500, 500);
12 auxInitWindow ("Simple OpenGL Sample");
13
14 glClearColor (0.0, 0.0, 0.0, 0.0);
15 glClear(GL_COLOR_BUFFER_BIT);
16 glColor3f(1.0, 1.0, 1.0);
17 glMatrixMode (GL_PROJECTION);
18 glLoadIdentity ();
19 glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
20 glBegin(GL_POLYGON);
21 glVertex2f(-0.5, -0.5);
22 glVertex2f(-0.5, 0.5);
23 glVertex2f(0.5, 0.5);
24 glVertex2f(0.5, -0.5);
25 glEnd();
26 glFlush();
27 _sleep (10000);
28 return(0);
29 }
```

When converting to Fortran, lines 1 to 9 may be ignored. Lines 10 to 12 describe the window properties to GLAUX. These properties are:

AUX_SINGLE single buffered. This is ClearWin+ default and is not necessary in the ClearWin+ version.

AUX_RGB use RGB colours. ClearWin+ only supports RGB colours for OpenGL. Colour index modes are unsuitable for shading and anti-aliasing and so are not supported.

0, 0 initial window position. This is specified using %sp in ClearWin+.

500, 500 initial window size. This is specified using %og in ClearWin+.

"Simple OpenGL Sample" window caption.

In addition, GLAUX windows may be closed using the **Escape** key. Specify this with %es using ClearWin+.

ClearWin+ uses %lw to allow a window to stay open when the **winio@** call returns. If %lw were not specified, the drawing commands following would have no effect.

Lines 10 to 12 are replaced with the following calls to **winio@**. The ampersands at the end of every line are used to combine several **winio@** calls into one format window.

```
c----Allows the user to close the window with the <esc> key
    i=winio@('%es&')
c----Window caption
    i=winio@('%ca[Simple OpenGL Sample]&')
c----Sets initial window position to (0,0)
    i=winio@('%sp&',0,0)
c----Makes the same style of window as GLAUX
    i=winio@('%ww[no_border]&')
c----Creates the OpenGL window and returns leaving it open
    i=winio@('%og[static]%'lw',500, 500,ctrl)
```

This may all be combined into:

```
i=winio@('%es%ca[Simple OpenGL Sample]&')
i=winio@('%sp%ww[no_border]%'og[static]%'lw',
&           0,0,500,500,ctrl)
```

The use of line 28 with the *Red Book* example is to keep the window open for 10 seconds before closing it. This is unnecessary since the ClearWin+ window will stay open anyway until the user closes it. Note that the integer variable, **ctrl** is required since we have used %lw but that the variable itself is unused.

The Fortran include files *clearwin.ins* and *opengl.ins* must be included at the top of the main program.

The remainder of the program is virtually unchanged. The full ClearWin+ version appears below.

```
PROGRAM Simple
INCLUDE <clearwin.ins>,nolist
INCLUDE <opengl.ins>,nolist
REAL t1,t2
INTEGER i, ctrl
i=winio@('%es%ca[Simple OpenGL Sample]&')
i=winio@('%sp%ww[no_border]%'og[static]%'lw',
```

```
&           0, 0, 500, 500, ctrl)
CALL glClearColor (0.0, 0.0, 0.0, 0.0)
CALL glClear(GL_COLOR_BUFFER_BIT)
CALL glColor3f(1.0, 1.0, 1.0)
CALL glMatrixMode(GL_PROJECTION)
CALL glLoadIdentity()
CALL glOrtho(-1d0, 1d0, -1d0, 1d0, -1d0, 1d0)
CALL glBegin(GL_POLYGON)
CALL glVertex2f(-0.5, -0.5)
CALL glVertex2f(-0.5, 0.5)
CALL glVertex2f( 0.5, 0.5)
CALL glVertex2f( 0.5, -0.5)
CALL glEnd()
CALL glFlush()
END
```

As can be seen, the OpenGL calls and their positions are unchanged. A couple of calls to `winio@` have replaced the calls to GLAUX. Overall, the two programs are very similar. This will be reinforced but qualified when we look at other examples.

Argument types

Some OpenGL functions take single precision arguments, others double precision. In many cases it is possible to determine the argument type from the variant of the name. For example, `glColor3f` requires single precision arguments whereas, `glColor3d` requires double precision arguments.

With other OpenGL functions, the name does not help. For example, `glAccum` and `glClearColor` require single precision arguments whereas, `glOrtho` takes double precision arguments.

Unlike OpenGL functions, all GLAUX functions that take floating-point arguments require the arguments to be double precision.

You should carefully check the types of the arguments to any function before using it.

You must specify a double precision constant by using a 'D' exponent. Any constants for which you use neither 'D' nor DBLE() will be regarded as single precision and will yield unexpected results.

When linking, you are recommended to link directly with the import libraries rather than the DLLs. Using the import library will cause mismatched arguments to result in a link failure and enable you to quickly correct any errors.

Call-backs and initialisation

This next sample program is considerably more advanced. It uses *double.c* and Listing 1.3 from the *Red Book*. It illustrates the use of the mouse and various call-back functions. We will convert it to ClearWin+.

```
1 #include "glos.h"
2
3 #include <GL/gl.h>
4 #include <GL/glu.h>
5 #include <GL/glaux.h>
6
7 void myinit(void);
8 void CALLBACK spinDisplay (void);
9 void CALLBACK startIdleFunc (AUX_EVENTREC *event);
10 void CALLBACK stopIdleFunc (AUX_EVENTREC *event);
11 void CALLBACK myReshape(GLsizei w, GLsizei h);
12 void CALLBACK display(void);
13
14 static GLfloat spin = 0.0;
15
16 void CALLBACK display(void)
17 {
18     glClear (GL_COLOR_BUFFER_BIT);
19
20     glPushMatrix ();
21     glRotatef (spin, 0.0, 0.0, 1.0);
22     glRectf (-25.0, -25.0, 25.0, 25.0);
23     glPopMatrix ();
24
25     glFlush();
26     auxSwapBuffers();
27 }
28
29 void CALLBACK spinDisplay (void)
30 {
31     spin = spin + 2.0;
32     if (spin > 360.0)
33         spin = spin - 360.0;
34     display();
35 }
36
37 void CALLBACK startIdleFunc (AUX_EVENTREC *event)
38 {
39     auxIdleFunc(spinDisplay);
```

```
40 }
41
42 void CALLBACK stopIdleFunc (AUX_EVENTREC *event)
43 {
44     auxIdleFunc(0);
45 }
46
47 void myinit (void)
48 {
49     glClearColor (0.0, 0.0, 0.0, 1.0);
50     glColor3f (1.0, 1.0, 1.0);
51     glShadeModel (GL_FLAT);
52 }
53
54 void CALLBACK myReshape(GLsizei w, GLsizei h)
55 {
56     if (!h) return;
57     glViewport(0, 0, w, h);
58     glMatrixMode(GL_PROJECTION);
59     glLoadIdentity();
60     if (w <= h)
61         glOrtho (-50.0, 50.0, -50.0*(GLfloat)h/(GLfloat)w,
62                  50.0*(GLfloat)h/(GLfloat)w, -1.0, 1.0);
63     else
64         glOrtho (-50.0*(GLfloat)w/(GLfloat)h,
65                  50.0*(GLfloat)w/(GLfloat)h, -50.0, 50.0, -1.0, 1.0);
66     glMatrixMode(GL_MODELVIEW);
67     glLoadIdentity ();
68 }
69
70 /* Main Loop
71 * Open window with initial window size, title bar,
72 * RGBA display mode, and handle input events.
73 */
74 int main(int argc, char** argv)
75 {
76     auxInitDisplayMode (AUX_DOUBLE | AUX_RGB);
77     auxInitPosition (0, 0, 500, 500);
78     auxInitWindow ("Double Buffering");
79     myinit ();
80     auxReshapeFunc (myReshape);
81     auxIdleFunc (spinDisplay);
82     auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, startIdleFunc);
83     auxMouseFunc (AUX_RIGHTBUTTON, AUX_MOUSEDOWN, stopIdleFunc);
```

```

84 auxMainLoop(display);
85 return(0);
86 }

```

The converted program *double.for* is presented below. Note the use of common blocks rather than C global variables.

Lines 76 to 78 in the *Red Book* sample have been replaced by a few ClearWin+ calls at lines 109 to 111. %og has been replaced by %^og indicating that a call-back function, opengl_proc, has been provided to handle messages. The use of %pv means that RESIZE messages are sent to the call-back.

Mouse messages are also sent to opengl_proc, as are the special SETUP and DIRTY messages. SETUP is sent only once, just after the window has been created and its arrival precedes that of RESIZE. DIRTY is sent when the window needs repainting, for example, after RESIZE or when the window is brought to the top after being partially obscured. If you take action on the DIRTY call-back, you will not need the static option on %og.

The *Red Book* variant of the program supplies several functions to handle events. These functions are still needed and perform the same task, but they are called directly in response to ClearWin+ messages sent to opengl_proc. The reshape function myReshape(x,y) is called in response to RESIZE. display is called in response to DIRTY. The mouse handling functions startIdleFunc and stopIdleFunc are called in response to MOUSE_LEFT_CLICK and MOUSE_RIGHT_CLICK. These are registered as call-backs with GLAUX at lines 70 and 82-83.

In the *Red Book* example, the initialising call to myinit at line 79 (just after the window has been created) is handled by ClearWin+ as a message sent to opengl_proc. This message is SETUP and is always sent before the first RESIZE message. The GLAUX idle function has no direct ClearWin+ equivalent. Instead, %lw is used in the winio@ call and program execution falls through into the following idle loop. The test for ctrl<0 at line 113 in the ClearWin+ example, allows the program to detect when the window has been closed so that program termination can proceed.

```

1      SUBROUTINE display()
2      INCLUDE <opengl.ins>,nolist
3      REAL spin
4      LOGICAL do_draw
5      COMMON /double_com/do_draw,spin
6
7      CALL glClear(GL_COLOR_BUFFER_BIT)
8
9      CALL glPushMatrix()

```

```
10    CALL glRotatef(spin, 0.0, 0.0, 1.0)
11    CALL glRectf(-25.0, -25.0, 25.0, 25.0)
12    CALL glPopMatrix()
13
14    CALL glFlush()
15    CALL swap_opengl_buffers()
16    END
17
18    SUBROUTINE spinDisplay()
19    INCLUDE <clearwin.ins>,nolist
20    REAL spin
21    LOGICAL do_draw
22    COMMON /DOUBLE_COM/do_draw,spin
23
24    IF(do_draw)THEN
25        spin = spin + 2.0
26        IF(spin.GT.360.0)THEN
27            spin = spin - 360.0
28        ENDIF
29        CALL display()
30    ENDIF
31    CALL temporary_yield@()
32    END
33
34    SUBROUTINE startIdleFunc ()
35    REAL spin
36    LOGICAL do_draw
37    COMMON /DOUBLE_COM/do_draw,spin
38    do_draw=.TRUE.
39    END
40
41    SUBROUTINE stopIdleFunc ()
42    REAL spin
43    LOGICAL do_draw
44    COMMON /double_com/do_draw,spin
45    do_draw=.FALSE.
46    END
47
48    SUBROUTINE myinit ()
49    INCLUDE <opengl.ins>,nolist
50    CALL glClearColor (0.0, 0.0, 0.0, 1.0)
51    CALL glColor3f (1.0, 1.0, 1.0)
52    CALL glShadeModel (GL_FLAT)
53    END
```

```
54      SUBROUTINE myReshape(w, h)
55      INCLUDE <opengl.ins>,nolist
56      INTEGER w,h
57      DOUBLE PRECISION aspect_ratio
58      IF(h.NE.0) THEN
59          aspect_ratio=REAL(w)/h
60
61          CALL glViewport(0, 0, w, h)
62          CALL glMatrixMode(GL_PROJECTION)
63          CALL glLoadIdentity()
64          IF (w.LE.h) THEN
65              CALL glOrtho(-50d0, 50d0, -50d0/aspect_ratio,
66              &      50d0/aspect_ratio, -1d0, 1d0)
67          ELSE
68              CALL glOrtho(-50d0*aspect_ratio,
69              &      50d0*aspect_ratio, -50d0, 50d0, -1d0, 1d0)
70          ENDIF
71          CALL glMatrixMode(GL_MODELVIEW)
72          CALL glLoadIdentity ()
73          CALL display()
74      ENDIF
75
76  END
77
78
79      INTEGER FUNCTION opengl_proc()
80      INCLUDE <clearwin.ins>,nolist
81      CHARACTER*256 reason
82      INTEGER w,h
83      REASON=clearwin_string@('CALLBACK_REASON')
84      IF(reason.EQ.'SETUP')THEN
85          CALL myinit()
86      ELSE IF(reason.EQ.'RESIZE')THEN
87          w=clearwin_info@('OPENGL_WIDTH')
88          h=clearwin_info@('OPENGL_DEPTH')
89          CALL myReshape(w,h)
90      ELSE IF(reason.EQ.'MOUSE_LEFT_CLICK')THEN
91          CALL startIdleFunc()
92      ELSE IF(reason.EQ.'MOUSE_RIGHT_CLICK')THEN
93          CALL stopIdleFunc()
94      END IF
95
96      opengl_proc=2
97  END
```

```
98
99      PROGRAM double
100     INCLUDE <clearwin.ins>,nolist
101     INTEGER i
102     INTEGER opengl_proc,ctrl
103     EXTERNAL opengl_proc
104     REAL spin
105     LOGICAL do_draw
106     COMMON /double_com/do_draw,spin
107     DATA spin,do_draw/0.0,.TRUE./
108
109    i=winio@('%es%ca[Double Buffering]&')
110    i=winio@('%sp%ww[no_border]%pv%og[double]%'lw',
111    &          0,0,500,500,opengl_proc,ctrl)
112
113    WHILE(ctrl.LT.0)DO
114        CALL spinDisplay()
115    ENDWHILE
116    END
```

This program follows the structure that will be used throughout this chapter, and your attention is drawn again to the use of `opengl_proc` to call the reshape, display, initialisation and mouse handling functions.

Animation and multiple windows

OpenGL achieves animation by using two buffers, the front and the back buffer. When this, so called, double buffering is enabled, drawing (or rendering in OpenGL terminology) is directed to the back buffer. When the image is complete, you transfer it to the front buffer, or display, by calling `swap_opengl_buffers@`. This ClearWin+ routine merely remembers the current OpenGL context and so saves the programmer effort. When the back buffer is transferred to the display, it is supposed to be no longer valid so two successive swap buffer calls should produce garbage on the display. Although this does not seem to happen under Win32, the retention of the buffer should not be relied upon, as it might not be retained on other platforms.

The program below is not a *Red Book* sample. It illustrates the use of multiple windows, independently animated. This is something not possible using GLAUX.

The use of multiple OpenGL windows requires that you know the *device context* and *rendering context* for each of the windows. Changing from one window to another requires that you swap both the device context and the rendering context. Therefore, you must “enquire” them and store them away. These enquiry and swapping functions (together with some functions) are known as the WGL set. They do not form part of OpenGL, but are essential to the Windows implementation of it. Documentation for

these functions will be found in the Microsoft Developer Network help files. You can be sure that when `opengl_proc` is called, the expected device and rendering contexts are current.

The font is also set by means of a WGL call, `wglUseFontOutlines`. Note how, in the the program, a C type NULL pointer is simulated by `CORE4(0)`. This function call will only be successful if the format code `%fn` is used in the definition of the parent window.

```
*****
*          *
*      Animation using OpenGL      *
*          *
*****  
  
SUBROUTINE display()  
INCLUDE <opengl.ins>,nolist  
INCLUDE <clearwin.ins>,nolist  
INCLUDE 'danimate.ins'  
  
CALL glClear (OR(GL_COLOR_BUFFER_BIT,GL_DEPTH_BUFFER_BIT))  
CALL glMatrixMode (GL_MODELVIEW)  
CALL glLoadIdentity()  
CALL glTranslated(0d0,0d0,-10d0)  
CALL glRotated(spin,1d0,0d0,0d0)  
CALL glCallList(101)  
CALL swap_opengl_buffers()  
CALL glFlush()  
END  
  
SUBROUTINE swapContexts(window)  
INTEGER window  
INCLUDE <opengl.ins>  
INCLUDE 'danimate.ins'  
CALL wglGetCurrent(devicecontext(window),  
&                                renderingcontext(window))  
END  
  
SUBROUTINE spinSlab(window)  
INTEGER window  
INCLUDE 'danimate.ins'  
do_draw(window)=.TRUE.  
END  
  
SUBROUTINE stopSlab(window)
```

```

INTEGER window
INCLUDE 'danimate.ins'
do_draw(window)=.FALSE.
END

SUBROUTINE assemble_list
INCLUDE <clearwin.ins>,nolist
INCLUDE <opengl.ins>,nolist

REAL*4 white_colour(4),grey_colour(4),dark_grey_colour(4)
REAL*4 red_colour(4), yellow_colour(4), green_colour(4)
REAL*4 blue_colour(4), purple_colour(4), cyan_colour(4)
REAL*8 dimension,scale,d,fd,front,back
INTEGER flags,k
INTEGER hdc
LOGICAL ok
DATA white_colour/1.0,1.0,1.0,1.0/
DATA grey_colour/0.5,0.5,0.5,1.0/
DATA dark_grey_colour/0.3,0.3,0.3,1.0/
DATA red_colour/1.0,0.0,0.0,1.0/
DATA yellow_colour/1.0,1.0,0.0,1.0/
DATA green_colour/0.0,1.0,0.0,1.0/
DATA blue_colour/0.0,0.0,1.0,1.0/
DATA purple_colour/1.0,0.0,1.0,1.0/
DATA cyan_colour/0.0,1.0,1.0,1.0/
DATA front,back/-0.01d0,-0.5d0/

CALL glEnable(GL_DEPTH_TEST)
hDC=clearwin_info@('OPENGL_DEVICE_CONTEXT')
CALL glColor3f(1.0,1.0,1.0)

ok=wglUseFontOutlines(hDC, 0, 255, 1000, 0.0, 0.1,
&                               WGL_FONT_POLYGONS,core4(0))

CALL glMatrixMode (GL_MODELVIEW)
CALL glLoadIdentity()
CALL glTranslated(0d0,0d0,-10d0)

c----Clear the color and depth buffers.

CALL glClear (OR(GL_COLOR_BUFFER_BIT,GL_DEPTH_BUFFER_BIT))
dimension=2.2d0
scale=dimension*0.9d0
CALL glDisable(GL_LIGHTING)

```

```
      CALL glNewList(101,GL_COMPILE)
c----Front face
      CALL glBegin(GL_POLYGON)
      CALL glColor3fv(blue_colour)
      CALL glVertex3d(-dimension,-dimension,front)
      CALL glVertex3d(-dimension, dimension,front)
      CALL glVertex3d( dimension, dimension,front)
      CALL glVertex3d( dimension,-dimension,front)
      CALL glEnd()
c----Back face
      CALL glBegin(GL_POLYGON)
      CALL glColor3fv(yellow_colour)
      CALL glVertex3d(-dimension,-dimension,back)
      CALL glVertex3d( dimension,-dimension,back)
      CALL glVertex3d( dimension, dimension,back)
      CALL glVertex3d(-dimension, dimension,back)
      CALL glEnd()
c----Top face
      CALL glBegin(GL_POLYGON)
      CALL glColor3fv(red_colour)
      CALL glVertex3d(-dimension, dimension,front)
      CALL glVertex3d(-dimension, dimension,back)
      CALL glVertex3d( dimension, dimension,back)
      CALL glVertex3d( dimension, dimension,front)
      CALL glEnd()
c----Bottom face
      CALL glBegin(GL_POLYGON)
      CALL glColor3fv(green_colour)
      CALL glVertex3d( dimension,-dimension,front)
      CALL glVertex3d( dimension,-dimension,back)
      CALL glVertex3d(-dimension,-dimension,back)
      CALL glVertex3d(-dimension,-dimension,front)
      CALL glEnd()

c----Draw graph

      CALL glBegin(GL_LINES)
      CALL glColor3fv(white_colour)
      CALL glVertex2d(-scale,0d0)
      CALL glVertex2d(scale,0d0)
      CALL glVertex2d(0d0,-scale)
      CALL glVertex2d(0d0,scale)
      k=-10
```

```

      WHILE(k .LE. 10) DO
        CALL glVertex2d((scale/10)*k,0.0d0)
        CALL glVertex2d((scale/10)*k,0.1d0)
        CALL glVertex2d(0.0d0,(scale/10)*k)
        CALL glVertex2d(0.1d0,(scale/10)*k)
        k=k+1
      ENDWHILE
      CALL glEnd()
      CALL glColor3fv(red_colour)
      CALL glBegin(GL_LINE_STRIP)
      d=-10d0
      WHILE(d .LT. 10d0)DO
        fd=5*(d/10)**3+3*(d/10)**2-d/10
        CALL glVertex2d(d*scale/10,fd*scale/10)
        d=d+0.05d0
      ENDWHILE
      CALL glEnd()
      CALL glColor3fv(green_colour)
      CALL glBegin(GL_LINE_STRIP)
      d=-10d0
      WHILE(d .LT. 10)DO
        fd=5*(d/10)**3+6*(d/10)**2-d/10
        CALL glVertex2d(d*scale/10,fd*scale/10)
        d=d+0.05d0
      ENDWHILE
      CALL glEnd()
      CALL glColor3fv(red_colour)
      CALL glListBase(1000)
      CALL glTranslated(0.15d0,scale*0.95,0d0)
      CALL glScaled(0.2d0,0.2d0,0.2d0)
      CALL glCallLists(5, GL_UNSIGNED_BYTE, '+10.0')
      CALL glEndList()
    END

    SUBROUTINE myinit()
    CALL assemble_list()
    END

    SUBROUTINE myreshape(w,h)
    INCLUDE <opengl.ins>,nolist
    INTEGER w
    INTEGER h
    DOUBLE PRECISION aspect_ratio
  
```

```
IF(h.NE.0)THEN
    aspect_ratio=dble(w)/h
    CALL glMatrixMode(GL_PROJECTION)
    CALL glLoadIdentity()
    CALL gluPerspective(30.0d0,aspect_ratio,1d0,15d0)
    CALL glViewport(0,0,w,h)
ENDIF

END

INTEGER FUNCTION opengl_proc1()
INCLUDE <clearwin.ins>,nolist
INCLUDE <opengl.ins>,nolist
INCLUDE 'danimate.ins',nolist
INTEGER w,h
CHARACTER*256 reason
reason=clearwin_string@('CALLBACK_REASON')

IF(reason.EQ.'SETUP')THEN
    CALL myinit()
ELSEIF(reason.EQ.'RESIZE')THEN
    w=clearwin_info@('OPENGL_WIDTH')
    h=clearwin_info@('OPENGL_DEPTH')
    CALL myreshape(w,h)
ELSEIF(reason.EQ.'DIRTY')THEN
    CALL display()
    window1_active=.true.
ELSEIf(reason.EQ.'MOUSE_LEFT_CLICK')THEN
    CALL spinSlab(1)
ELSEIf(reason.EQ.'MOUSE_RIGHT_CLICK')THEN
    CALL stopSlab(1)
ENDIF

opengl_proc1=2
END

INTEGER FUNCTION opengl_proc2()
INCLUDE <clearwin.ins>,nolist
INCLUDE <opengl.ins>,nolist
INCLUDE 'danimate.ins'
INTEGER w,h
CHARACTER*256 reason
reason=clearwin_string@('CALLBACK_REASON')
```

```

IF(reason.EQ.'SETUP')THEN
    CALL myinit()
ELSEIF(reason.EQ.'RESIZE')THEN
    w=clearwin_info@('OPENGL_WIDTH')
    h=clearwin_info@('OPENGL_DEPTH')
    CALL myreshape(w,h)
ELSEIF(reason.EQ.'DIRTY')THEN
    CALL display()
    window2_active=.true.
ELSEIF(reason.EQ.'MOUSE_LEFT_CLICK')THEN
    CALL spinSlab(2)
ELSEIF(reason.EQ.'MOUSE_RIGHT_CLICK')THEN
    CALL stopSlab(2)
ENDIF
CALL temporary_yield@()
opengl_proc2=2
END

PROGRAM Animate
INCLUDE <opengl.ins>,nolist
INCLUDE <clearwin.ins>,nolist
DOUBLE PRECISION spin_array(2)
INTEGER i,window1,window2
INTEGER opengl_procl, opengl_proc2
EXTERNAL opengl_procl,opengl_proc2
INCLUDE 'danimate.ins'

do_draw(1)=.true.
do_draw(2)=.true.
devicecontext(1)=0
devicecontext(2)=0
renderingcontext(1)=0
renderingcontext(2)=0
spin=0d0
spin_array(1)=0d0
spin_array(2)=0d0
window1_active=.false.
window2_active=.false.

c----Create a couple of OpenGL Windows
c----Window 1
i=wini@('%es%ca[Rotating Slab]&')
i=wini@('%fn[Times New Roman]%ts&',3.0d0)
i=wini@('%sp%ww[no_border]%pv%^og[double,depth16]%'lw'.

```

```
&          0,0,350,350,opengl_proc1,window1)

c----Get and save hDC and the Rendering Contexts

devicecontext(1)=wglGetCurrentDC()
renderingcontext(1)=wglGetCurrentContext()

c----Create a couple of OpenGL Windows
c----Window 2
    i=winio@('%es%ca[Rotating Slab]&')
    i=winio@('%fn[Times New Roman]%ts&',3.0d0)
    i=winio@('%sp%ww[no_border]%pv%og[double,depth16]%'lw',
&           400,0,350,350,opengl_proc2,window2)

c----Get and save hDC and the Rendering Contexts

devicecontext(2)=wglGetCurrentDC()
renderingcontext(2)=wglGetCurrentContext()

do_draw(1)=.true.
do_draw(2)=.true.

WHILE(window1.LT.0 .OR. window2.LT.0)DO
    IF(window1.LT.0)THEN
        IF(do_draw(1))THEN
            spin_array(1)=spin_array(1)+2d0
            spin=spin_array(1)
            CALL swapContexts(1)
            CALL display()
        ENDIF
    ENDIF
    IF(window2.LT.0)THEN
        IF(do_draw(2))THEN
            spin_array(2)=spin_array(2)+2d0
            spin=spin_array(2)
            CALL swapContexts(2)
            CALL display()
        ENDIF
    ENDIF
    CALL temporary_yield@()
ENDWHILE
END
```

Using a printer with OpenGL

When using OpenGL, the quality of the printer output and the rendering of colour are especially important and it is recommended that you check your printer driver settings, ensuring (for example) that you are using the highest setting for the number of dots per inch.

The following functions are provided to enable you to print OpenGL images.

Function	See page
OPEN_GL_PRINTER@	381
OPEN_GL_PRINTER1@	381
PRINT_OPENGL_IMAGE@	389

References

- Woo, Mason, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*. Reading, MA: Addison Wesley Longman Inc. 1997. ISBN 0-201-46138-2.
- OpenGL Architecture Review Board; Renate Kempf and Chris Frazier, editors. *OpenGL Reference Manual. The Official Reference Document for OpenGL, Version 1.1*. Reading, MA: Addison Wesley Longman Inc. 1996. ISBN 0-201-46140-4.
- Fosner, Ron. *OpenGL Programming for Windows95 and Windows NT*. Reading, MA: Addison Wesley Longman Inc. 1996.
- Wright Jr., Richard S. and Michael Sweet. *OpenGL Superbible: The Complete Guide to OpenGL Programming for Windows NT and Windows 95*. Waite Group Press. 1996.
- Angel, Edward. *Interactive Computer Graphics: A top-down approach with OpenGL*. Reading, MA: Addison Wesley Longman Inc. 1997. ISBN 0-201-85571-2.

Online information

The reference pages for OpenGL are available at:

<http://www.opengl.org/>

<http://www.sgi.com/software/opengl/>

<http://www.digital.com/pub/opengl/>
<http://msdn.microsoft.com/>

The OpenGL World Wide Web Centre contains links to a variety of articles, the OpenGL specification, and the OpenGL usenet mailing list.

Other items are available to members of the Microsoft Developer Network. Online versions of technical articles are also available on the Microsoft Developer homepage.

17.

SIMPLEPLOT

Introduction

SIMPLEPLOT is a library of Fortran subroutines designed and marketed by BUSS Ltd.¹ to produce pictures of data with speed and accuracy. The basic SIMPLEPLOT library offers a wide range of facilities for drawing Cartesian and polar graphs. Extensions are available for contouring and surface pictures, presentation graphics for high quality output and representations of 4D data.

ClearWin+ provides access to SIMPLEPLOT via a dynamic link library (*simple.dll*) that is distributed with Salford compilers. Access to this library is available by using the `wini o@` function together with the `%pl` format code. `%pl` is designed to be similar to `%gr`.

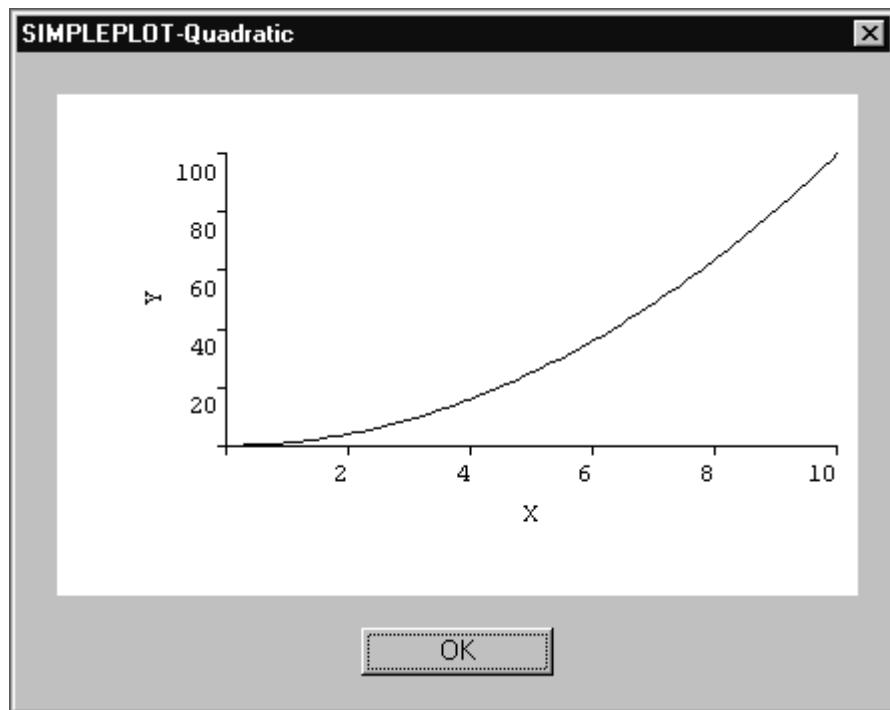
In this chapter we shall look at three elementary programs that illustrate the essential features of the interface between ClearWin+ and SIMPLEPLOT. The first two examples illustrate an *automatic* mode of access. This is by far the easiest way to access the SIMPLEPLOT library from ClearWin+. The final example shows you how to make direct calls to the SIMPLEPLOT library. Further information about SIMPLEPLOT can be found in the documentation provided by BUSS Ltd.

¹ Bradford University Software Services: Web site <http://www.buss.co.uk/buss/>

A simple example

In our first example we create a window and draw a quadratic curve within it.

```
c--Simple1.for
    WINAPP
    INTEGER N,winio@,i
    PARAMETER(N=11)
    REAL*8 x(N),y(N)
c--Create the data to be plotted.
    DO i=1,N
        x(i)=i-1
        y(i)=x(i)**2
    ENDDO
c--Plot the data.
    i=winio@('%ca[SIMPLEPLOT-Quadratic]%bg[grey]&')
    i=winio@('%pl[x_array]&',400,250,N,x,y)
    i=winio@('%ff%nl%cn%tt[OK]')
END
```



%pl is like %gr and takes the basic form:

```
winio@('%pl[options]',width,height)
```

width and *height* represent the pixel dimensions of a SIMPLEPLOT graphics region within the window. Some of the options take additional arguments. In this example, the option called *x_array* takes three arguments namely the number of points *N* followed by an array of x co-ordinates and an array of y co-ordinates.

Following the usual ClearWin+ convention, all real data is passed as DOUBLE PRECISION values. However, SIMPLEPLOT uses single precision, so data values should not exceed the single precision range.

%pl options

The following options are available for use with %pl in automatic mode:

Option	Description
SCALE=LINEAR	Specifies a Cartesian plot with linear x and y data. This is the default.
SCALE=LOG_LINEAR	Specifies a Cartesian plot with linear x and log y data.
SCALE=LINEAR_LOG	Specifies a Cartesian plot with log x and linear y data.
SCALE=LOG_LOG	Specifies a Cartesian plot with log x and log y data.
N_GRAPHS=<n>	Specifies the number of graphs to be drawn on the same axes (the default is N_GRAPHS=1)
X_AXIS=<text>	Specifies text for x-axis label (the default is X_AXIS=X)
Y_AXIS=<text>	Specifies text for y-axis label (the default is Y_AXIS=Y)
TITLE=<text>	Specifies the title for graph (by default there is no title).
X_ARRAY	Specifies that an array of x values is supplied. If this option is not specified then the first x value together with a constant increment must be supplied.
Y_MIN=<value>	Specifies the minimum y value. By default this is computed from the data.
Y_MAX=<value>	Specifies the maximum Y value. By default this is computed from the data.
COLOUR=<name>	Specifies the colour of the plot (red, black, blue, etc.). The first use of COLOUR refers to the first plot, the second to the second plot and so on.

COLOR=<name>	The same as COLOUR.
STYLE=<n>	STYLE=0 gives a smooth plot. With STYLE=1, points are plotted using small squares and the relevant colour. The first use of STYLE refers to the first plot, the second to the second plot and so on.

Note that where text is supplied for the label of an axis or for the title, this text is terminated by the next comma, space or square bracket. In order to include commas, spaces or square brackets in the text you should enclose it in single or double quotation marks.

Here is a program that uses some of these options.

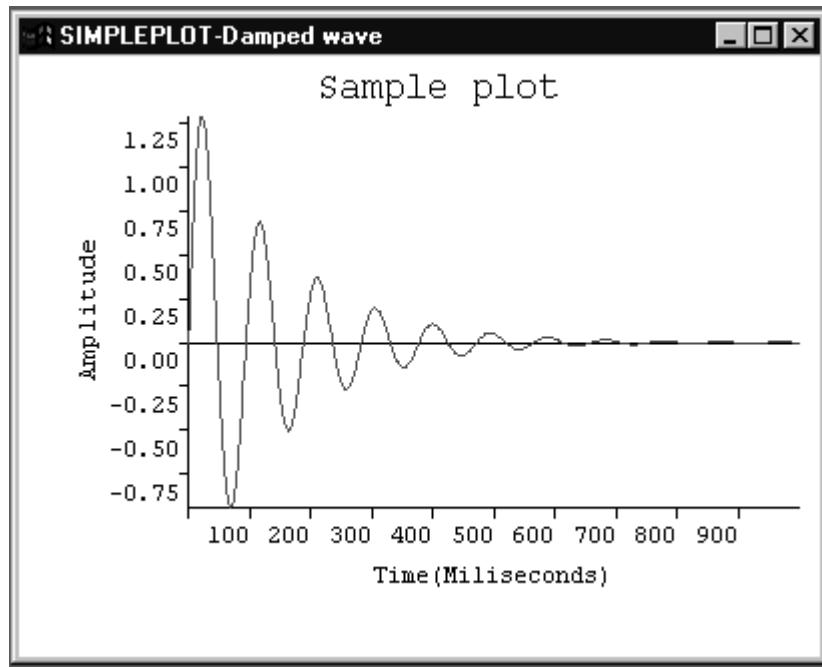
```
c---Simple2.for
      WINAPP
      INTEGER i,x,winio@,N
      PARAMETER(N=1000)
      REAL*8 p1,p2,p3,y(N)
c---Prepare the data.
      p1=1.5
      p2=150.0
      p3=15
      x=0
      DO i=1,N
          y(i)=p1*sin(x/p3)*exp(-x/p2)
          x=x+1
      ENDDO
c---Display a window containing the graph.
      i=winio@(%ww[no_border]%ca[SIMPLEPLOT-Damped wave]%pv&)
      i=winio@(%pl[x_axis=Time(Milliseconds),
      +//y_axis=Amplitude,title="Sample plot",colour=red],
      + 400,300,N,0.0D0,1.0D0,y)
      END
```

In this case, since the option X_ARRAY is not used, the argument list for %pl includes the initial value of x (0.0D0) and the increment for x (1.0D0). Note also that %ww and %pv are used with %pl. As a result, the window can be sized and the graph is redrawn to fill the window. The output is illustrated below.

Here is a fragment of code that shows how to plot two graphs on one set of axes.

```
i=winio@(%pl[X_AXIS=Time(Milliseconds),Y_AXIS=Amplitude,//%
+ 'TITLE="Sample",N_GRAPHS=2,COLOUR=red,COLOUR=blue,X_ARRAY'],
+ 300,300,N,xarr,yarr1,yarr2)
```

There is one x-array for the two y-arrays. The first graph would be red in colour and the second blue.



Note that the arrays of data supplied to %pl usually need to be available for as long as the window is displayed. In particular, the data will be re-plotted if the window is resized. It is also possible to call the routine simpleplot_redraw@ to force all such plots to be re-drawn using whatever values are currently present in the arrays. simpleplot_redraw@ is a subroutine that takes no arguments.

Direct calls to SIMPLEPLOT

As an alternative to the automatic mode described above, %pl can take the form:

```
winio@('%pl[user_drawn]',width,height)
```

width and *height* give the pixel dimensions of a SIMPLEPLOT graphics region. There are no other arguments to go with %pl. Direct calls can be made to the SIMPLEPLOT library from within a %sc call-back. Alternatively you could use %lw and put the library calls after the set of winio@ statements that describe the window.

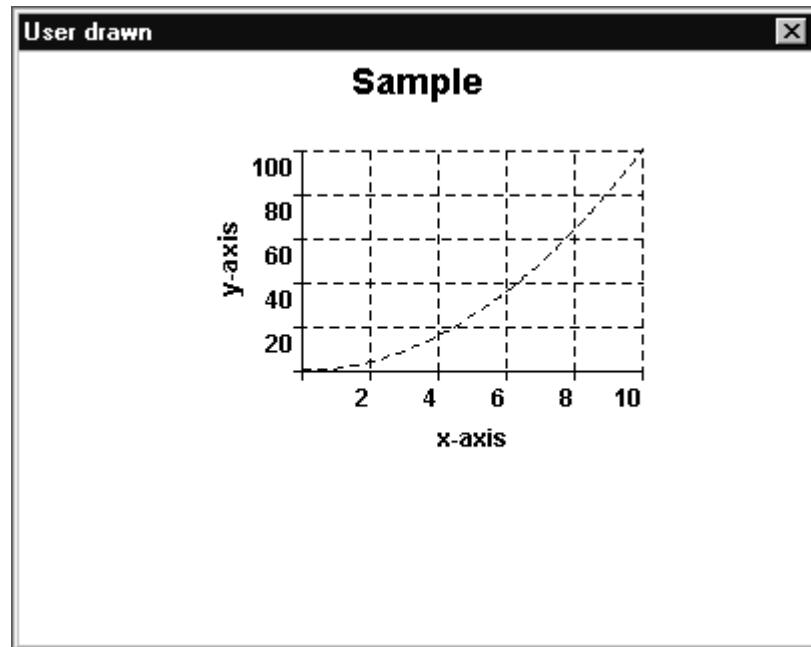
Here is a simple example that illustrates this approach.

```
c--Simple3.for
WINAPP
INTEGER winio@,i,StartCB
EXTERNAL StartCB
i=winio@('%sy[no_border]%'ca[User drawn]&')
i=winio@('%pl[user_drawn]&',400,300)
i=winio@('%sc',StartCB)
END
c--The start-up callback
INTEGER FUNCTION StartCB()
INTEGER N,i
PARAMETER(N=101)
REAL x(N),y(N)
c--Create the data.
DO i=1,N
    x(i)=i-1
    y(i)=x(i)**2
ENDDO
c--Call SIMPLEPLOT routines to plot the graph.
CALL INITSP
CALL PAGE(12.0,10.0)
CALL CHSET(-11)
CALL NEWPAG
CALL TITLE7('Higher','Centre','Sample')
CALL SCALES(0.0,10.0,1,0.0,100.0,1)
CALL AXGRID('*Cartesian',1,-1)
CALL NEWPIC
CALL AXIS7('XCartesian','x-axis')
CALL AXIS7('YCartesian','y-axis')
CALL BRKNCV(x,y,N,-1)
StartCB=1
END
```

When you link this program you must link with *simple.dll*.

Notice that this window cannot be re-sized. As a result the x and y arrays for the data can be local to the start-up call-back function. Further information about the SIMPLEPLOT routines that are used in this example can be found in the documentation provided by BUSS Ltd.²

² Web site <http://www.buss.co.uk/buss/>



18.

Using the printer

Introduction

One of the advantages of the Windows operating system is its built-in support for a wide range of printers. ClearWin+ caters for three different types of printer output:

- Simple text output to the printer.
- HTML text output.
- Graphics output.

Before you can send output to the printer it is usually necessary to prompt the user to select the printer to use (this also gives users the opportunity to cancel the print request). There are several methods to obtain this information each of which is described below.

The resolution of a printer varies from one machine to another, but typically printers offer a much finer resolution than even a high quality graphics driver. This means that screen images must be re-drawn for a printer after determining the resolution of the printer selected by the user. If you are using bitmap images as part of your printer output, you may need to use a higher resolution image on the printer than on the screen.

If you send simple text to the printer it will look rather plain because a single font will be used. However, this is sometimes useful for debugging purposes. It is almost as easy to send HTML output to the printer, and this is the recommended method for producing professional printed output from ClearWin+.

For printed output using OpenGL see page 179.

Selecting a printer

Using a standard call-back function

In most Windows applications, such as Word for Windows, a standard printer dialog box is produced in response to a user button press or menu activation. Several ClearWin+ standard call-back functions are supplied to perform this task, each of which takes another call-back as an additional argument. The standard printer dialog is automatically displayed and the call-back will contain your code to write or draw to the printer. This call-back will only be called if the user selects a printer from the dialog (as opposed to pressing the Cancel button).

PRINTER_OPEN	Takes an integer Fortran unit number and another callback function as arguments. Displays a dialog box to select the printer and connects it to the given Fortran unit number.
PRINTER_OPEN1	This also takes a Fortran unit number and another callback function as arguments. It uses the printer previously selected using PRINTER_OPEN.
HTML_PRINTER_OPEN	This takes a Fortran unit number and another call-back function. It displays a dialog box to select the printer and connects it to the given Fortran unit number.
HTML_PRINTER_OPEN1	This also takes a Fortran unit number and another call-back function. It uses the previously selected printer.
GPRINTER_OPEN	This takes another call-back as argument and displays a dialog box to select the printer. The printer is connected as the current <i>drawing surface</i> for the duration of the call-back, which should perform the required graphics operations.
GPRINTER_OPEN1	This operates in the same way as GPRINTER_OPEN, except that a previously opened printer is used.

For example, here is a program that will respond to the button press by prompting the user for a printer and printing ‘Hello’.

```

EXTERNAL printer
i=winio@('Press this button to print a page: ')
i=winio@('%^bt[Print]&','PRINTER_OPEN',7,printer)
i=winio@('%2n1%cn%bt[OK]')
END
C-----
INTEGER FUNCTION printer()
```

```

      WRITE(7,*)'Hello'
      CLOSE(7)
      printer=2
      END

```

Notice that the unit is closed before the call-back function exits. Typically, the call-back function should return a value of 2 to indicate that the associated window should stay open without a screen update. However, other call-back return values may be used as described on page 205.

Using the `HTML_PRINTER_OPEN` or `HTML_PRINTER_OPEN1` standard call-backs it is easy to generate pleasantly formatted text output to the printer. This mechanism uses the same subset of HTML as that used by `%ht` except that one mark-up `<PB>` is added to set a page break.. Here is a simple example:

```

WINAPP
INTEGER i,winio@
EXTERNAL test
i=winio@('%sc','HTML_PRINTER_OPEN',7,test)
END
C---
INTEGER FUNCTION test()
WRITE(7,'(a)')'<h1>The output</h1>'
WRITE(7,'(a)')'This is <i>Italic</i>. <p>'
WRITE(7,'(a)')'And this is <b>Bold</b>. <p>'
CLOSE(7)
test=2
END

```

Under Win16, it is necessary to call `INIT_PRINTER_HTML@()`.

The standard call-backs `GPRINTER_OPEN` and `GPRINTER_OPEN1` do not connect the printer to a Fortran unit number, but rather they create a *drawing surface* that becomes the current *drawing surface*. The call-back that is an additional argument to these standard call-backs is used to draw to this surface. The current *drawing surface* reverts to the previous surface (if any) at the end of the call-back. An example is given on page 194.

Using `OPEN_PRINTER@` etc. to create a drawing surface

Alternatively, you can use explicit function calls to create a *drawing surface* and to attach it to a printer for graphics drawing:

<code>OPEN_PRINTER@</code>	Displays the standard printer dialog to enable graphics output to be sent to a printer.
<code>OPEN_PRINTER1@</code>	Re-opens a graphics printer using settings from a previous call to <code>OPEN_PRINTER@</code> .

OPEN_PRINTER_TO_FILE@	Displays the standard printer dialog to enable graphics output to be sent to a file.
OPEN_PRINTER1_TO_FILE@	Sends graphics output to a file supplied as an argument.

Each of these functions returns zero if the user cancels the operation, and each takes a handle argument as input that is used to switch between *drawing surfaces* using select_graphics_object@. An example is given on page 195.

Customising the printer dialog box

ClearWin+ uses the standard Windows dialog box to interrogate the user, in each of the above approaches. This has the advantage that the style of the box will take on the 'look and feel' of the operating system in use and will be familiar to your users.

In order to change the default behaviour of the printer dialog box you can call printer_dialog_options@ in order to specify the information that is to be requested in the printer dialog box when it opened using PRINTER_OPEN etc. or open_printer@ etc. (also you can obtain the values selected by the user with calls to clearwin_info@).

printer_dialog_options@ takes nine integer arguments referred to as A1 to A9. The following table indicates the changes that are possible:

Feature	CLEARWIN_INFO@ parameter for result	Default behaviour	Remarks
Print to file	-	Available, initially not checked	If a printer is selected with this option checked, the user will be prompted for a file name. No programming action is required to support this feature. This box will be hidden if A2 is non-zero, and will be initially checked if A9 is non-zero.
Selection	PRINTER_SELECTION	Available, initially not checked	This is a radio button which is used to indicate that only selected material is to be printed. It is up to you to read the parameter and act on it as required. This box will be

Page numbers	PRINTER_FIRST_PAGE PRINTER_LAST_PAGE	Available, set to 1,1.	greyed out (not hidden) if A3 is non-zero, and will be initially selected if A6 non-zero.
Collate	PRINTER_COLLATE	Available	This enables a range of pages to be selected in various ways. The control is disabled if A2 is non-zero. If all pages are selected the range is returned as 1-32767. If A5 is non-zero the range selection option will be initially selected. If A8 is non-zero the all-pages option will be initially selected.
Copies	PRINTER_COPIES	Available	This control is always available. Some printer drivers support this option explicitly and the program is not informed of the selection because no action is required. If the parameter is set, an appropriate response is required from the program. If A7 is non-zero the collate box is initially checked.
Default warning	-	Will occur	This control is always available. Some printer drivers support this option explicitly and the program is not informed of the selection because no action is required. If the number of copies is greater than 1, the program should reproduce the output.
<hr/>			
Also <code>clearwin_info@('PRINTER_PAGENUMBERS')</code> returns 1 if the 'Page Range' is set to 'Pages'.			

At most one of A5, A6, A8 should be set non-zero, as they select mutually exclusive options. These defaults work for both built in call-backs and explicit routine calls.

Four other `clearwin_info@` parameters are also used to control the initial appearance of the printer dialog box. These parameters are set using `set_clearwin_info@`. The parameters are:

```
PRINTER_DEFAULT_MIN_PAGE
PRINTER_DEFAULT_MAX_PAGE
PRINTER_DEFAULT_FROM_PAGE
PRINTER_DEFAULT_TO_PAGE
```

In some situations it may be necessary to set these parameters in a multiple call-back, for example:

```
i=winio@('%^bt[Print]', '+', set_printer_info, 'PRINTER_OPEN', 10)
where set_printer_info is a user-defined function that makes appropriate calls to
set_clearwin_info@.
```

Printer graphics

Here is a simple program that uses the standard call-back `GPRINTER_OPEN` to respond to a button press and draws a diagonal line across the entire page. `GPRINTER_OPEN` creates a *drawing surface* and an additional call-back argument is used to contain code to draw to this surface.

```
WINAPP
INTEGER i,winio@
EXTERNAL test
i=winio@('%^bt[Open]', 'GPRINTER_OPEN', test)
END
C---
INTEGER FUNCTION test()
INCLUDE <windows.ins>
INTEGER xmax,ymax
CALL get_graphical_resolution@(xmax,ymax)
CALL draw_line_between@(0,0,xmax-1,ymax-1,15)
test=2
END
```

Notice that the call to `get_graphical_resolution@` will work for all types of *drawing surfaces*. This means that the same code can be used to generate graphics output to the screen and to the printer.

By using `open_printer@` etc., it is possible to have two *drawing surfaces*, one attached to the screen and the other to a graphics printer or plotter. In the following example one handle is attached to a `%gr` region (a screen *drawing surface*) whilst a second handle is attached to a graphics printer *drawing surface*.

```
INTEGER ctrl,hnd1,hnd2
DATA hnd1,hnd2/1,2/
i=winio@('%`gr&',400L,300L,hnd1)
i=winio@('%ww%lw',ctrl)
j=select_graphics_object@(hnd1)
CALL draw_line_between@(100,100,300,200,15)
k=open_printer@(hnd2)
j=select_graphics_object@(hnd2)
CALL draw_filled_ellipse@(200,150,75,50,2)
k=close_printer@(hnd2)
```

As usual, `open_printer@` generates a standard “Open Printer” dialog box from which the user selects a graphics printer or plotter. Subsequent calls to **ClearWin+** drawing routines are written to this *drawing surface* and the output is generated when the **ClearWin+** routine `close_printer@` or the routine `new_page@` is called.

Notes:

- `open_printer@` etc. may be used independently of `winio@`.
- In the above example, `j` and `k` can be used to test for the success of the associated function calls.

Using a metafile to print multiple copies

Here is another sample program which (like the above example) shows how to draw to a screen *drawing surface* in parallel with output to a graphics printer or plotter. In this case the image is scaled according to which device is used. Multiple copies are printed by calling `do_copies@` but this routine requires a metafile recording of the current image before it is called.

```
INCLUDE <windows.ins>
REAL scaling
INTEGER i,wc,res,ncopies
INTEGER screen_height,printer_height
INTEGER screen_width,printer_width
INTEGER screen_id,printer_id
EXTERNAL graphics
COMMON scaling,screen_id,printer_id
CALL set_rgb_colours_default@(1)
```

```

        scaling=1.0
c--- Give id numbers to the screen and printer
screen_id=1
printer_id=2
c---
i=winio@('%ww%ca[Salford graphics]&')
i=winio@('%`gr[black]&',640,480,screen_id)
i=winio@('%`bt[OK]%'lw',wc)
CALL graphics()
CALL get_graphical_resolution@(screen_width,
+           screen_height)
c--- Select printer and start document
res=open_printer@(printer_id)
IF(res.NE.0)THEN
    CALL use_rgb_colours@(printer_id,.true.)
    CALL get_graphical_resolution@(printer_width,
+                               printer_height)
c--- Scaling factor for printer
scaling=printer_width/screen_width
c--- Open metafile to record printer drawing
CALL open_metafile@(printer_id)
CALL graphics()
ncopies=clearwin_info@('printer_copies')
c--- Stop recording
CALL close_metafile@(printer_id)
ncopies=ncopies-1
c--- Make copies of the page
CALL do_copies@(printer_id,ncopies)
c--- Print document
res=close_printer@(printer_id)
CALL select_graphics_object@(screen_id)
scaling=1.0
ENDIF
END

SUBROUTINE graphics()
INCLUDE <windows.ins>
INTEGER x(10),y(10)
INTEGER hpol,z,xa,xb,err
REAL scaling
INTEGER screen_id,printer_id
COMMON scaling,screen_id,printer_id
CALL set_line_width@(3)
c--- Select font

```

```
CALL set_text_attribute@(101,3*scaling,0,0)
xa=300*scaling
xb=350*scaling
CALL draw_characters@('HELLO',xa,xb,RGB@(255,255,255))
CALL set_line_width@(1)
z=200*scaling
CALL draw_line_between@(0,0,z,z,RGB@(255,0,0))
CALL draw_line_between@(0,z,z,0,RGB@(0,255,0))
c--- Ellipse
z=100*scaling
xa=80*scaling
xb=40*scaling
CALL draw_ellipse@(z,z,xa,xb,RGB@(255,0,0))
xa=60*scaling
CALL draw_filled_ellipse@(z,z,xa,xb,RGB@(0,0,255))
c--- Polyline
x(1)=230*scaling
y(1)=20*scaling
c--- .... Code for x(2..4) and y(2..4)
x(5)=x(1)
y(5)=y(1)
CALL draw_polyline@(x,y,5,RGB@(255,0,0))
c--- Polygon
x(1)=30*scaling
y(1)=250*scaling
c--- .... Code for x(2..4) and y(2..4)
x(5)=30*scaling
y(5)=250*scaling
CALL draw_filled_polygon@(x,y,5,RGB@(255,0,0))
END
```


19.

Hypertext windows

Hypertext %N.Mht

Most programmers and many users will be familiar with the use of Web Browsers such as Internet Explorer. Such browsers display information that has been encoded as ASCII files containing Hypertext Markup Language (HTML). A subset of HTML forms the basis for ClearWin+ hypertext windows.

Hypertext windows provide a way of building very easy to use GUI applications. A hypertext window contains text with hypertext links, that enable users to switch between pages (by clicking on highlighted text and images) or to initiate parts of your program. A hypertext window is created using %ht and can be surrounded by other controls as required.

For example, the format %70.20ht[introduction] would create a hypertext window of a width sufficient to display 70 average width characters and 20 rows. The text would be supplied by a prior call to add_hypertext_resource@ with the name of a hypertext resource, say ‘start’, containing the required initial topic called ‘introduction’. This would be defined as a resource thus:

```
start HYPERTEXT "my file.htm"
```

You must use the Salford resource compiler SRC for hypertext files, rather than the Microsoft compiler RC. By default, SRC is automatically invoked when a program containing the WINAPP directive is compiled.

The file called *browse.htm*, that is built into the ClearWin+ browser program, is available as an example program and is an excellent illustration of a hypertext file. Note that hypertext resources are stored in a compressed format, which is not accessible to your users if they choose to binary edit your program file.

The HTML text is displayed in the window that is both left and right justified with scroll bars if necessary. The %ht format may be preceded by a pivot format (%pv) so that the text area changes size as the window is adjusted by the user.

Mark-up codes are contained within diamond brackets <>. For example, a paragraph end is marked using <P>. Mark-up codes are case insensitive. Some codes define a condition that pertains until cancelled by a corresponding mark-up starting with a '/' character. For example, bold face text is preceded by and followed by . Special characters (such as the diamond brackets used to define mark-up codes) are encoded as follows:

< or < produces '<'
> or > produces '>'
& or & produces '&'

Hypertext files may contain portions of text that are included conditionally, under program control. This is called *conditional hypertext*. The browse program uses this feature to change the text depending on the programming language selected from the corresponding menu item.

The following HTML codes are currently available in ClearWin+:

<TITLE> ... </TITLE>

This supplies a title to a topic. The title will be placed in the caption of the format window using %ht.

<H1> ... </H1>

Supplies a principle heading to the text. The text will be enlarged and centred. Although the other heading styles (H2 .. H6) may be used, they currently operate in the same way as H1. This may change in future versions of the software.

<HTML> ... </HTML>

In standard HTML these codes are almost redundant. They optionally enclose the entire text. However they are vital for ClearWin+ hypertext as they are used to delimit separate topics. Each topic should be surrounded by these tags and contain a DOC tag.

<DOC name="doc_name">

This defines the topic name, and has no analogue in non-ClearWin+ HTML. Each topic should have a distinct name enclosed in double quotation marks as shown.

** ... **

This is an HTML anchor that encloses text (or an image) that will be highlighted in blue and will react to a left mouse click. Note that standard HTML anchors have a somewhat more general syntax. The name will usually be another topic name (defined by DOC). If the name does not correspond to a topic and a call-back function is supplied with %ht, then the call-back function will be called. The

function can determine the text on the anchor by calling `clearwin_string@('CURRENT_TEXT_ITEM')`. In this way, it is possible to execute code in response to a mouse click.

Hypertext can also contain Uniform Resource Locators (URL's) that link to the web, for example `` or to local files, for example ``. Local files must have the .HTM or .HTML extension. Alternatively, if the file has a .EXE extension, it will be executed (see also `USE_URL@` and `INTERNET_LINK` standard call-back). By using file URLs, it is possible to create complex webs of inter-related documents without having all the information stored in the executable file.

<P>

Marks the end of paragraph. Subsequent text will start on a fresh line with a small gap separating it from the previous text.

**
**

Forces a line break with no extra gap.

<HR>

Rules a horizontal line across the page.

** ... **

Creates bold text.

<U> ... </U>

Creates underlined text.

<I> ... </I>

Creates italic text.

<RED> ... </RED>

Sets the text colour. Other available colours are **BLUE**, **BLACK**, **GREEN**, **WHITE**, and **YELLOW**. Hypertext links are coloured blue by default so you may want to avoid the use of this text colour (although the colour of a link can be changed by nesting a colour change inside the link). These are not standard HTML mark-ups.

Using this mark-up you can insert a bitmap into your text. *name* should be the name of a BITMAP resource you have added in the list of included resources (quotation marks are required), and *align_option* should be one of TOP, MIDDLE, BOTTOM. The alignment specification is optional and the default is BOTTOM. This determines the way in which the image is aligned with the surrounding text. By embedding an image in an anchor construct you can display an image that responds to mouse clicks. GIF files may also be used in this context (see %gi, page 243, for further information).

** ... **

Text embedded between these mark-up codes is considered to form an unordered list. Each list item is preceded by a mark-up and will start on a fresh line with a bullet mark. Lists embedded in lists will nest to the right.

 ...

Produces an ordered list numbered from 1.

<IF name> ... </IF> - See *conditional hypertext* below.

<PB>

Sets a page break for use with `HTML_PRINTER_OPEN`, see page 191.

A format window can contain at most one hypertext control. It may also contain other controls, menus, etc. as desired. It is, however, possible to embed several child windows, each containing hypertext, within a single parent window. Only one hypertext control per window is allowed because the window provides certain services to the control. Thus the containing window will be displayed with vertical scroll bars if the hypertext is too long for the space provided. Note that this happens entirely automatically. It is an error to attempt to attach vertical scroll-bars (%vs) to a window containing hypertext. Horizontal scroll bars are never needed, since hypertext is always adjusted to fit the width available.

A standard call-back function 'PREVIOUS_TEXT' is available within a window containing hypertext. This would normally be attached to a menu item and causes the hypertext window to 'go back' one topic. A menu item attached to this call-back will be automatically greyed if the hypertext is at the top level.

A call-back function 'TEXT_HISTORY' is also available. When invoked it displays a history box associated with a hypertext control in the window. The user can choose to switch to any previously viewed hypertext topic. It is an error to invoke this call-back from a window that does not contain hypertext.

By default, hypertext windows are given a grey background, although this can be changed using %`bg. In most cases it is desirable to give the parent window a matching grey background using %bg[GREY].

The title of a window containing hypertext will consist of any explicit title (%ca) followed by the title of the current hypertext topic (as supplied by the <TITLE>) mark-up separated by a hyphen (-).

If there are syntax errors in your mark-up codes, such errors will be reported at run time rather than when the resource is compiled. A error dialog box will appear showing the offending code together with a few lines of the text that follows it.

If a call-back function is supplied to %ht, this function will be called when a jump is made to a topic that has not been defined. Alternatively, a grave accent (`) is supplied in order to ensure that the call-back will be called the first time and each time the hypertext topic changes. The call-back function can then interrogate the following `clearwin_string@` parameters:

```
CURRENT_TEXT_ITEM
CURRENT_TEXT_DOCUMENT
CURRENT_TEXT_TITLE
```

Equation character strings, that follow the rules for %eq (see page 100), may also be used in hypertext documents. Equation strings are embedded between <equation> and </equation>. The string between these delimiters is treated as a pure equation - no mark-up codes are recognized in this context. So, for example, the '<' symbol should be written as it is.

Conditional Hypertext

The ClearWin+ BROWSER program uses conditional hypertext to display different information depending on the programming language selected.

For example:

Call the <if Fortran> WINIO@ <else> winio </if> function to display a window.

This uses a named integer parameter with the name 'Fortran', set up using the set_clearwin_info@ function. The <if> clause is selected if the parameter is non-zero. The <else> clause is optional. Conditional constructs can be nested, and may contain arbitrary amounts of other mark-up codes and text.

If a call-back function changes the value of a parameter referenced in a conditional construction, it should either return 1 (to cause all controls in the window to be updated) or use window_update@ to ensure that the hypertext is updated.

To get a working example of the use of this format code, open the *browse.htm* file and the associated program *browse.cpp*.

For example:

```
WINAPP 0,0,'resource.rc'
...
c--- Only call the next line once.---
CALL add_hypertext_resource@('hyperhelp')
i=winio@('%ca[Hypertext help system]&')
i=winio@('%ww%bg[grey]%pv&')
i=winio@('%^60.22ht[Introduction]',callback)
```

Where *resource.rc* contains:

```
hyperhelp HYPERTEXT 'helpfile.htm'
```

A summary of ClearWin+ hypertext functions is given on page 325. Details can be found in Chapter 27.

Linking to the web

When whole URLs (i.e. URLs with '://' embeded in their name) are used within a `<a>HREF=...` link in a (%ht) hypertext document, the default web browser is invoked to process the link.

A routine is also available to perform this task:

```
SUBROUTINE USE_URL@(URL)
CHARACTER*(*) URL
```

This routine uses the Windows shell to open the URL. As a result it can be used to 'open' other objects such as a file with a name that has a registered extension (e.g. .DOC files). You can use this routine in a ClearWin+ window call-back function (attached to a menu item or button) in order to create a link to a web site or document.

This facility is also available via the `INTERNET_HYPERLINK` standard call-back function (see page 210).

There is also a routine to read the contents of a URL (assuming the machine has access to the internet):

```
SUBROUTINE READ_URL@(URL,FILE,MODE,ERROR)
CHARACTER*(*) URL
CHARACTER*(*) FILE
INTEGER MODE
INTEGER ERROR
```

If this routine succeeds (i.e. an internet connection is available and the URL can be accessed), `ERROR` is set to zero and the data from the URL is transferred to the specified file (which can be a full path name). If `MODE=0`, the data is assumed to be text and newlines are converted to DOS style. If `MODE=1`, binary data is assumed.

The URL can be quite general. In particular, you can use an FTP address assuming anonymous access is possible, or an HTTP address with extra information attached. For example, you could query a search engine directly using this routine.

If necessary, the system will dial up the service provider to process this call. By default the connection will not be closed after the transfer has completed. The following call will close a modem connection if one exists.

```
CALL BREAK_MODEM_CONNECTION@
```

20.

Standard call-back functions

ClearWin+ call-back functions

`winio@` call-back functions consist of functions with no arguments that return an integer result. A returned zero or negative value causes the parent window to close. The corresponding call to `winio@` will return the same value made positive. A returned value of 1 leaves the window open and updates the whole display area to reflect any changes in the data. This is not normally the most suitable return value because frequent returns can cause the screen to flicker. A returned value of 2 leaves the window open but does not update it. This value is best for repetitive display updates but you must update the parts of a window that have changed by calling `window_update@` (see page 26). The call-back function used with `%mg` may also return the value 3 (see page 257).

A number of frequently used call-back functions can be specified as character strings that denote *standard call-back functions*. (Note that there is never any confusion between a pointer to one of these character strings and a genuine function pointer, because all functions start with a distinctive sequence of machine instructions. The system decides whether the supplied pointer points to a function or to one of the special strings described below. If the supplied pointer points to an illegal address, this will cause an immediate program fault to be reported.)

Standard call-back functions usually appear within the argument list that follows the format string in a call to `winio@`. Some standard call-back functions require additional arguments that are placed in the `winio@` argument list immediately after the string that denotes the standard call-back. The following functions are available.

ABOUT

This takes one character string argument (which may include newline characters) and displays a simple “about” box. Each line of the text is centred, and an OK button is supplied for the user to close the box.

BEEP[*sound option*]

If you require a button to make a (Windows) beep sound this call-back will provide a quick and simple way of doing so. It may be most useful when used in conjunction with the call-back joiner ‘+’. It takes one of the following five possible arguments:

[EXCLAMATION]
[QUESTION]
[HAND]
[MBOK]
[ASTERISK]

These correspond to the warning icons built into Windows (see page 78).

For example:

```
i=winio@(' %^bt[PRESS] ', '+', 'BEEP[MBOK]', cb_openfile )
```

CASCADE

If this function is invoked from a window containing a MDI frame (%fr). The child windows created with %aw are cascaded leaving the active window at the top.

CONFIRM_EXIT

This call-back function takes one character string argument and displays it as a *yes/no* question. It is designed to be used with a button or menu option that will exit a window. If the user responds *yes*, the function returns zero, thus closing the window with the return number of the original button. If the user responds *no* (the default response) the window remains open.

COPY, CUT, PASTE

These three call-back functions transfer information between the edit box (%eb, %rd, %rf, %fs, or %re) that has the focus and the clipboard. Typically, they are used as the call-back function for an edit-copy/cut/paste menu item. These call-back functions require no additional winio@ arguments.

Menu items and buttons attached to these callback functions are greyed according to circumstances. For example, a control associated with the CUT function would be greyed if no text were currently marked as selected on the screen. However, if the control has a grey control variable then this variable takes precedence.

Apart from %eb, standard accelerator keys (Ctrl+C, Ctrl+X, and Ctrl+V) can also be used for COPY, CUT and PASTE.

CONTINUE

This function returns a non-zero value which therefore leaves the window open. It is used as a dummy when no action is required.

EDIT_FILE_OPEN

This standard call-back is used with %eb and takes a character string argument containing a file pattern (e.g. '*.TXT'). A second character string argument (called *fname* say) is required for the resultant file name (of size 129). A standard Windows ‘file open’ dialog is displayed and the resultant file is written to *fname*. The file is then displayed in a %eb edit box that is included in the current format window, replacing any text already there. This standard call-back must only be used in a format window containing one and only one %eb edit box. The corresponding edit box is usually supplied with a buffer length of zero when it is set up, so that the system allocates the buffer and adjusts its size as necessary. An example program appears on page 212.

EDIT_FILE

This is used with %eb and takes one character string argument that is the file name of the file to be opened. Unlike EDIT_FILE_OPEN, the standard Windows ‘file open’ dialog is not presented.

EDIT_FILE_SAVE, EDIT_FILE_SAVE_AS

These call-back functions operate on a %eb edit box in same same manner as EDIT_FILE_OPEN. If an edit box has not been filled using EDIT_FILE_OPEN, EDIT_FILE_SAVE operates like EDIT_FILE_SAVE_AS and displays the standard ‘file save’ dialog. Each call-back takes two character string arguments: one for the file pattern and the other for the resultant file name (see EDIT_FILE_OPEN above). The contents of the edit box are written to the file but the edit box is not closed. An example program appears on page 212.

EXIT

This closes the window. If a window closure call-back (%cc) has been specified, the %cc call-back is called before the closure takes place.

DEC

This takes one INTEGER variable as an argument and subtracts 1 from the variable each time DEC is called. Any controls that use this variable are automatically updated.

FATAL

The FATAL call-back terminates the program at once without performing any of the normal cleanup activities. Typically this is attached to the OK button in a window that displays a fatal error condition. Use this only if the program is likely to be corrupt and where performing a normal cleanup might hang the system.

FILE_OPENR[*caption*]

FILE_OPENR is used to display the standard Windows ‘file open’ dialog. It takes a character string argument which is a buffer to receive the file name selected by the user from the dialog. Initially this buffer must be preset to be either blank or have the form ‘*.abc’ otherwise the behaviour of FILE_OPENR is indeterminate. A call-back function should follow the buffer in the argument list. This call-back is called if the user selects a file (as opposed to pressing cancel, etc.).

For example:

```
EXTERNAL cb
INTEGER cb
CHARACTER*129 str_file
str_file=' '
i=winio@('%^bt[Open file]', 'FILE_OPENR[Open]', str_file, cb)
```

The value returned to the calling winio@ is the return value of this call-back function, or 1 if no file is selected (so that the underlying window is kept open). By default (if the file name string is initially blank) the file selection box starts by displaying all files from the current directory. However, this and several other default actions of FILE_OPENR can be modified by the prior use of the %fs and %ft formats or by setting the file name string to the form ‘*.abc’ before the call. For an example of the use of the FILE_OPENR and “+” call-back functions see the description of the %aw format on page 123. See also %fs and %ft on page 105.

FILE_OPENW[*caption*]

FILE_OPENW is analogous to FILE_OPENR but allows the user to type in a new file name, since it is assumed that the file is required for writing.

GPRINTER_OPEN

GPRINTER_OPEN is used to produce output on a graphics printer or plotter. It operates in a manner similar to PRINTER_OPEN. However, GPRINTER_OPEN takes only one argument which is a call-back function supplied in the program. This call-back function will use *drawing surface* routines to produce the graphics output (see Chapter 15).

Note that graphics printing and plotting can also be carried out using open_printer@ (see page 382).

HELP_CONTENTS

HELP_CONTENTS takes one character string argument giving the full path name of a help file (.HLP). The file is opened at its contents page. When the main window closes, the help window will also be closed if it has been left open.

HELP_ON_HELP

HELP_ON_HELP provides the standard help-on-help information. This also takes a character string argument giving the name of a help file.

HTML_PRINTER_OPEN

HTML_PRINTER_OPEN provides for the output of text with embedded HTML markups. See page 191 for further details.

INC

INC takes one INTEGER variable as an argument and adds 1 to this variable each time INC is called. Any controls that use this variable are automatically updated.

INTERNET_HYPERLINK

INTERNET_HYPERLINK takes one CHARACTER string argument which is the full URL of a file on the web. For example:

```
i=winio@('%^bt[Salford]', 'INTERNET_HYPERLINK',
           'http://www.salford.co.uk')
```

See also USE_URL@.

PRINT_ABORT

PRINT_ABORT is used to cancel a print job whilst it is being spooled.

PRINTER_OPEN

PRINTER_OPEN displays the standard Windows ‘Print’ dialog that enables the user to select a printer. The function takes two arguments. The first is an integer that is the Fortran unit number on which to open the printer. The second argument is a call-back function supplied in the program. If the printer is successfully selected, the call-back function is called. (The value returned by the call-back becomes the value returned by the enclosing winio@. Hence, a return value of 1, for example, has the effect of keeping the parent window open). If no printer is selected, PRINTER_OPEN returns 1 to keep the window open. The call-back function should perform print operations using standard Fortran output statements (for example, WRITE) on the specified unit number. It is necessary to CLOSE the unit number in order to dispatch the output to the printer.

PRINTER_OPEN1

PRINTER_OPEN1 has the same purpose and takes the same arguments as PRINTER_OPEN but does not display the standard Windows ‘Print’ dialog. Instead it uses the default printer or the printer that was selected in an earlier call to PRINTER_OPEN.

PRINTER_SELECT

PRINTER_SELECT displays the standard Windows ‘Print’ dialog and is used in conjunction with the standard call-back OPEN_PRINTER1 or the routine open_printer1@.

SELECT_ALL

SELECT_ALL has no arguments and selects all of the text in the current %eb edit box.

SET

The SET call-back takes two arguments each of type INTEGER. When the call-back is called, the first argument is set to the value of the second at the time when winio@ is called. SET returns the value 1 so that any necessary updates take place. Here is an example.

```
WINAPP
INTEGER v,w,winio@,x,ctrl
v=0
x=1
w=winio@('%`cn%^tt[Set]&','SET',v,x)
w=winio@('%2n1%4rd&',v)
w=winio@('%lw',ctrl)
x=2
END
```

The variable *v* is initially zero and takes the value 1 (rather than 2) when the button is clicked.

SOUND

If you require a sound that is not one of the standard system sounds (see BEEP above) and a sound card is installed, you can include small sound samples previously stored in standard wave file format (.WAV). The wave file must be included in a resource script or in the resource section of the program. For example:

```
MySound SOUND dognoise.wav
```

The following code illustrates how the sample is played back.

```
i=winio@('%^bt[Bark]','SOUND','MySound')
```

STOP

This call-back function closes the window and terminates the program.

SUPER_MAXIMISE

The SUPER_MAXIMISE (the spelling SUPER_MAXIMIZE also accepted) call-back is used to expand a window so that only the client area is visible. This is useful to

enable a graphic display to be shown on the entire screen. Programs that use this call-back should provide an accelerator key or other means to exit from this mode (there will be no standard or system menu visible).

TEXT

The TEXT call-back can only be invoked from a format window that uses %ht. It provides a quick way of displaying text in a window. The text itself is provided in a character string that is placed in the `winio@` argument list after this call-back.

TEXT_HISTORY

The TEXT_HISTORY call-back can only be invoked from a format window that includes %ht. A dialog box is displayed that shows the hypertext history and allows the user to select an earlier topic.

TOGGLE

The TOGGLE call-back is used to toggle an INTEGER variable between 0 and 1. The variable follows the call-back in the `winio@` argument list.

+

Sometimes it is useful to invoke more than one call-back function at once. The “+” call-back function takes two subsequent call-back functions as arguments and calls each in turn. The return value from the “+” call-back function is the return from the second call. For an example of the use of the “+” and FILE_OPENR call-back functions see the description of %aw on page 123. Note that the result of `clearwin_string@('CALLBACK_REASON')` reflects the original reason for the call-back and not “+”.

A simple text editor

The following example uses a number of the above call-back functions. It implements a simple editor.

```
WINAPP
INTEGER i,winio@
CHARACTER*129 file,new_file,help_file
help_file='myhelp.hlp'
i=winio@('%mn[&File[&Open]]&','EDIT_FILE_OPEN','*.*',file)
```

```
i=winio@('%mn[&Save]&',      'EDIT_FILE_SAVE', '*.*', new_file)
i=winio@('%mn[Save &As]&', EDIT_FILE_SAVE_AS, '*.*', new_file)
i=winio@('%mn[E&xit]&',     'EXIT')
i=winio@('%mn[&Edit[&Copy]]&', 'COPY')
i=winio@('%mn[&Cu&t]&',      'CUT')
i=winio@('%mn[&Paste]&',      'PASTE')
i=winio@('%mn[&Help[&Contents]]&', 'HELP_CONTENTS', help_file)
i=winio@('%mn[&Help on help]]&', 'HELP_ON_HELP', help_file)
i=winio@('%60.20eb', '*', 0)
END
```


21.

Format code reference

This chapter provides a detailed reference to format codes that can be used with `winio@`. New features are continuously being added to ClearWin+, details of which can be found in the enhancements files on the distribution disks.

An introduction to format windows and the `winio@` function is presented on page 22. It includes general information about *format strings* and the form of a *format code*; about *format modifiers*, option lists, *standard character strings*, and help strings. It also describes the general order in which components appear in the *format string* and in the `winio@` argument list. Please note that lowercase *n* and *m* are used to represent optional size modifiers, whilst uppercase *N* and *M* denote mandatory size modifiers (e.g. `%br`).

%ac

Purpose To define an accelerator key.

Syntax `winio@(%ac[key_description], cb_func)`
external `cb_func`

Description *key_description* is a standard character string (can be replaced by @ and a WINIO@ argument) that may include key words of the form: `ctrl`, `shift`, `alt`, `esc`, `del`, `ins`, `space`, `backspace`, `enter`, `left`, `right`, `up`, `down`, `pgup`, `pgdn`, `tab`, `home`, `end`, `keypad_centre`, `f1`, ..., `f12`. Character keys must be combined with one or more of `ctrl`, `shift` and `alt`, but the forms `ctrl+alt+character` are reserved for use by the system.

Example `winio@(%ac[Ctrl+Shift+Del], cb_func)`

See also %mn, add_accelerator@

%ap

Purpose To set the absolute position of the next control.

Syntax

```
winio@('%ap', ix, iy)
      integer ix, iy      (input)
      winio@('%`ap', rx, ry)
      double precision rx, ry (input)
```

Modifiers Grave accent (`) - use DOUBLE PRECISION real rather than INTEGER input.

Description The units of measurement corresponds to the average width and the height of the default font. Use with care as this format overrides the normal mechanism to prevent controls overlapping.

Example winio@('%ap', 2L, 6L)

See also %rp, %gd, %ob, %nl, %ff, %dy

%aw

Purpose To attach a MDI child to a parent window.

Syntax

```
winio@('%aw', ctrl)
      integer ctrl      (input)
```

Description Used in conjunction a frame (%fr). *ctrl* is a variable associated with %lw or %cv in the frame. %aw can be used with %pv.

- Notes**
- a) The system menu will contain Ctrl-F4 to close a child window (as opposed to the parent window), and Ctrl-F6 to move to the next MDI child.
 - b) As additional children are added to a frame these windows are offset with respect to the origin to make all windows visible. This offset is not reset to zero when all the children in a frame are closed.
 - c) If the standard call-back function 'CASCADE' is invoked from a window containing a MDI frame, this will cascade the child windows with the active window left at the top.

Example See page 124.

See also %fr, %lw, %cv, %pv

%bc

Purpose To specify the colour of the next %bt button.

Syntax

```
winio@('%bc[colour]')
winio@('%bc', colour_value)
integer colour_value (input)
```

Modifiers Grave accent (`) - allows *colour_value* to be changed dynamically.

Description *colour* is one of: black, white, grey, red, green, blue, and yellow. Alternatively one of the additional named colours listed for %bg can be used.

colour_value is typically a value returned by RGB@. If the grave accent is used, this value can be changed after the button has been displayed. *colour_value* is then passed to window_update@ in order to update the screen.

Example

```
winio@('%bc', RGB@(120,120,0))
winio@('%bc[white]')
```

See also %bt, %tt, %bg

%bd

Purpose To specify all four window borders individually.

Syntax `winio@('%bd', left, top, right, bottom)`
`double precision left,top,right,bottom (input)`

Modifiers Grave accent (`) - the arguments are integer pixel values.

Description The units of measurement correspond to the average width and the height of the default font. The exact size of each border can be set using this format. A negative value leaves the corresponding border unchanged.

%bf

Purpose To switch to bold font.

Syntax `winio@('%bf')`

Modifiers Grave accent (`) - use the grave to switch off.

Description %bf is cancelled by %`bf, %fn and %sf.

See also %it, %ul, %sf

%bg

Purpose Background colour format.

Syntax `winio@('%bg[colour]')`
`winio@('%bg', colour_value)`
`integer colour_value (input)`

Modifiers Grave accent (`) - set the colour of the next control.

Description *colour* is one of: black, white, grey, red, green, blue, and yellow. Alternatively one of the following additional named system colours can be used: scrollbar, background, activecaption, inactivecaption, menu, window (use this for the default window background), windowframe, menutext,

`windowtext`, `captiontext`, `activeborder`, `inactiveborder` (use this for dialog box backgrounds), `appworkspace` (use this for the MDI backdrop), `highlight` (use this when you are adding your own draw focus), `highlighttext`, `btnface`, `btntext`, `inactivecaptiontext`, and `btnhighlight`. The definition of these colours depends on the user's desktop configuration (if you require the actual value for an RGB colour, use the API function `GetSysColor` with the named colour parameters defined the file `windows.ins`; e.g. `COLOR_BTNFACE`).

colour_value is typically a value returned by `RGB@`.

Use without the grave modifier to set the background colour of the main window. Use with the grave modifier to set the background colour of the next control (e.g. `%rd`, `%rf`, `%rs`, `%ls`, but not `%bt`).

Example `winio@('%bg', RGB@(120,120,0))`
`winio@('%bg[white]')`
`winio@('%bg[btnface]')` the default button face colour.

See also `%bc`, `%wp`

%bh

Purpose To provide a bubble help system.

Syntax `winio@('%bh',ctrl)`
`integer ctrl` (input)

Modifiers Grave accent (`) - introduces a delay before the message appears.

Description Set *ctrl* to 1 in order to enable help messages and to 0 in order to disable messages.

See also `%he`, `%th`

%bi

Purpose To supply an icon for the next button.

Syntax `winio@('%bi[icon_name]')`

Description *icon_name* is a standard character string (can be replaced by @) that provides a name appearing in a resource script.

Notes *icon_name* can be constructed from a known Windows handle (see page 79).

Example `winio@('%bi[myicon]')`
`myicon ICON file.ico` (resource file)

See also %ic

%bk

Purpose To allow a right mouse click to be used on the next %bt button.

Syntax `winio@('%bk')`

Description %bk can be used before %^bt with the result that the associated call-back function is also called with a right mouse click. To differentiate between right and left mouse button clicks use `clearwin_string@('CALLBACK_REASON')`.

%bm

Purpose To draw a bitmap.

Syntax `winio@('%bm[bitmap_name]')`
`winio@('%`bm', handle)`
`integer handle (input)`

Modifiers Grave accent (`) - use a Windows API bitmap handle.

Caret (^) - the call-back is called when the user clicks on the bitmap.

Question mark (?) - a help string is supplied.

Description *bitmap_name* is standard character string (can be replaced by the @ symbol) giving the name appearing in a resource script.

handle is a value returned by a modified form of the API function **LoadBitmap**.

Example `winio@('%bm[mybitmap]')`
`mybitmap BITMAP file.bmp` (resource file)

See also %ic,%gi

%br

Purpose To draw a horizontal or vertical bar which is partially filled with a user-selected colour.

Syntax `winio@('%Nbr[options]', fill, colour_value)`
`double precision fill` (input)
`integer colour_value` (input)

Modifiers Question mark (?) - a help string is supplied.

Description `fill` specifies the amount of fill in the range 0 to 1 inclusive.

`colour_value` is the RGB value of the fill colour.

`N` is a integer constant giving the length of the bar in average characters.

`options` may include `no_border` and `percentage`, and one of `left_right`, `right_left`, `top_bottom`, `bottom_top`. `percentage` adds a progress indicator (in the form of a percentage) to a horizontal bar only.

Example `winio@('%15br[bottom_top]', fill, RGB@(200,200,0))`

See also %sl

%bt

Purpose To insert a standard button.

Syntax `winio@('%nbt[button_text]')`
`winio@('%~nbt[button_text]', grey_ctrl)`
`integer grey_ctrl` (input)

Modifiers Tilde (~) - used to add a grey control.

Grave accent (`) - used to make this the default button (one for each dialog).

Caret (^) - the call-back function is called when the user clicks on the button.

Question mark (?) - a help string is supplied.

Description *n* is an optional positive integer which is used to specify the width of the button.

grey_ctrl is set to 1 to enable the button and 0 to disable (grey) it.

button_text is a standard character string (can be replaced by @) and can include an ampersand (&) to mark an accelerator key. To change the text dynamically use %lc and the API function **SetWindowText**.

The background colour can be specified using %bc and the text font and size can be specified using %fn and %ts. Other effects can be obtained by using %lc.

Example `winio@('%`?~^10bt[OK][Click when done]', grey, cb_func)`

See also %tt, %bc, %bi, %ts, %fn, %lc

%bv

Purpose To insert a hierarchical tree.

Syntax `winio@('%bv[options]',width,height,items,num_items,sel)`
`winio@('%`bv[options]',width,height,items,num_items,sel,`
`bmp_str)`
`character*(*) items(num_items) (input/output)`
`integer num_items (input parameter)`
`integer sel (input/output)`
`integer width,height (input)`
`character*(*) bmp_str (input)`

Modifiers Grave accent (`) - used to provide a list of bitmaps.

Caret (^) - call-back function is provided.

Question mark (?) - a help string is supplied.

Description %bv (branch view) provides a tree view that is similar in some respects to %tv but the %bv control has a different appearance and %bv has more options. It also shares common features with %lv.

width and *height* provide the dimensions of the control in pixels.

num_items is an integer constant giving the number of nodes in the tree. Nodes with blank data are not displayed in the control. This means that the number of nodes that are displayed can be changed under program control.

items is an array of *num_items* character strings, each string describing a node in turn

(see the example below).

sel is an integer that provides the index of the node that is currently selected.

If a grave accent is provided, *bmp_str* is a character string giving a list of 16x16 bitmap resources in the form 'bmp1,bmp2'. If a bitmap has a one-pixel border of a certain colour, then this colour is used as a mask for the bitmap. Pixels with the mask colour are replaced by the background colour. The default mask is white.

%bv can take a pivot (%pv). Also the right mouse button can be used to trigger a popup menu (%pm) defined in the parent window. The background colour for %bv is white and at the moment this cannot be changed.

Example

```
CHARACTER*80 items(4)
INTEGER sel
items(1)='AEBBook'
items(2)='BCAChapter 1'
items(3)='CCASEction 1.1'
items(4)='CCASEction 1.2'
sel=1
i=wini@(%`bv',200,200,items,4,sel,'bmp1,bmp2')
```

As for %tv, the first character in each string provides the index for the level of the node. The second character marks the node as either expanded (E) or collapsed (C). The third character is only required if a list of bitmaps is provided and gives the index to select a bitmap from the list.

Options

edit_labels	Allows the text for a node to be edited.
show_selection_always	Always show the selection, if any, even if the control does not have the focus.
no_border	Displays the control without a border.
has_lines	The tree is displayed with nodes joined by lines.
lines_at_root	The root node also has a line to its left.
has_buttons	Use this with has_bitmaps if you need buttons at the first level.
paired_bitmaps	+/- expansion buttons are displayed. 'Expanded'/'collapsed' bitmaps are paired in the list.

Label editing

edit_labels enables label editing and requires a call-back function that can test for the call-back reasons 'BEGIN_EDIT', 'END_EDIT' and 'EDIT_KEY_DOWN'.

With BEGIN_EDIT you can get the index of the selected node by the call

`clearwin_info@('ROW_NUMBER')`. If the call-back function returns the value 4 then editing of this node is prevented.

With `EDIT_KEY_DOWN` you can validate each character as it is typed. Use `clearwin_info@('KEYBOARD_KEY')` to get the key and/or `clearwin_string@('EDITED_TEXT')` to get the resulting string with the character inserted. If the call-back function returns the value 4 then the character is not accepted. If you want to translate the input key to another key then the call-back function must return the ASCII character code (31<code<256) of the translated key.

With `END_EDIT` you can get the result of the edit by the call `clearwin_string@('EDITED_TEXT')`. If the call-back function returns the value 4 then the new label is rejected the original label is restored. Otherwise, unlike `%lv`, the edit is accepted and your data is automatically updated. You cannot delete a node by accepting a blank entry.

Expanding a node

When a node is expanded or collapsed, the call-back function is called with the reason '`ITEM_EXPANDED`'. You can respond to this (as for `%tv`) by changing the bitmap index and calling `window_update@(items)`. However, it is simpler and much faster if you use the option `paired_bitmaps`. If you provide this option, each 'expanded' bitmap must follow immediately after the corresponding 'collapsed' bitmap in the list. Also the bitmap change is automatically done for you and your data is automatically modified to reflect the change. You can save and restore the data if you want to reload the control with the same expanded state.

Call-back reasons

<code>ITEM_EXPANDED</code>	A node has been expanded or collapsed (see above).
<code>BEGIN_EDIT</code>	A label is about to be edited (see above).
<code>EDIT_KEY_DOWN</code>	The user has pressed a keyboard key during a label edit. <code>clearwin_info@('KEYBOARD_KEY')</code> provides the key code (see above).
<code>END_EDIT</code>	A label has been edited (see above).
<code>SET_SELECTION</code>	A node has been selected. <code>clearwin_info@('ROW_NUMBER')</code> provides the index.
<code>MOUSE_DOUBLE_CLICK</code>	The user has double-clicked using the left mouse button. <code>clearwin_info@('ROW_NUMBER')</code> provides the index.
<code>KEY_DOWN</code>	The user has pressed a keyboard key. <code>clearwin_info@('KEYBOARD_KEY')</code> provides the key code.

Example See page 71.

See also %tv, %ls, %ms, %pb, %ps

%bx

Purpose To add a raised grey bar to a tool bar.

Syntax `winio@('%bx', depth)`
double precision depth (input)

Description %bx is used, particularly after %tt or %tb, in order to provide a grey toolbar at the top of the window. *depth* is measured as a fraction of the character height and this amount is added to the height, one half above and one half below the buttons.

%bx can only be used once in any window. Its use is not restricted to %tt and %tb. %bx creates a grey background from the point where it is used to the top of the window. It automatically removes the border (as in %ww[no_border]) and the grey background spans the width of the window.

See also %tb, %tt

%ca

Purpose To provide a caption for the window or property sheet.

Syntax `winio@('%ca[caption]')`

Modifiers Grave accent (`) - when used, the minimum width of the window is not limited by the width of the caption and part of the caption may be hidden.

Description *caption* is a standard character string (can be replaced by @).

In order to change the caption dynamically use the @ symbol and follow a change in the string variable by a call to `window_update@`. Alternatively use the API function `SetWindowText` together with %hw.

When used with %sh in a property sheet, the caption can include an ampersand (&) to mark an accelerator key.

Example `winio@('%ca[Edit Window]')`

See also %sh

%cb

Purpose To close a box that has been opened using %ob.

Syntax `winio@('%cb')`

See also %ob

%cc

Purpose To control the closure of a window.

Syntax `winio@('%cc', cb_func)`
`external cb_func`

Modifiers Grave accent (`) - the call-back is called after the window has closed rather than before..

Description Provides a link to a call-back function by which the user controls the action to be taken just before a window is closed. If a grave is used (%`cc) then the associated call-back function is called immediately after the window is closed. You can use both %cc and %`cc in one window.

See also %lw, %ew

%ch

Purpose Child window format - inserts a child window.

Syntax `winio@('%ch', ctrl)`
`integer ctrl (input)`

Description `ctrl` is a handle returned by %`lw .

See also %lw

%Cl

Purpose To display a colour palette.

Syntax `winio@('%cl[options]', colour_values)`
integer `colour_values(3)` (output)

Description Up to three values can be returned from the resulting dialog box, the first for the left mouse button, the next for the right button and the third for the middle button.

options may be one or more of:

b1	Right mouse button only
b2	Left and right buttons (the default)
b3	Left , right and middle buttons
title1	Title above button boxes
title2	Title above palette
iconic	<i>Transparent and inverse transparent selection</i>

Example `winio@('%cl[b3,title2=Palette]', colour_values)`

%cn

Purpose To centre text and controls in the window.

Syntax `winio@('%cn')`

Modifiers Grave accent (`) - extends the scope to the remainder of the window.

Description %cn is principally used on text and buttons. It forces everything which follows it, up to the next new line or form-feed character, to be centred in the window.

When used within a box (%ob) the centring applies within that box.

See also %rj

%CO

Purpose To modify subsequent %rd, %rf , %rs, %re, %el boxes.

Syntax `winio@('%co[options]')`

Description *options* is a list containing one or two of:

<code>check_on_focus_loss</code>	Perform checks, calls the call-back, and updates the corresponding data item only when focus is transferred to another control. This is particularly valuable if numeric limits have been imposed by %il/%fl.
<code>data_border</code>	Put a border round data boxes (default).
<code>full_check</code>	Perform checks and call-back every change (default).
<code>right_justify</code>	Align the text to the right border. Not for %el.
<code>left_justify</code>	Align the text to the left border (default).
<code>no_data_border</code>	Omit border round data boxes. Not for %el.

Example `winio@('%co[check_on_focus_loss]')`

See also %rd, %rf, %rs, %re, %el

%CU

Purpose Establishes a cursor for the next control in a format.

Syntax `winio@('%cu[cursor_name]')`
`winio@('%`cu', const)`
`integer const (input)`
`winio@('%Ncu[name1][name2] . . . [nameN]', sel)`
`winio@('%`Ncu', const1, const2, . . . , constN, sel)`
`integer sel, const1, const2, . . . , constN (input)`

Modifiers Grave accent (`) - cursor is specified by using a ClearWin+ parameter from *windows.ins*.

Description *cursor_name* is a standard character string (can be replaced by @) that is used to

identify a cursor via a resource script.

The alternative uses a grave accent modifier together with *const* which is one of:

CURSOR_ARROW	Standard arrow cursor
CURSOR_IBEAM	Text I-beam cursor
CURSOR_WAIT	Hourglass cursor
CURSOR_CROSS	Cross hair cursor
CURSOR_UPARROW	Vertical arrow cursor
CURSOR_SIZE	A square with a smaller square inside its lower-right corner
CURSOR_ICON	Empty icon
CURSOR_SIZENWSE	Double-pointed cursor with arrows pointing Northwest and Southeast
CURSOR_SIZENESW	Double-pointed cursor with arrows pointing Northeast and Southwest
CURSOR_SIZEWE	Double-pointed cursor with arrows pointing west and east
CURSOR_SIZENS	Double-pointed cursor with arrows pointing North and South

If *const* is not one of these values then it is assumed to be a valid Windows API handle of an existing cursor (cf. %bm and %ic).

Another pair of alternatives uses a positive integer *N* to provide a selection of *N* different cursors. In this case the variable *sel* is given a value in the range 1 .. *N* in order to specify a particular cursor. The examples below illustrate the first pair of alternatives.

Example `winio@('%cu[mycursor]')`
`mycursor CURSOR file.cur (resource file)`
`winio@('%`cu',CURSOR_IBEAM)`

See also `%dc`, `set_cursor_waiting@`

%CV

Purpose To set a control variable for a window.

Syntax `winio@('%cv', ctrl)`
`integer ctrl` (output)

Description %cv has the same effect as %lw but does not leave the window open. The returned value of *ctrl* can be used with %aw.

See also %aw

%CW

Purpose To insert a *ClearWin* window.

Syntax `winio@('%N.Mcw[options]', fortran_unit)`
`winio@('%`N.Mcw[options]', fortran_unit, handle)`
`integer fortran_unit` (input)
`integer handle` (output)

Modifiers Grave accent (`) - used to return a handle for the window.

Question mark (?) - a help string is supplied.

Description *N* is the width and *M* the depth of the window in average characters.

%cw can take a pivot (use %pv before %cw) in order to size the window at run time.

fortran_unit is the unit number of the input/output stream. Use zero for the default stream.

handle can be used with routines such as `set_max_lines@`.

If a caption, etc. is required (for example to make the window movable) the %cw format should be embedded in a child window which is itself embedded in the main window.

The standard call-back functions CUT, COPY and PASTE can be used in this context.

%cw is normally used for output and in particular for debugging.

options can be one or both of vscroll and hscroll in order to provide scroll bars.

See also %uw

%dc

Purpose Establishes a default cursor for the window.

Syntax

```
winio@('%dc[cursor_name]')
winio@('%`dc', const)
integer const      (input)
winio@('%Ndc[name1][name2] . . . [nameN]', sel)
winio@('%`Ndc', const1, const2, . . . , constN, sel)
integer sel, const1, const2, . . . , constN      (input)
```

Modifiers Grave accent (`) - cursor is specified by using a ClearWin+ parameter from *windows.ins*.

Description *cursor_name* is a standard character string (can be replaced by @) used to identify a cursor via a resource script.

The alternative uses a grave accent modifier together with *const* which is one of:

CURSOR_ARROW	Standard arrow cursor
CURSOR_IBEAM	Text I-beam cursor
CURSOR_WAIT	Hourglass cursor
CURSOR_CROSS	Cross hair cursor
CURSOR_UPARROW	Vertical arrow cursor
CURSOR_SIZE	A square with a smaller square inside its lower-right corner
CURSOR_ICON	Empty icon
CURSOR_SIZENWSE	Double-pointed cursor with arrows pointing Northwest and Southeast
CURSOR_SIZENESW	Double-pointed cursor with arrows pointing Northeast and Southwest
CURSOR_SIZEWE	Double-pointed cursor with arrows pointing west and east
CURSOR_SIZENS	Double-pointed cursor with arrows pointing North and South

If *const* is not one of these values then it is assumed to be a valid Windows API handle of an existing cursor (cf. %bm and %ic).

Another pair of alternatives uses a positive integer *N* to provide a selection of *N* different cursors. In this case the variable *sel* is given a value in the range 1 .. *N* in order to specify a particular cursor. It is not necessary to call *window_update@* when

sel is changed. The examples below illustrate the first pair of alternatives.

Example `winio@('%dc[mycursor]')`
`mycursor CURSOR file.cur (resource file)`
`winio@('%`dc',CURSOR_IBeam)`

See also `%cu`

%dd

Purpose To insert a spin wheel for an integer edit box (%rd) or a string edit box (%rs).

Syntax `winio@('%dd', step)`
`integer step (input)`

Description %dd can be used with %rd in order to provide an attached spin wheel.

step is the size of the increase/decrease. The resulting %rd value will be a multiple of *step*. Any call-back function should be attached to %rd. %dd is placed before the %rd to which it refers.

%dd can also be used with %rs. In this case it should have a non-zero *step* value (which is ignored). The subsequent %rs box should have a call-back function which uses `clearwin_string@('CALLBACK_REASON')` to identify the reasons '`SPIN_UP`' and '`SPIN_DOWN`'. The call-back function must provide the response to the spin (for example, by cycling through the days of the week).

Example `winio@('%dd', 1L)`

See also `%rd, %il, %rs`

%de

Purpose To disable some other window while this window is active.

Syntax `winio@('%de', hwnd)`
`integer hwnd (input)`

Description %de is used to force some other window to be disabled while the current window is active. This is the default behaviour for other ClearWin+ windows unless %ww is used. This format is particularly useful in situations in which there are non ClearWin+ windows present.

%df

Purpose To insert a spin wheel for a floating point value.

Syntax `winio@('%df', step)`
double precision *step* (input)

Description %df can be used with %rf in order to provide an attached spin wheel.

step is the size of the increase/decrease. The resulting %rf value will be a multiple of *step*. Any call-back function should be attached to %rf.

Example `winio@('%df', 1.0D0)`

See also %rf, %fl

%dl

Purpose To set a call-back function to be called at regular intervals via a timer.

Syntax `winio@('%dl', interval, cb_func)`
double precision *interval* (input)
external *cb_func*

Description *interval* is the minimum time in seconds between calls. A call is made at idle time (e.g. awaiting input) and when `yield_program_control@` is called. A given window cannot have more than one timer.

Example `winio@('%dl', 2.0D0, cb_func)`

%dr

Purpose To provide a call-back that is called when a file or a group of files is dragged and dropped onto the window.

Syntax `winio@('%dr', cb_func)`
`external cb_func`

Description Use a call to `clearwin_string@('DROPPED_FILE')` within the call-back in order to get the name of the file that has been dropped. If there is more than one file then the call-back function is called for each in turn. You can use `clearwin_info@('DROPPED_COUNT')` to get the number of files that have been dropped (call this *n* say) and then use `clearwin_info@('DROPPED_CURRENT')` to get the index of the current file in the range from 1 to *n*.

Example See page 125.

See also %fs, %ft, get_filtered_file@

%dw

Purpose To insert an owner draw graphics box.

Syntax `winio@('%dw[options]', context)`
`integer context` (input)

Modifiers Caret (^)

Question mark (?) - a help string is supplied.

Description *context* is value returned by a call to `get_bitmap_dc@`.

%dw will take a pivot (%pv).

See page 157 for further details.

See also %gr

%dy

Purpose To provide a vertical displacement by a non-integral number of character depth units.

Syntax `winio@('%dy' , d)`
double precision *d* (input)

Description %dy is used to force a vertical displacement by a possibly fractional number of character depths. The displacement is calculated in terms of the initial font size, regardless of any %ts formats that may have been used. The displacement can be negative, but unlike %rp, the normal mechanism will operate to prevent two controls overlapping, so %dy cannot be used to place a control over something else.

Notes A value for *d* of 0.5D0 is equivalent to a half line feed.

%dy has no effect when there is no space for the next item to move into.

See also %ap, %rp

%eb

Purpose To insert an edit box.

Syntax `winio@('%N.Meb[options]' , buffer, buff_size)`
`winio@('%`N.Meb[options]' , buffer, buff_size, edit_info)`
character*(*) *buffer* (input/output)
integer *buff_size* (input)
integer *edit_info*(24) (input/output)

Modifiers Caret (^) - the call-back function is called when a key is pressed or the mouse is moved.

Question mark (?) - a help string is supplied.

Grave accent (`) - the program supplies edit information. The array name *edit_info* is placed in the argument list after any grey control variable and before any call-back function.

Tilde (~) - adds a variable that controls the grey (enable/disable) state. The variable should precede the call-back function (if any) in the argument list.

Description An edit box created using %eb provides a wide range of advanced text editing features. See Chapter 12 for details.

N represents the width of the box in average characters. *M* represents the depth which, if omitted, defaults to 1.

buffer is a character string for the text which must be terminated with chr(0).

buff_size is the size of the buffer. A zero value may be supplied to allow the edit box to allocate its own memory re-sizing as required (*buffer* is ignored).

%eb will take a pivot (%pv).

If the caret modifier (^) is used and the call-back function does not handle the response to control keys such as Delete, BackSpace, etc., then it should return a zero value.

See also %cc

%el

Purpose To insert an editable combo box.

Syntax

```
winio@('%n.mel[option]', items, num_items, cur_item)
winio@('%~n.mel[option]', items, num_items, cur_item, grey_ctrl)
character*n items(num_items)           (input)
integer num_items                      (input parameter)
character*n cur_item                  (input/output)
integer grey_ctrl                     (input)
```

Modifiers Caret (^) - call-back function is called when the user selects an item.

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

%co[check_on_focus_loss] can be used with %el.

Description This control is similar to %`ls but *cur_item* is a string for the current text rather than the index of the selected item.

If a call-back is supplied, *option* can be set to the keyword extended. The result is that the call-back is called when the list drops down or closes up. These events must be detected from the call-back by calling clearwin_string@ with the argument 'CALLBACK_REASON' which returns 'DROPDOWN' or 'CLOSEUP'.

Using clearwin_info@ with the argument 'EDITABLE_LISTBOX_INDEX' provides the index of the listbox item that has been selected by the user. If this value is obtained within the %el callback function then you may need to add the option extended in

order or ensure that the callback function is called every time an item is selected.

See %ls for further details.

See also %ls, %ms, %lv, %pb, %ps

%eq

Purpose To insert a mathematical equation in a window.

Syntax `winio@('%eq[equation]', width, height)`
integer *width*, *height* (input)

Modifiers Grave accent (`) - avoids a fatal error condition when the equation is not correctly constructed.

Description *width* and *height* provide the dimensions in pixels of a box enclosing the equation. If the equation is fixed, zero values can be supplied and the size will be automatically adjusted to fit the contents.

equation is a standard character string (can be replaced by @) that uses the standard character set except for the three characters {} that are used to define special symbols. See page 100 for further details.

See also `winio@('%eq[E=MC{sup 2}]', 0L, 0L)`

%es

Purpose To cause the window to close when the Escape key is pressed.

Syntax `winio@('%es')`

Description This is equivalent to `winio@('%ac[Esc]', 'EXIT')`.

See also %ac

%ew

Purpose To specify a call-back function to be called if the user shuts down the Windows session (exit Windows) whilst the program is running.

Syntax `winio@(%ew, cb_func)`
`external cb_func`

Description The call-back function could be used to ask the user if results are to be saved before the session ends, etc.. If the call-back function returns zero, the shutdown is allowed to proceed. Any other value will abort the shutdown and leave the application running.

Note that if your program allows shutdown to proceed, other applications that are still running may still elect to abort the shutdown.

See also %cc

%fb

Purpose To select a font for use with all subsequent buttons.

Syntax `winio@(%fb)`

Description If %fn is used to select a font (optionally with %ts, %tc, %bf, %it and %ul) then following this by %fb (with no arguments) will cause the font to be remembered for use with all subsequent buttons unless and until %fb is used again.

See also %fn

%fd

Purpose To set the font to the default Windows 95 font (Windows 95/98 only).

Syntax `winio@(%fd)`

See also %fn, %sf

%off

Purpose Form feed. To move down to below any existing controls.

Syntax `winio@('%ff')`

See also `%nl`, `%ta`, `%ob`

%fh

Purpose To use a Windows API font.

Syntax `winio@('%fh' , handle)`
`integer handle (input)`

Description `handle` is the font handle of a font created by, for example, the Windows API function `CreateFont`. `%fh` makes this the current font for the window.

See also `%fn`

%fl

Purpose To specify the lower and upper limits for subsequent %rf formats.

Syntax `winio@('%f1' , lower , upper)`
`winio@('%`f1')`
`double precision lower , upper (input)`

Modifiers Grave accent (`) - cancels an earlier %fl and restores the limits to the minimum and maximum for the data type.

Description `lower` is the lower limit and `upper` is the upper limit.

Example `winio@('%f1' , 0.0D0 , 10.0D0)`

See also `%rf`, `%df`, `%co[check_on_focus_loss]`

%fn

Purpose To select a font for subsequent text.

Syntax `winio@('%fn[font_name]')`

Description *font_name* is a standard character string (can be replaced by @) that provides the name of a font that is in the user's system.

%fn cancels %bf but not %it and %ul.

Example `winio@('%fn[Times New Roman]')`

See also %sf, %ts, %bf, %it, %ul

%fr

Purpose To define a MDI frame to contain child windows attached by %aw.

Syntax `winio@('%fr', width, height)`
`winio@('%`fr', width, height, ctrl)`
`integer width, height (input)`
`integer ctrl (output)`

Modifiers Caret (^) - call-back function is called when the topmost child window changes.

Grave accent (`) - used to provide access to the handle of the current child window.

Description *width* and *height* give the width and height of the frame in pixels.

If %fr is preceded by %pv then the frame can be re-sized at run time.

ctrl is returned as the handle of the currently selected child window (see %hw) or zero if none is selected.

A given window cannot contain more than one frame.

Notes The call-back function is called with the 'reason' MDI_PAGE_TURN (see clearwin_string@) whenever the topmost child window changes. Typically this would be used in conjunction with the grave accent to get the handle of the topmost window in a variable.

Example See page 124.

See also %aw, %hw, %pv

%fs

Purpose To specify the working directory and file filter for subsequent file open/save call-backs.

Syntax `winio@('%fs[path_name]')`

Description *path_name* is a standard character string (can be replaced by @) giving the current working directory which optionally may include a wild-card. %fs cannot be used to change the path dynamically at run time (use `get_filtered_file@` instead).

Example `winio@('%fs[c:\project*.txt]')`

See also %ft, `get_filtered_file@`, `set_open_dialog_path@`

%ft

Purpose To specify filter information for subsequent file open call-backs.

Syntax `winio@('%ft[filter_type][wild_card]')`

Modifiers Grave accent (`) - used to start again rather than add to earlier %ft information.

Description *filter_type* is the description of the type of file, *wild_card* represents the file extension of the files to be listed in the standard file-open and file-save dialog boxes. *filter_type* and *wild_card* are standard character strings (can be replaced by @) but %ft cannot be used to change this information dynamically at run time (use `get_filtered_file@` instead).

%ft can be used more than once to supply a number of different filters.

Example `winio@('%ft[Text files][*.txt]')`

See also %fs, `get_filtered_file@`, `set_open_dialog_path@`

%ga

Purpose To enable radio buttons and/or bitmap buttons to be ganged together so that if one is switched *on*, all the others are switched *off*.

Syntax `winio@('%Nga',state1, state2, . . . , stateN)`
`integer state1, state2, . . . , stateN (output)`

Modifiers Grave accent (^) - used to specify that one control is always on.

Description *N* is a positive integer greater than 1. *state1 . . . stateN* are the on/off state controls for radio buttons and/or bitmap buttons defined elsewhere. A given state variable can be used in more than one control (%rb and/or %tb). However, it is unusual for a variable to appear in more than one %ga format for a given window. If it does then there must only be one variable that is common and the effect is to merge the two gang sets. If the grave accent is used on one of the gangs then it must be used on both.

Example `winio@('%3ga',rb1, rb2, rb3)`

See also %rb, %tb

%gd

Purpose To supply a temporary grid to help with the positioning of controls.

Syntax `winio@('%gd')`

Description The grid is placed at positions corresponding to the units in %ap.

See also %ap

%gf

Purpose To get the handle of a font.

Syntax `winio@('%gf',font_hdle)`
`integer font_hdle (output)`

Description *font_hdle* is returned as the handle of the current font. This handle can be used in calls to Windows API functions that need a font handle.

%gi

Purpose To draw a (possibly animated/transparent) GIF image.

Syntax `winio@(%gi[gif_name])`

Modifiers Caret (^) - the call-back is called when the user clicks on the image.

Question mark (?) - a help string is supplied.

Description *gif_name* is a name appearing in a resource script if type GIF. The resource compiler stores such resources in a non-GIF format (for legal reasons). The GIF file can be animated and/or partially transparent.

A call-back can be supplied to receive single and double clicks. The call-back function can interrogate the `clearwin_info@` parameters `GIF_MOUSE_X` and `GIF_MOUSE_Y` to determine where within the image the click occurred. This means that a medium to large image can contain regions which respond in different ways to mouse clicks. Co-ordinates are in pixels relative to the top left of the image. Note that if the GIF image is displayed at another screen resolution, these pixel co-ordinates will consistently refer to the same point in the image.

Example `winio@(%gi[my_gif])`
my_gif GIF file.gif (resource file)

Notes A GIF file can contain more than one image frame, and each image can specify the duration for which it is shown. Animations can be one-shot, repeated for a finite number of times, or (most usefully) cycled indefinitely. Many image processing and paint tools will create simple (non-animated) GIF files, and these can be assembled into animated GIFs using a variety of tools, such as the Microsoft GIF Animator, available from the Microsoft Web site. GIF files are commonly available on the internet. MATHEMATICA can export graphics as GIF files.

A GIF image contains size information, so none is required with the format.

Hypertext (%ht) containing images normally refers to BITMAP resources. However, if a particular named resource is not found as a bitmap, ClearWin+ will look for the corresponding GIF resource. This means that GIFs (possibly animated/transparent) can also be used in hypertext.

Animations require memory and processing time, so it might be unwise to use a large and complex animation on a ‘please wait’ screen that is displayed while a lengthy operation is being performed. Obviously, slower animations are less expensive than rapid ones. Also, for the animation to be maintained, it is important that very lengthy calculations are broken up with calls to `temporary_yield@`.

One way to reduce the performance cost of many animations is to exploit the fact that

much of an image may remain the same from frame to frame. Consider for example an animated ballot paper. It might be that only the region that contains the cross would actually need to change. It is possible to assemble a GIF consisting of one full image and a series of patches, which will obviously require less memory and processing resources than the corresponding set of full images. If you decide to proceed this way your program will remain unchanged, it will simply require more work with the image processing tools.

On 16 or 256 colour systems, GIF images are displayed without the use of a palette. This is to avoid some of the problems with Windows when two or more processes compete for the use of the palette. GIF images are best viewed in a high colour mode. If in doubt it is worth switching your screen to 256 colour mode to ensure that your graphic looks acceptable. You can read the `clearwin_info@` parameter, `SCREEN_NUM_COLOURS` do determine what to do at run time.

GIF images may be used to provide fancy buttons that attract attention because of their animation, to provide front cover screens for programs, and to add colour and interest. Larger images can even be used as image maps by reading the click co-ordinates, as explained above.

The GIF resources are decompressed by the resource compiler and recompressed by a different method. This means that your executables and the DLL `salfibc.dll` do not contain any code related to the processing of the GIF format. For this reason we have not supplied any routines to enable you to export graphics to GIF files. However, if you want to incorporate your graphics in animated GIFs this is still possible. Simply create a series of still images as BMP files, convert them to GIFs using a shareware image processing program, and assemble the result as an animated GIF.

See also `%ic,%bm`

%gp

Purpose To get the current co-ordinates of the window position.

Syntax `winio@('%gp', x, y)`
`integer x, y (output)`

Modifiers Grave accent (`) - used only in conjunction with `%aw` in order to provide co-ordinates that are relative to an enclosing MDI frame.

Description *x* and *y* are returned as the co-ordinates in pixels of the current position in the window relative to the top left-hand corner (0,0) of the window. These values can be used with `%sp`.

See also %sp

%gr

Purpose To provide a rectangular area for use with *drawing surface* routines.

Syntax `winio@('%gr[options]', width, height)`
`winio@('%`gr[options]', width, height, handle)`
`integer width, height, handle (input)`

Modifiers Grave accent (`) - used to input a handle for the graphics area.

Caret (^) - used to supply a call-back function.

Question mark (?) - a help string is supplied.

Description *width* and *height* provide the pixel dimensions of the area. *handle* is a programmer-supplied handle to distinguish between different graphics areas.

%gr can be preceded with %pv in order to re-size the area at run time. For details and a list of the options see Chapter 15.

See also %dw

%he

Purpose To specify the location of help information.

Syntax `winio@('%n.mhe')`

Description If %he, %bh and %th are not used, help information (obtained using the question mark modifier with certain format codes) is placed at the bottom of the window. %he specifies the current position as the required position for help information.

n and *m* are optional (*m* cannot be specified without *n*). *m* defaults to 1. They specify the width and depth of the area to take the help string measured in average character units.

See also %bh, %th

%ht

Purpose To attach a hypertext document.

Syntax `winio@('%N.Mht[start_topic]')`

Modifiers Caret (^) - called when a jump is made to a topic that does not exist.

Grave accent (`) - used to enable the given call-back function to be called every time a topic is changed.

Description *N* and *M* are the width and depth of the viewing area measured in average character units. %ht may be preceded by %pv in order to allowing re-sizing at run time.

start_topic is a standard character string giving the name of the first topic to be viewed.

The hypertext document is attached by a call to `add_hypertext_resource@`.

For further information see Chapter 19.

Example

```
call add_hypertext_resource@('help')
      winio@('%pv%^60.20ht[Contents]',cb_func)
      help HYPERTEXT file.htm           (resource file)
```

See also %pv

%hw

Purpose To return the handle of the current window.

Syntax `winio@('%hw', handle)`
`integer handle` (output)

Description The returned value can be used in calls to Windows API functions and is the same as that given by `clearwin_info@('LATEST_FORMATTED_WINDOW')`. You can test if the current window has the focus by calling `clearwin_info@('FOCUS_WINDOW')`.

See also %lc

%hx

Purpose To attach a horizontal scroll bar to the next control.

Syntax

```
winio@('%hx', page_step, max_val, cur_val)
integer page_step, max_val      (input)
integer cur_val                (input/output)
```

Modifiers Caret (^) - the call-back is called when the user clicks on the scroll bar.

Description *cur_val* is the value that corresponds to the position of the scroll box (also called the *thumb*). It returns a value between zero and *max_val* - 1.

page_step is the amount that *cur_val* is to be increased/decreased when the user clicks on the scroll bar (other than on the thumb). For a document, *page_step* will be set to the number of lines that are to be displayed in the window and *max_val* should be set to *n* - *page_step*, where *n* is the total number of lines in the document.

With %hx and %vx the three arguments *cur_val*, *page_step* and *max_val* can be passed as variables and changed dynamically with a call to window_update@.

A call to clearwin_info@('INTERMEDIATE_SCROLL') will determine if the callback was called whilst the thumb was moving. A return value of 1 indicates movement whilst zero indicates that the thumb is stationary. Both of these events occur at the end of a move.

%hx and %vx should not be used with controls like %eb and %cw which have their own scroll bar mechanism.

Example

```
winio@('%^hx', 5L, 20L, pos, cb_func)
```

See also %vx

%ib

Purpose To insert a image button or toolbar.

Syntax

```
winio@('%n.mib[options]', bitmap_name, stat_ctrl, cb_func, ...)
character*(*) bitmap_name
integer stat_ctrl
external cb_func
```

Modifiers Question mark (?) - A help string for each button is placed in order (in square

brackets) after the options (use empty brackets for the options list if it is blank).

Description %ib provides an enhanced and simplified replacement for %tb. Like %tb, %ib can be used to create toolbars or rectangular arrays of image buttons. However, only one flat image is required for each button - all the variants are created automatically. Also, text can be automatically added to the image.

%n.mib creates block of buttons *n* buttons wide and *m* deep. *n* and *m* are optional and both default to 1. When *n* is used .*m* is optional. The arguments consist of the following triplet repeated for each button in turn: *bitmap_name*, *stat_ctrl*, *cb_func*. Note that every button has a call-back function.

bitmap_name is the name of a bitmap resource. If text is to be added to the image, the name is followed by a forward slash and then the text. For example, here is the code for a single button with some text:

```
i=winfo@('%ib','my_button/Press me',ctrl,cb)
```

The button text can include newline characters:

```
'my_button/Press'//CHAR(10)//'me'
```

The integer variable *stat_ctrl* encodes the state of the button with 1=up, 2=down, 3=greyed, and 4=push button. States 1 and 2 are used with a button that toggles up and down, i.e. a button that requires two separate clicks to return it to the up state. Each time the user presses and releases a button that is not greyed, the call-back function is invoked with call-back reason 'BUTTON_PRESS'. If you decide to use the same call-back function for several buttons you can use clearwin_info@('BUTTON_NUMBER') to establish which button was pressed (buttons are numbered along the rows starting at 1).

A call-back function (or any other code) can alter the control variable *stat_ctrl* and pass it to window_update@ to change the state of a button. For example, you could disable/enable a button. Passing any of the control variables belonging to a given %ib toolbar will update the entire block.

%ib[flat] produces a different style of button which pops up and becomes coloured when the mouse is moved over it.

Images are padded as required to produce a rectangular array, however for best results you should try to make the images the same size.

%ib is designed for bitmaps that have a grey background. If the image has a one-pixel border then the border colour is used as the background colour, otherwise the background is assumed to be light grey (192,192,192).

Notes If strings are included, you can make the buttons of equal width by making the images wide enough to allow for the strings.

When using High Color (16 bit) mode some display drivers convert grey (192,192,192)

to (200,200,200). To avoid this problem use (191,191,191) as the background colour.

Example See page 58.

See also %bt, %tt, %bx.

%ic

Purpose To draw an icon.

Syntax

```
winio@('%ic[icon_name]')
winio@('%`ic', handle)
integer handle           (input)
```

Modifiers Grave accent (`) - use a Windows API handle.

Caret (^) - the call-back function is called when the user clicks on the icon.

Question mark (?) - a help string is supplied.

Description *icon_name* is a standard character string (can be replaced by @) associated with a icon file in a resource script.

handle is a value returned by a modified form of the API function **LoadIcon**.

Icons can include transparent regions where the background can be seen.

Example

```
winio@('%^ic[myicon]', cb_func)
myicon ICON file.ico          (resource file)
```

See also make_icon@,%bm,%gi

%if

Purpose To specify that the next control will have the initial focus.

Syntax

```
winio@('%if')
```

See also add_cursor_monitor@, remove_cursor_monitor@, add_focus_monitor@, remove_focus_monitor@, clearwin_info@('FOCUSSED_WINDOW')

%il

Purpose To specify the lower and upper limits for subsequent %rd formats.

Syntax `winio@('%il', lower, upper)`
`winio@('%`il')`
`integer lower, upper (input)`

Modifiers Grave accent (%`il) - cancels an earlier %il and restores the limits to the minimum and maximum for the data type.

Description *lower* is the lower limit and *upper* is the upper limit.

Example `winio@('%il', 0L, 10L)`

See also %rd, %dd, %co[check_on_focus_loss]

%it

Purpose To switch to italic font.

Syntax `winio@('%it')`

Modifiers Grave accent (%`it) - used to switch off the effect.

Description %it is cancelled by %`it and %sf.

See also %bf %ul, %sf

%lc

Purpose To return the handle of the previous (last) control.

Syntax `winio@('%lc', handle)`
`integer handle (output)`

Description The returned value can be used in calls to Windows API functions. For example controls that do not have grey control variable can be greyed by using %lc to obtain the window handle and then calling the API function **EnableWindow**.

See also %hw

%ld

Purpose To display a round LED symbol (a coloured disk with a black border).

Syntax `winio@('%ld[options]', rgb_value)`
`integer rgb_value (input)`

Description `rgb_value` is usually set by a call to RGB@. By supplying a variable for `rgb_value` the value can be changed dynamically with a call to window_update@.

options can take the keyword square in order to change from a disk to a square shape and/or the keyword blinking to provide a flashing effect by switching between the given colour and black at approximately half second intervals.

Example `winio@('%ld', RGB@(255,0,0))`

%ls

Purpose To insert a list box or combo box.

Syntax `winio@('%n.mls[options]', items, num_items, cur_item)`
`winio@('%~n.mls[options]', items, num_items, cur_item,`
`grey_ctrl)`
`character*n items(num_items) (input)`
`integer num_items (input parameter)`
`integer cur_item (input/output)`
`integer grey_ctrl (input)`

Modifiers Grave accent (`) - used to provide a drop-down combo box (for editable lists see %el).

Caret (^) - call-back function is called when the user selects an item.

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

Description `n` and `m` are optional (`m` cannot be specified without `n`). `n` represents the width of the listbox in characters and `m` the depth. `n` and `m` can be replaced by asterisks (*) in which case the values of `n` and `m` must be supplied as the first two arguments in the

argument list.

num_items is the number of items in the list.

cur_item is the item number that is selected in the range zero to *num_items* both on input and output. Zero signifies that no item is selected.

items is an array of text entries, one for each item. Blank entries at the end of the list are not displayed. This means that you can use blank entries to create a list that can be changed dynamically.

If the grave accent is used and a call-back is supplied, *option* can be set to the keyword extended. The result is that the call-back is called when the list drops down or closes up. These events must be detected from the call-back by calling clearwin_string@('CALLBACK_REASON') which returns 'DROPDOWN' or 'CLOSEUP'.

For a list box (i.e. when a grave accent modifier is not used), the option hscrollbar provides a horizontal scroll bar whilst the option multicol column provides a multiple column output (see page 62).

If the grave accent modifier is not used, %ls can be preceded by a pivot %pv.

grey_ctrl is 1 (enabled) or 0 (disabled/greyed). Any call-back function must be placed after this control variable.

Notes The %ls control is limited to 32768 characters.

Example character*10 names(4)
winio@('%^`ls', names, 4L, sel, cb_func)

See also %el, %ms, %lv, %pb, %ps

%|v

Purpose To include a list view control.

Syntax winio@('%|v[options]', width, height, items, num_items, sel, view)
winio@('%`lv[options]', width, height, items, num_items, sel,
view, icon_str)
character*(*) items(num_items+1) (input/output)
integer num_items (input parameter)
integer sel(num_items) (input/output)
integer width, height, view (input)

```
character*(*) icon_str           (input)
```

- Modifiers** Grave accent (`) - used to provide a list of icons.
Caret (^) - call-back function is provided.
Question mark (?) - a help string is supplied.

Description A list view control is a window that displays a collection of items, each item consisting of an optional icon and a label. List view controls provide several ways of arranging items and displaying individual items. For example, additional information about each item can be displayed in columns to the right of the icon and label. This arrangement is called a report view. Microsoft Explorer uses a list view control in order to display directory information.

width and *height* provide the dimensions of the control in pixels.

num_items is an integer constant giving the number of rows in the report view. Blank rows are not displayed in the control. This means that the number of rows that are displayed can be changed under program control. The first blank row terminates the input.

items is an array of *num_items*+1 character strings the first describing the column headers and widths, then one string describing each row in turn (see the example below).

sel is an integer array whose elements are set to 1 if the item is selected otherwise to zero.

view is an integer variable in the range 0..3 that represents the type of view. 0=Icon view; 1=Report view; 2=Small icon view; 3=List view. These are available even when no icons are provided.

If a grave accent is provided, *icon_str* is a character string giving a list of icon resources in the form 'icon1.icon2'.

%lv can take a pivot (%pv). Also the right mouse button can be used to trigger a popup menu (%pm) defined in the parent window.

Example

```
CHARACTER*80 items(4)
INTEGER sel(3),view
items(1)='|Header_100|_50'
items(2)='|AItem1|Data1'
items(3)='|BItem2|Data2'
items(4)='|BItem3|Data3'
sel(1)=1;sel(2)=0;sel(3)=0;view=1
i=wini@('%`%lv',300,200,items,3,sel,view,'icon1,icon2')
```

Here we have 3 rows with 2 columns. *items(1)* provides data for the column headings

and widths. The first character defines the separator between the data for each column. In this example, the first column has the caption "Header". A trailing underscore followed by an integer provides the pixel width of the column. If this is omitted then the length of the text defines the width. In this example the second column header is blank and the second column width is 50 pixels. The number of characters (80 in this case) must be sufficient to allow ClearWin+ to add the column width to each heading (4 characters per column).

items(2) provides data for the first row of a report. Again the first character defines the separator. The second character defines the icon as an index (A,B,C...) into the list of icons 'icon1,icon2' (when a grave accent is used). Next comes the label and then the data for column two etc..

Options

<code>single_selection</code>	Allows only one item at a time to be selected. By default Shift and Control keys can generate multiple selections.
<code>edit_labels</code>	Allows item text to be edited (first column in report view).
<code>no_column_headers</code>	Column headers are not displayed in report view (<i>items(1)</i> is still needed for column widths)
<code>show_selection_always</code>	Always show the selection, if any, even if the control does not have the focus.
<code>no_border</code>	Displays the control without a border.
<code>no_label_wrap</code>	Displays the item label on a single line in an icon view.
<code>align_top</code>	Items are top-aligned in icon and small icon view.
<code>align_left</code>	Items are left-aligned in icon and small icon view
<code>edit_cells</code>	Allows all columns in the report view to be edited (see below).

Label editing

`edit_labels` enables label editing and requires a call-back function that can test for the call-back reasons '`BEGIN_EDIT`', '`END_EDIT`' and '`EDIT_KEY_DOWN`'.

With `BEGIN_EDIT` you can get the index of the selected item by the call `clearwin_info@('ROW_NUMBER')`. If your call-back function returns the value 4 then editing of this item is prevented.

With `EDIT_KEY_DOWN` you can validate each character as it is typed. Use `clearwin_info@('KEYBOARD_KEY')` to get the key and/or `clearwin_string@('EDITED_TEXT')` to get the resulting string with the character inserted. If the call-back function returns the value 4 then the character is not accepted. If you want to translate the input key to another key then the call-back function must return the ASCII character code (31<code<256) of the translated key.

With `END_EDIT` you can get the result of the edit by the call `clearwin_string@('EDITED_TEXT')`. Having validated the label, you should then

get the index using `clearwin_info@('ROW_NUMBER')` and reconstruct the relevant row of the `items` array by adding an icon identifier and extra column data as required (otherwise the original data will be restored). Creating a blank row in response to a label edit is not recommended.

Selecting a view

It is possible to create a menu item or a toolbar button that changes the view state (Icon view, Report view, Small icon view, List view). Here is some sample code for the associated call-back function.

```
INTEGER FUNCTION change_view()
  INTEGER view
  COMMON view
  view=view-1
  IF(view < 0) view=3
  CALL window_update@(view)
  change_view=2
END
```

Column widths

If the last column does not extend to the right of the control then ClearWin+ will adjust the width of this column so that it fills the area.

If a column width is changed at run time then ClearWin+ stores the new width in row 1 of the data. This means that the adjusted widths will be used when the control is redrawn. If you want to use the same widths when the program is re-used then you will need to save and restore the modified data.

Cell editing

The option `edit_cells` is effective in a report view where it over-rides `edit_labels`. By using `edit_cells` you can edit the data in any column. A grid is displayed resulting in a spreadsheet appearance. The call-back function can be used to prevent the editing of any given cell and to validate each input character and as well as the final result. For this purpose `edit_cells` uses the call-back reasons `BEGIN_EDIT`, `EDIT_KEY_DOWN` and `END_EDIT` etc. as in `edit_labels`. The only difference is that unlike `edit_labels`, `edit_cells` initially responds to a single mouse click.

`clearwin_info@('ROW_NUMBER')` and `clearwin_info@('COLUMN_NUMBER')` identify the cell that is being edited.

When editing cells, use the arrow keys to move vertically and horizontally and **CTRL** with arrow keys to move horizontally once editing has commenced. This movement will step over any cells that are disabled. The selection array (with one element for each row) contains the row number of the current selection. However, when validating input it is simpler to use `clearwin_info@('COLUMN_NUMBER')`.

Call-back reasons

BEGIN_EDIT	A label or cell is about to be edited (see above).
EDIT_KEY_DOWN	The user has pressed a keyboard key during a label or cell edit. <code>clearwin_info@('KEYBOARD_KEY')</code> provides the key code (see above).
END_EDIT	A label or cell has been edited (see above).
SET_SELECTION	A item has been selected.
COLUMN_CLICK	<code>clearwin_info@('ROW_NUMBER')</code> provides the index.
MOUSE_DOUBLE_CLICK	The user has clicked on a column header.
	<code>clearwin_info@('COLUMN_NUMBER')</code> provides the index.
KEY_DOWN	The user has double-clicked using the left mouse button.
	<code>clearwin_info@('ROW_NUMBER')</code> provides the index.
	The user has pressed a keyboard key.
	<code>clearwin_info@('KEYBOARD_KEY')</code> provides the key code.

In response to a column click you might choose to sort the items using the given column index. Having reconstructed the data, you should then make a call to `window_update@(items)` in order to view the new order.

Example See page 65.

See also %ms, %ls, %el

%lw

Purpose To allow `winio@` to return without closing the window that it creates.

Syntax `winio@('%lw[option]', ctrl)`
 integer `ctrl` (output)

Modifiers Grave accent (`) - used to delay the creation of the window until `ctrl` has been passed to %ch.

Description Using %lw causes `winio@` to return whilst leaving the window open. `ctrl` is a control variable that is returned with the value -1. Subsequently the window can be closed by setting `ctrl` to zero (or a positive value) and calling `window_update@(ctrl)`.

%lw can take the option owned (%lw[owned]). This option defines certain properties of the window in relation to its parent. The “owned” properties are those of a window that does not use %lw, namely that the parent cannot overlay the child and, if the parent is closed, the child automatically closes. For these properties to be meaningful

the parent must be created using %ww. Also %lw[owned] must be used in the definition of a child window that appears in a call-back function of the parent.

See also %ch

%mg

Purpose To provide a call-back function for a given Windows message.

Syntax

```
winio@( '%mg', msg_no, cb_func)
integer msg_no      (input)
external cb_func
```

Description This format provides direct access to specified Windows messages. It should be used with care since it is possible to interfere with other ClearWin+ processing.

msg_no is the message number of the message to be processed. User numbers should begin at WM_USER +1000. *cb_func* is the call-back function that is to be called when the message is received. The call-back function can use clearwin_info@ with the strings 'MESSAGE_HWND', 'MESSAGE_WPARAM', and 'MESSAGE_LPARAM' in order to get the standard arguments for a Windows call-back function.

When writing the %mg call-back function, use a return value of 3 if you want ClearWin+ to apply the default processing of the message *msg_no*. Other values have the usual effect (zero closes the window etc.) but cause the default processing to be bypassed. Occasionally the internal Windows dialog procedure that processes messages requires a particular return other than the default value which is zero (e.g. under certain conditions when using WM_NOTIFY). In this case you should make a call to the subroutine set_mg_return_value@ before returning. This subroutine takes one integer argument which is the value to be returned by the internal Windows dialog procedure.

See also %rm

%mi

Purpose To supply the name of an icon resource to be used when the window is minimised.

Syntax

```
winio@( '%mi[icon_name]')
winio@( '%`mi', handle)
```

```
integer handle    (input)
```

Modifiers Grave accent (`) - use a Windows API handle.

Description *icon_name* is a standard character string (can be replaced by @) associated with the name of an icon file in a resource script.

handle is a value returned by a modified form of the API function **LoadIcon**.

Example `winio@('%mi[myicon]')`
`myicon ICON file.ico` (resource file)

See also `make_icon@, %ic`

%mn

Purpose To attach a menu to the window.

Syntax `winio@('%mn[menu_spec]', . . .)`

Modifiers Question mark (?) - All the strings are placed in order at the end of the format description, one for each and every menu item that has a call-back function.

Grave accent (`) -Creates an enhanced menu (see page 86).

Description See page 81.

See also `%pm, %sm`

%ms

Purpose To include a multi-selection box.

Syntax `winio@('%n.mms[options]', items, num_items, sel)`
`character*n items(num_items)` (input)
`integer num_items` (input parameter)
`integer sel(num_items)` (input/output)

Modifiers Grave accent (`) - used to provide multiple selection using the CTRL and SHIFT keys and mouse dragging.

Caret (^) - call-back function is called when the user selects an item.

Question mark (?) - a help string is supplied.

Description *n* and *m* are optional (*m* cannot be specified without *n*). *n* represents the width of the listbox in characters and *m* the depth.

The list of possible *options* is currently empty. This item is included for future developments. If you include a help string in square brackets then you must also include an empty options list.

num_items is the number of items in the list. If *m* is less than *num_items* then a scroll bar is presented.

sel is an array whose elements are set to 1 if the item is selected otherwise to zero.

items is an array of text entries, one for each item. Blank entries at the end of the list are not displayed. This means that you can use blank entries to create a list that can be changed dynamically.

%ms will take a pivot %pv.

Example

```
character*10 names(4)
integer sel(4)
winio@('%^`ms', names, 4L, sel, cb_func)
```

See also %ls, %el, %lv, %pb, %ps

%mv

Purpose To provide a call-back function that is to be called when the user moves or resizes the window.

Syntax

```
winio@('%mv', cb_func)
external cb_func
```

Modifiers Grave accent () - without the grave the call-back is for a moving window; with the grave the call-back is for a resizing window.

Description The call-back function is also called when the window is created (if two are used, one for moving and one for re-sizing, then both are called).

The call-back function will typically call the Clearwin+ function get_window_location@ in order to determine the new size and position.

Note that certain window operations involve both moving and resizing. For example if the user drags the top-left corner, then the window is both moved and resized ('moving' changes the co-ordinates of the top left corner, 'resizing' changes the width

and/or height).

See also [get_window_location@](#)

%nc

Purpose To provide a specified class name for the application.

Syntax `winio@(%nc[class_name])`

Description *class_name* is a standard character string (can be replaced by @) that specifies the new class name. Some applications display the class name in their output.

Notes The Windows API function **FindWindow** can be used in order to get a handle to another application from its class name. The handle can then be used to target messages. If you want the current application to act as such a target you can specify the class name using %nc.

Example `winio@(%nc[myappclass])`

See also [send_text_message@, %rm](#)

%nd

Purpose To prevent controls from sliding down when sizing a window.

Syntax `winio@(%nd)`

Modifiers Grave accent (`) - used to switch off the effect.

Description Used with %pv and %ww under the old re-sizing mechanism (see page 93).

See also [set_old_resize_mechanism@, %ww, %nr](#)

%nl

Purpose To insert new lines.

Syntax `winio@(%nl)`

Description *m* is the number of new lines to insert. *m* is optional and defaults to 1. To move directly below a control or graphics area use %ff instead.

Example `winio@('3nl')`

See also %ff, %cn, %rj

%nr

Purpose To prevent controls from sliding right when sizing a window.

Syntax `winio@('%nr')`

Modifiers Grave accent (`) - used to switch off the effect.

Description Used with %pv and %ww under the old re-sizing mechanism (see page 93).

See also set_old_resize_mechanism@, %ww, %nd

%ns

Purpose To disable a screen saver whilst the current window is open.

Syntax `winio@('%ns')`

See also %sv

%ob

Purpose To open a box.

Syntax `winio@('%ob[options][title]')`
`winio@('%n.mob[options][title]')`

Modifiers Grave accent (`) - provides a shaded background.

Description Defines the top left hand corner of a rectangular box into which subsequent objects are to be placed until a corresponding %cb format is encountered.

title is optional and is a standard character string that is used with an option from the

options list that sets a title to the box.

options is can be one or more of:

named_c	Splices a name into the top line of the box (in the centre)
named_l	Splices a name into the top line of the box (on the left).
named_r	Splices a name into the top line of the box (on the right)
named	The same as named_c.
nameless	Occupies the same vertical space as named_c etc. but has no name.
no_border	Create an invisible box which is useful for grouping controls.
invisible	The same as no_border.
panelled	This replaces the simple line box with a 3-dimensional panel.
thin_panelled	Similar to panelled but perhaps a slightly more attractive effect.
shaded	This has the same effect as the use of the grave accent.
status	This provides a box in the form of a status bar.
bottom_exit	Begins the next control below this box rather than to the right.
scored	Double line, 3-dimensional shade effect.
depressed	Single line, 3-dimensional shade effect.
raised	Single line, 3-dimensional shade effect

When *n* and *m* are used, %ob creates a rectangular grid of boxes (*n* across, *m* down) each requiring its own %cb. Thus %3.2ob would require six %cb codes. A title cannot be supplied for a grid of boxes. When used with the option no_border, this provides a mechanism for positioning text and controls within rectangular sub regions of the window (see the section on page 91 for a simple example).

%ap can be used before %ob in order to specify the position of the box from the %ap arguments.

Notes %ob boxes can be nested, one within another, in order to provide a fine control over the positioning of items in a window.

The options scored, depressed and raised should not be used with a background colour of white or black (grey is typical).

Example winio@('%ob[named_r][Results]')

See also %ap

%og

Purpose To create an OpenGL graphics region (Win32 only).

Syntax `winio@(%og[options], width, height)`
`integer width, height (input)`

Description %og takes two arguments namely the pixel width and pixel height of the region and sets up an OpenGL graphing region. %og only works under Win32, and requires the presence of associated DLLs. See Chapter 16 for further details.

%pb

Purpose To insert a parameter box.

Syntax `winio@(%N.Mpb[options])`

Modifiers Question mark (?) - can be attached to %dp, %fp, %tp, %ep and %up.

Description N is the width of the box in average characters. M is the number of lines in the control which will include a scroll bar if M is not large enough.

options can be omitted or can take the value sorted to specify that the parameters are to be alphabetically sorted by name.

After a parameter block has been defined parameters can be attached by subsequent formats thus:

- %dp - Integer parameter
- %fp - Floating point parameter
- %tp - Text parameter
- %ep - Enumerated parameter (a set of named alternatives)
- %up - User parameter (just activates a call-back)

Each of these codes is followed by a *standard character string* that provides the text to be displayed. See page 42 for further information.

See also %ls, %ms, %ps

%pd

Purpose To insert a Sterling pound symbol ‘£’.

Syntax `winio@('%pd')`

%pl

Purpose To create a SIMPLEPLOT graphics region (Win32 only).

Syntax `winio@('%pl[options]', width, height)`
 `integer width, height (input)`

Modifiers Question mark (?)

Description In order to use the SIMPLEPLOT graphics drawing library, the executable must have access to a DLL called *simple.dll*. This DLL is supplied with Salford compilers. When making direct calls to the SIMPLEPLOT library (see the `user_drawn` option), the program must also be linked using the DLL.

width and *height* represent the pixel dimensions of a SIMPLEPLOT graphics region within the window. Information about the options that are available is given in Chapter 17. Some of the options take additional arguments. Data that is real must be passed as double precision values despite the fact that the SIMPLEPLOT library uses only single precision arithmetic.

Data that is provided via additional arguments should normally be available throughout the life time of the window. This enables the window to be re-drawn, for example after a re-sizing operation.

See also `%gr`, `%og`, `%dw`

%pm

Purpose To supply a popup menu.

Syntax `winio@('%pm[menu_spec]', ...)`

Modifiers Question mark (?) - All the strings are placed in order at the end of the format description, one for each and every menu item that has a call-back function.

Grave accent (`) -Creates an enhanced menu (see page 86).

- Description** Provides a popup menu so that when the right mouse button is pressed in the main window a menu appears. The menu does not appear when the mouse points to a control within the main window.
- See page 84 for further details.
- See also** %mn, %sm

%ps

- Purpose** To produce layered windows in a card-index style (property sheet).
- Syntax**
- ```
winio@(%Nps', sheet1, sheet2, . . . , sheetN)
winio@(%`Nps', sheet1, sheet2, . . . , sheetN, sheet_n)
integer sheet1, sheet2, . . . , sheetN (input)
integer sheet_n (input/output)
```
- Modifiers** Grave accent (`) - provides for a handle (*sheet\_n*) to set and return the current sheet number. *sheet\_n* must be SAVED or global (e.g. in a module or common block).  
Caret (^) - the call-back function is called each time the sheet is changed and when the first sheet is initially displayed.
- Description** *N* is the number of sheets. *sheet1* to *sheetN* are handles for the various sheets returned by %sh.  
Each sheet is set up using a complete winio@ format and %sh.  
clearwin\_info@( 'SHEET\_N0' ) can be used to return the current visible sheet number which is initially set to one. Alternatively, a grave accent modifier can be supplied in order to set the initial sheet (at a value other than one) and to change the sheet number under program (rather than user) control. The program can change the sheet number by changing *sheet\_n* and calling see\_propertysheet\_page@( *sheet\_n* ).  
%ca can be used with %sh to provide a caption. The caption can include an accelerator key indicator '&' which operates in the same way as in a menu or on a button to select the sheet.  
If a call is made to the Windows API function **EnableWindow** to disable one sheet, then that sheet will no longer be selectable unless/until the window is re-enabled. If the sheet that is currently visible is disabled, another sheet will not automatically be selected. However, when another sheet is selected, a disabled sheet will not be re-selected.

selectable.

**Notes** If an individual sheet is closed (e.g. by placing a button without a callback function in the sheet) then the parent window will close.

**See also** %sh

## %pv

**Purpose** To allow certain controls to be re-sized under user control.

**Syntax** `winio@(%'pv')`

**Description** Use %pv before %eb, %fr, %dw, %gr, %cw, etc. so that the control can be re-sized at run time. A thick frame is added to the window to enable it to be resized.

**See also** %eb, %fr, %dw, %gr, %cw

## %rb

**Purpose** To define a radio button or check box.

```
Syntax winio@('%rb[button_text]', ctrl)
 winio@('%~rb[button_text]', ctrl, grey_ctrl)
 integer ctrl (input/output)
 integer grey_ctrl (input)
```

**Modifiers** Grave accent (`) - provides a check box rather than a radio button.

Caret (^) - call-back is called when the control is selected.

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

**Description** *button\_text* is a standard character string (can be replaced by @) for the text attached to the radio button. To change the text dynamically use %lc and the API function **SetWindowText**.

*ctrl* is 1 (*on*) or 0 (*off*).

*grey\_ctrl* is 1 (enabled) or 0 (disabled/greyed). A call-back function (if present) is placed after this control variable.

%rb can be used in conjunction with the gang format code %ga. Note that it is not conventional to gang check boxes.

**Example** `winio@('%rb[Button 1]', rb1)`

**See also** %ga

## %rd

**Purpose** To create an edit box and display an integer that can be updated.

**Syntax** `winio@('%nrd[option]', value)`  
`winio@('%~nrd[option]', value, grey_ctrl)`  
`integer value` (input/output)  
`integer grey_ctrl` (input)

**Modifiers** Grave accent (`) - makes the control read-only (no box is supplied, see %co).

Caret (^) - the call-back is called when a change is made (see also %co).

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

**Description** *n* is optional and specifies the number of average width characters in the displayed output. Only values in range can be entered (see %co). The range can be set using %il.

If %`rd is specified, then the value presented can only be changed by the program using `window_update@(value)`.

*grey\_ctrl* is 1 (enabled) or 0 (disable/greyed). A call-back function (if present) is placed after this control variable.

*option* can be set to the keyword `initially_blank` in order to create an edit box that is initially empty.

The standard call-back functions 'COPY', 'CUT', and 'PASTE' can be used in this context by attaching them to menu items and/or accelerator keys.

When %dd is used before %rd, a spin wheel is added (see %dd for details).

The subroutine `set_highlighted@` can be called to select all of the text in the edit box. It takes one argument which is the handle given by %lc.

**See also** %il, %dd, %co, `set_highlighted@`

## %re

**Purpose** To create an multi-line edit box and display a character variable (i.e. a string).

**Syntax**

```
winio@('%N.Mre[options]', buffer)
winio@('%~N.Mre[options]', buffer, grey_ctrl)
character*(*) buffer (input/output)
integer grey_ctrl (input)
```

**Modifiers** Grave accent (`) - makes the control read-only (scroll bars are disabled).

Caret (^) - the call-back is called when a change is made.

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

**Description** This control is similar to %rs but is multi-line. It has a buffer limit of 32K characters.

*N* represents the width of the box in average characters. *M* represents the depth which should not be less than 2.

*buffer* is a character string for the text (maximum of 32\*1024 characters).

The following options are available:

|             |                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------|
| UPPERCASE   | Alphabetical characters (a . . z) are converted to upper case and displayed and stored as upper case.                                |
| LOWERCASE   | Alphabetical characters (A . . Z) are converted to lower case and displayed and stored as lower case.                                |
| CENTRE_TEXT | Text is always centred on every line.                                                                                                |
| RIGHT_TEXT  | Text is right justified on every line (the default is left justified text).                                                          |
| NO_HSCROLL  | The text will not scroll to the right past the end of the window.<br>The text will auto wrap.                                        |
| NO_VSCROLL  | The text will not scroll beyond the bottom of the window.                                                                            |
| HSCROLLBAR  | A horizontal scroll bar is added - do not use with NO_HSCROLL.                                                                       |
| VSCROLLBAR  | A vertical scroll bar is added - do not use with NO_VSCROLL.                                                                         |
| KEEP_FOCUS  | When the user tabs between windows, normally the focus will be lost, however with KEEP_FOCUS the selected text will remain selected. |

**Notes** The %rs PASSWORD option is not available with %re. %re does not take a pivot (%pv).

%co can be used with %re.

**See also** %rs, %eb, %co

## %rf

**Purpose** To create an edit box and display a floating point value that can be updated.

**Syntax**

```
winio@(%nrf[option], value)
winio@(%~nrf[option], value, grey_ctrl)
double precision value (input/output)
integer grey_ctrl (input)
```

**Modifiers** Grave accent (`) - makes the control read-only (no box is supplied, see %co).

Caret (^) - the call-back is called when a change is made (see also %co).

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

**Description** *n* is optional and specifies the number of average width characters in the displayed output. Only values in range can be entered (see %co). The range can be set using %fl. Values are entered in decimal or exponent form.

If %`rf is specified, then the value presented can only be changed by the program using window\_update@(value).

*grey\_ctrl* is 1 (enabled) or 0 (disable/greyed). A call-back function (if present) is placed after this control variable.

*option* can be set to the keyword initially\_blank in order to create an edit box that is initially empty.

The standard call-back functions 'COPY', 'CUT', and 'PASTE' can be used in this context by attaching them to menu items and/or accelerator keys.

When %df is used before %rf, a spin wheel is added (see %df for details).

The subroutine set\_highlighted@ can be called to select all of the text in the edit box. It takes one argument which is the handle given by %lc.

**See also** %fl, %df, %co, set\_highlighted@

## %rj

**Purpose** To right justify text and controls.

**Syntax** `winio@('%rj')`

**Description** %rj forces everything that follows it, up to the next new line or form-feed character, to be right justified in the window. A window margin, if any, is still applied.

When %rj is used within a box (%ob), the format applies only to that box.

**See also** %cn, %nl, %ff

---

## %rm

**Purpose** To provide the name of a call-back function to handle messages from another ClearWin+ application.

**Syntax** `winio@('%rm', cb_func)`  
`external cb_func`

**Description** If one application uses the subroutine `send_text_message@` to send text to a second application, the second application should use %rm to supply the name of a call-back function to handle the message and send a reply.

**See also** `send_text_message@`, `reply_to_text_message@`,  
`clearwin_string@('MESSAGE_TEXT')`.

---

## %rp

**Purpose** To set the position of the next control relative to the current position.

**Syntax** `winio@('%rp', ix, iy)`  
`integer ix, iy (input)`  
`winio@('%`rp', rx, ry)`  
`double precision rx, ry (input)`

**Modifiers** Grave accent (`) - use real rather than integer input.

**Description** The units of measurement correspond to the average width and the height of the

default font.

**Example** `winio@(%rp', 8L, 0L)`

**See also** `%ap`, `%gd`, `%ob`, `%nl`, `%ff`

## %`rs

**Purpose** To create an single-line edit box and display a character variable (i.e. a string).

**Syntax** `winio@(%nrs[options], string)`  
`winio@(%~nrs[options], string, grey_ctrl)`  
`character*(*) string` (input/output)  
`integer grey_ctrl` (input)

**Modifiers** Grave accent (`) - makes the control read-only (no box is supplied, see %co).

Caret (^) - the call-back is called when a change is made (see %co).

Question mark (?) - a help string is supplied.

Tilde (~) - adds a variable that controls the grey (enable/disable) state.

**Description** For the equivalent multi-line edit box see %re.

*n* is optional and specifies the number of average width characters in the displayed output. The output will scroll horizontally within the specified width if necessary.

If %`rs is specified, then the string presented can only be changed by the program using `window_update@(value)`.

*grey\_ctrl* is 1 (enabled) or 0 (disable/greyed). If a call-back function is used, it is placed after this control variable.

The standard call-back functions 'COPY', 'CUT', and 'PASTE' can be used in this context by attaching them to menu items and/or accelerator keys.

When %dd is used before %rs, a spin wheel is added (see %dd for details).

The subroutine `set_highlighted@` can be called to select all of the text in the edit box. It takes one argument which is the handle given by %lc.

The following options are available:

PASSWORD

An asterisk (\*) appears on the screen for each character typed whilst the string is correctly stored.

|           |                                                                                                       |
|-----------|-------------------------------------------------------------------------------------------------------|
| UPPERCASE | Alphabetical characters (a . . z) are converted to upper case and displayed and stored as upper case. |
| LOWERCASE | Alphabetical characters (A . . Z) are converted to lower case and displayed and stored as lower case. |

**See also** %st, %co, %re, %eb, set\_highlighted@

## %SC

**Purpose** To provide a call-back function when a window is initially displayed.

**Syntax** `winio@(%sc', cb_func)`  
`external cb_func`

**Description** Causes a given call-back to be called once only when a window is initially displayed (for example, for drawing initial %gr data).

**Notes** If %gr (say) has a call-back function that receives the RESIZE message from clearwin\_info@('CALLBACK\_REASON'), this message is received before the %sc call-back is called.

Call-back functions can often be shared by different format codes within one format window. For example, a call-back associated with %sc may also be suitable for %mv.

## %SD

**Purpose** To provide subscript text.

**Syntax** `winio@(%sd')`

**Modifiers** Grave accent (`) - cancels the effect of an earlier %sd.

**Description** This format code cannot be applied to the default font so select a font and then enclose the text that is to be subscripted between %sd and %`sd.

**Example** `winio@('%fn[arial]Water is H%sd2%`sd0')`

**See also** %su

## %Sf

**Purpose** To return to the standard font.

**Syntax** `winio@( '%sf' )`

**Modifiers** Grave accent (`) - Windows 95 only (see page 98).

**Description** Resets to default text attributes after use of any combination of %bf, %it, %ul, %fn, and %ts.

**See also** %bf, %it, %ul, %fn, %ts, `use_windows95_font@`

---

## %Sh

**Purpose** To create a property sheet.

**Syntax** `winio@( '%sh', handle )`  
integer *handle* (output)

**Description** Each time %sh is used, it creates a window that is to be linked to %ps. The returned value of *handle* is an input value for %ps.

Each sheet can be provided with a caption using %ca. In this situation the caption can include an ampersand (&) to mark an accelerator key.

**See also** %ps, %ca

---

## %Si

**Purpose** To insert a standard icon.

**Syntax** `winio@( '%nsisymbol' )`

**Modifiers** Caret (^) - call-back is called when the user clicks on the icon.

Question mark (?) - the help string goes after the *symbol*.

**Description** Defines a standard icon that is to be placed to the left of the block of text that follows %si.

*symbol* is one of:

|   |                           |
|---|---------------------------|
| ! | hazard ‘!’ road sign      |
| ? | question mark in bubble   |
| * | information ‘i’ in bubble |
| # | stop ‘X’ on disc.         |

*n* is optional and can be added in order to centre the icon vertically to the left of *n* lines of text.

**Example** `winio@('%si#')`

**See also** `%ic`

## %SL

**Purpose** To insert a slider control.

**Syntax** `winio@('%Ns1[option]', value, min, max)`  
`double precision min, max` (input)  
`double precision value` (input/output)

**Modifiers** Caret (^) - call-back is called repeatedly as the user moves the slider or when an arrow key is pressed.

Question mark (?) - a help string is supplied.

**Description** *N* is mandatory and sets the length of the control in average character units.

*value* is the current value represented by the slider in the range *min* to *max*. This value can also be accessed by a call-back function whilst the slider is being moved.

*option* can be set to the keyword `vertical` to provide an alternative to the default which is a horizontal slider.

The slider will respond to the arrow keys on the keyboard.

The value returned by `clearwin_info@('LATEST_VARIABLE')` can also be used with %sl.

**Example** `winio@('%20s1', x, 0.0D0, 10.0D0)`

**See also** `%br`

## %SM

**Purpose** To modify the system menu.

**Syntax** `winio@(%sm[menu_spec], . . .)`

**Modifiers** Grave accent (`) - used to replace rather than add to the existing system menu.

Question mark (?) - All the strings are placed in order at the end of the format description, one for each and every menu item that has a call-back function.

**Description** See page 86.

**See also** %mn, %pm

---

## %SP

**Purpose** To set the position of a window.

**Syntax** `winio@(%sp, x, y)`  
integer *x, y* (input)

**Modifiers** Grave accent (`) - used in a call-back function to give the position relative to the position of the event that resulted in the call.

**Description** *x* and *y* represent the position of the top left corner of the window in screen co-ordinates. Typically these values are returned by %gp.

**Example** `winio@(%sp, 200L, 100L)`

**See also** %gp

---

## %SS

**Purpose** To save and load settings using an INI file.

**Syntax** `winio@(%ss[filename/section], ctrl)`  
integer *ctrl* (input)

**Description** %ss is used before %rd, %rf, or %rs in order to automatically save and restore data in these controls.

*filename/section* is a standard character string (can be replaced by @).

*filename* is the stem of a file with the .INI extension that is stored in the standard Windows directory.

*section* is the name of a section in the INI file. This name appears in square brackets at the head of a section in the file.

The item name for the data is automatically formed from the text that appears immediately before the control. Spaces are removed from the text in order to form the name. See page 48 for further details.

*ctrl* is set to 1 to enable saving and to 0 to disable saving.

**Example** `winio@('%ss[app1/block2]', ctrl)`

**See also** `%rd`, `%rf`, `%rs`

## %St

**Purpose** To insert a variable string.

**Syntax** `winio@('%N.mst', string)`  
`character *(*) string` (input)

**Modifiers** Grave accent (`) - right justifies the string in the field.

**Description** Lays out a string out in a field *N* average characters wide and *m* deep. *m* is optional and defaults to 1 but *N* must be specified. The string is read-only (unlike %rs, the string cannot be changed by the user). It can, however, be changed by the programmer. The change is followed by a call to `window_update@(string)` in order to make a change visible. The string is re-drawn each time the window is renewed. (%`rs provides a similar effect.)

When used as the last item in a status bar, the control will automatically expand to fill the remaining space.

**Example** `winio@('%20st', str)`

**See also** `%`rs`

## %SU

**Purpose** To provide superscript text.

**Syntax** `winio@( '%su' )`

**Modifiers** Grave accent (`) - cancels the effect of an earlier %su.

**Description** This format code cannot be applied to the default font. You must select a font and then enclose the text that is to be superscripted between %su and %`su.

**Example** `winio@( '%fn[arial]Einstein said E=MC%su2%`su.' )`

**See also** %sd

---

## %SV

**Purpose** To create a screen saver executable.

**Syntax** `winio@( '%sv' )`

**Description** A window whose definition includes %sv will close as soon as it receives mouse or keyboard input. See page 132 for further information.

**See also** %ns

---

## %SY

**Purpose** To change the style of a *dialog box*.

**Syntax** `winio@( '%sy[ options ]' )`

**Description** By default, a format window that uses neither %ww nor %sy has a caption bar but does not have: a) system menu, b) minimise box and c) a maximise box. Such a window is called a *dialog box* and by default has a double frame that is raised. This means that a *dialog box* cannot be maximised, minimised or re-sized. %sy is used to change the style of such a *dialog box*. The list includes options for changing the *border* of the window. This refers to the space that, by default, is provided by ClearWin+ to surround the text and controls within a window (as in %bd). Do not confuse this *border* with the *frame* of the window that is determined by the Windows API style.

At least one of the following options must be included:

---

|                           |                                                                                                                                                                                                                                      |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>no_border</code>    | Removes the white border - like <code>%ww[no_border]</code>                                                                                                                                                                          |
| <code>thin_border</code>  | Produces a half-size white border.                                                                                                                                                                                                   |
| <code>3d</code>           | Provides a three-dimensional style for subsequent controls in the window (see below).                                                                                                                                                |
| <code>3d_thin</code>      | Like <code>3d</code> but provides a Windows 95 style edge for controls.                                                                                                                                                              |
| <code>3d_raised</code>    | Like <code>3d</code> but provides a raised thin edge for controls.                                                                                                                                                                   |
| <code>3d_depressed</code> | Like <code>3d</code> but provides a depressed thin edge for controls.                                                                                                                                                                |
| <code>no_sysmenu</code>   | Omit the box at the top left which can be used to produce the system menu.                                                                                                                                                           |
| <code>independent</code>  | Makes the window independent of other windows in the program that are already open. This can be used to create several windows between which the user can move at will (that is, there is no implied window hierarchy in this case). |

---

The `3d` style has an effect on the position of subsequent controls and must be used before the controls are defined. Some controls are not affected (for example, buttons are not changed). The window is created using the standard button face colour for its background with the controls appearing in relief. With `%sy[3d]`, by default `%ob` boxes are given the `scored` property provided the `raised` and `depressed` properties are not used with `%ob`. `%ob[shaded]` boxes are rendered as `%ob[depressed]` boxes.

**See also** `%ww`

---

## **%SZ**

**Purpose** To size a window.

**Syntax** `winio@('%sz', width, height)`  
`integer width, height` (input/output)

**Description** `width` and `height` are variables whose current values specify the pixel width and height of the window. When non-zero values are supplied, these are used to specify the initial width and height of the window. If the window is re-sized (see `%ww`) then the variables `width` and `height` are automatically updated accordingly.

Alternatively, if zero values are supplied, the window is given a default size that suits its contents. `width` and `height` will be updated to these values and will also be changed

when the window is re-sized. If code is added to save these settings, then the values can be used when the same window is next opened.

Special values are used to represent a maximised window. Consequently, if the dimensions of a maximised window are saved and used to re-open the window, the window will be restored in its maximised state.

**See also** %sp

---

## %ta

**Purpose** To skip to the next tab stop.

**Syntax** `winio@( '%nta' )`

**Description** Tab stops can be used for text and for controls. Default tab stops are at 8-average character width intervals. The default settings can be changed by using %tl.

*n* is an optional positive integer which defaults to 1. It provides an alternative to multiple consecutive applications of %ta.

**Example** `winio@( '%3ta' )`

**See also** %tl

---

## %tb

**Purpose** To insert a bitmap button or toolbar.

**Syntax** `winio@( '%tb', off, on, down, stat_ctrl )`  
`winio@( '%~tb', off, on, down, disb, stat_ctrl, grey_ctrl )`  
`character*(*) off, on, down, disb (input)`  
`integer stat_ctrl, grey_ctrl (input/output)`  
`winio@( '%n.mtb', . . . )`

**Modifiers** Tilde (~) - adds a grey control option.

Caret (^) - the call-back function is called when the user clicks on the button. The function name is added to the end of the list of arguments for a given button.

Question mark (?) - A help string for each button is placed in order (in square brackets) after 'tb'.

**Description** This control has been superceded by %ib. For new code use %ib instead.

*off*, *on*, *down* and *disab* provide the names of bitmap resources through a resource script.

*stat\_ctrl* is set to 1 when the button is *on* and to 0 when the button is *off*.

*grey\_ctrl* is set to 1 when the button is *enabled* and to 0 when it is *disabled* (greyed).

*n* and *m* are optional and both default to 1. When *n* is used *.m* is optional. These provide for a toolbar *n* buttons wide and *m* deep. A full set of arguments are required for each button (scanning in order across the rows of buttons) using the same order of arguments as for a single button.

For further information see page 68.

**Notes** Bitmap resource names can be contructed from known Windows handles (see page 79).

**Example**

```
winio@('%tb', 'off', 'on', 'dwn', stat)
 off BITMAP bt1.bmp (resource file)
 on BITMAP bt2.bmp
 dwn BITMAP bt3.bmp
```

**See also** %ib, %bt, %tt, %ga, %bx

## %otc

**Purpose** To set the colour of subsequent text.

**Syntax**

```
winio@('%tc[colour]')
 winio@('%tc', rgb_value)
 integer rgb_value (input)
```

**Modifiers** Grave accent (`) - used with an integer argument in order to set the text colour for %eb (see page 113).

**Description** *colour* must be one of black, white, grey, red, green, blue and yellow.

*rgb\_value* is typically obtained by a call to RGB@. Using the value (-1L) resets the colour of subsequent text to its default value.

**Notes** The text colour for a given control can be changed dynamically (after the window has been created) by calling the subroutine SET\_CONTROL\_TEXT\_COLOUR@.

**Example** `winio@('%tc[red]')`

```
winio@('%tc', RGB@(200,200,0))
```

**See also** %fn, %ts, %bf, %it, %ul, and the Windows API call  
GetSysColor(COLOR\_WINDOWTEXT)

## %th

**Purpose** To provide “tooltip” help.

**Syntax** winio@('%th', *ctrl*)  
integer *ctrl* (input)

**Description** %th specifies that help strings will appear in volatile rectangular boxes adjacent to the associated control when the mouse is over the control. There is a programmed delay before the help string appears, similar to %`bh but without a spike. *ctrl* is set to 1 to make the help tooltip visible and zero to hide it.

**See also** %bh, %he

## %ti

**Purpose** To provide a taskbar icon.

**Syntax** winio@('%ti[*icon\_name*][*tooltip*]', *cb\_func*)  
external *cb\_func*

**Description** Under Windows 95/NT 4.0 (and later operating systems) there is a region on the taskbar to the right where some programs place an icon - usually because they are in a quiescent state. This area is typically used by the modem, speaker volume control, and clock. These icons usually display some text if the mouse is placed over them and respond to mouse clicks by becoming active in some way. Win32 ClearWin+ programs can use this area via the %ti format code.

This format takes two standard strings (the icon resource and help text respectively) and a call-back function. The call-back function will be called for all mouse actions over the taskbar icon. The different mouse actions can be distinguished by using clearwin\_string@('CALLBACK\_REASON').

Typically, the call-back function should respond to clicks (or perhaps double clicks) only. A window containing a %ti format will be invisible apart from the taskbar icon, so there is no point in adding other controls to a %ti window. However, in other ways

a %ti window is analogous to an ordinary window. In particular, if the call-back function returns a zero or negative number, the window will close and the icon will be removed from the taskbar. Closure can also be effected using %lw by setting the associated control variable to a zero value and passing it to `window_update@`.

**Notes** *icon\_name* can be constructed from a known Windows handle (see page 79).

**Example** `winio@('%ti[my_icon][Click here]', cb)`

Within the call-back function:

```
cb=2
IF(clearwin_string@('CALLBACK_REASON').EQ.'MOUSE_LEFT_CLICK')THEN
 .
 .
 cb=0
ENDIF
```

## %tl

**Purpose** To set up tab locations for use with %ta.

**Syntax** `winio@('%Ntl', pos1, pos2, . . . , posN)`  
`integer pos1, pos2, . . . , posN` (input)  
`winio@('%N`tl', pos1, pos2, . . . , posN)`  
`double precision pos1, pos2, . . . , posN` (input)

**Modifiers** Grave accent (`) - use real rather than integer values.

**Description** Tab positions are measured in average character widths. All values are measured from the left-hand edge of the window. Each %tl maintains its effect until the next %tl.

**Example** `winio@('%3`tl', 10.5D0, 20.5D0, 30.5D0)`

**See also** %ta, %ap, %ob

## %ts

**Purpose** To set the text size.

**Syntax** `winio@('%ts', size)`

```
double precision size (input)
```

**Description** *size* is the scaling factor that defaults to 1.0. Values between zero and 1.0 cause the font to be scaled down. Values greater than 1.0 cause it to be scaled up. The default can be restored by using a *size* of 1.0 or by using %sf.

%fn must be used before %ts because the standard font cannot be scaled.

**Example** `winio@('%ts', 1.5D0)`

**See also** %sf, %fn, %tc, %bf, %it, %ul

## %ott

**Purpose** To insert a textual toolbar.

**Syntax** `winio@('%tt[button_text]')`  
`winio@('%~tt[button_text]', grey_ctrl)`  
`integer grey_ctrl` (input)

**Modifiers** Tilde (~) - adds a grey control.

Grave accent (`) - specifies that this is the default button.

Question mark (?) - the help string is placed in square brackets after [button\_text].

Caret (^) - the call-back function is called when the user clicks on the button.

**Description** %tt is similar to %bt but provides a button that is not as deep and whose width is rounded to one of a set of standard sizes.

*button\_text* is a standard character string (can be replaced by @).

%tt is typically used with %ww[no\_border] and/or %bx.

**Example** `winio@('%?^tt[Cancel][Help text...]', cb_func)`

**See also** %bt, %tb, %bx

## %otv

**Purpose** To insert a hierarchical tree.

**Syntax** `winio@('%N.Mtv[options]', tree, nitems, sel)`

---

```

character*(*) tree(nitems) (input/output)
integer nitems (input parameter)
integer sel (input/output)
winio@('%`N.Mtv[options]', tree, nitems, sel, icon_str)
character*(*) icon_str (input)

```

**Modifiers** Grave accent (`) - for programmer supplied icons.

Caret (^) - call-back function that is called when the user selects an item.

**Description** Note that %bv provides an alternative to %tv and has more options.

*N* and *M* are not optional and specify the width and depth of the area in average characters.

*tree* is a character array of *nitems*.

*sel* is the index of the item selected by the user.

*icon\_str* is a character string giving a list of icon resources in the form 'icon1,icon2'.

*options* can take the keyword fixed\_font which causes the control to use a fixed font for the names of the tree nodes. This can be useful in cases where the node names are intended to line up. *options* can also take the keyword keep\_focus to prevent the focus from shifting from the selected item. This is similar to the effect produced by a list box.

If variable\_size is included as one of the *options*, *nitems* is treated as a variable that subsequently can be modified and updated by a call to window\_update@. However, the size cannot be increased above its initial value. Since blank elements in the input array are ignored, the only reason to use this feature is to reduce the processing time (and therefore possible flicker) when using a very large array, most of whose elements are blank.

For further details see page 68.

**Notes** Icon resource names can be constructed from known Windows handles (see page 79).

**Example** winio@('%^`20.16tv',*tree*,15L,*sel*, 'cl\_book,op\_book',*cb*)
 cl\_book ICON book1.ico (resource file)
 op\_book ICON book2.ico

**See also** %bv, %ls, %ms, %pb, %ps, see\_treeview\_selection@

**%otx**

**Purpose** To display an array of text with attributes in a rectangular region.

**Syntax** `winio@('%N.Mtx[options]', text, attrib, ncols, mrows)`  
 character `text(ncols,mrows)` (input)  
 character `attrib(ncols,mrows)` (input)  
 integer `ncols, mrows` (input parameter)

**Modifiers** Caret (^) - the call-back is called when a keyboard character is pressed and the control is in focus. The program can then update the text array.

**Description** The text array `text` is displayed in a box with `ncols` columns and `mrows` rows. Initially only `N` columns and `M` rows are visible and a scroll bar is provided.

Attributes for each character are provided through the array `attrib`, with each character set to a colour index value in the range 0 to 255. Colour indexes are defined in order by (%tc, %ty) format pairs. %tx can be preceded by a pivot (%pv).

`options` can include:

|                               |                                                                                                                      |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>full_mouse_input</code> | The call-back function is called whenever the mouse moves over the control.                                          |
| <code>full_char_input</code>  | The call-back function is called for each key-press and each key release using Microsoft VK_ parameters.             |
| <code>use_tabs</code>         | Pressing the tab key does not move to the next control; instead, a tab key press is available to the %tx call-back.  |
| <code>no_popup_menu</code>    | Used to disable %pm within this control and to make the right mouse click available to the this control if required. |

See page 103 for further details.

**Example** `winio@('%^60.15tx', text, attrib, 80, 20, cb_func)`

**See also** %eb

## %UL

**Purpose** To switch to underlined text.

**Syntax** `winio@('%ul')`

**Modifiers** Grave accent (`) - switches off underlined text.

**Description** %ul is cancelled by %`ul and %sf.

**See also** %bf, %it, %sf

---

## %UW

**Purpose** To allow Windows API code to be interfaced to ClearWin+.

**Syntax** `winio@('%uw[class]', width, height, style, xstyle, hwnd)`  
`integer width, height, style, xstyle (input)`  
`integer hwnd (output)`

**Description** *class* is the name of a registered class with its own Windows procedure.

*width* and *height* define the dimensions of the window in pixels.

*style* is the Windows style (WS\_CHILD is automatically included with *style*).

*xstyle* is the extended Windows style (typically zero).

*hwnd* is returned as the handle of the resultant child window.

**See also** %cw

---

## %VX

**Purpose** To attach a vertical scroll bar to the next control.

**Syntax** `winio@('%vx', page_step, max_val, cur_val)`  
`integer page_step, max_val (input)`  
`integer cur_val (input/output)`

**Modifiers** Caret (^) - the call-back is called when the user clicks on the scroll bar.

**Description** *cur\_val* is the value that corresponds to the position of the scroll box (also called the *thumb*). It returns a value between zero and *max\_val* - 1.

*page\_step* is the amount that *cur\_val* is to be increased/decreased when the user clicks on the scroll bar (other than on the thumb). For a document, *page\_step* will be set to the number of lines that are to be displayed in the window and *max\_val* should be set to *n* - *page\_step*, where *n* is the total number of lines in the document.

With %hx and %vx the three arguments *cur\_val*, *page\_step* and *max\_val* can be passed as variables and changed dynamically with a call to `window_update@`.

A call to `clearwin_info@('INTERMEDIATE_SCROLL')` will determine if the callback was called whilst the thumb was moving. A return value of 1 indicates movement whilst zero indicates that the thumb is stationary. Both of these events occur at the end of a move.

%hx and %vx should not be used with controls like %eb and %cw which have their own scroll bar mechanism.

**Example** `winio@('%^vx', 5L, 20L, pos, cb_func)`

**See also** %hx

## %oWC

**Purpose** To display a character.

**Syntax** `winio@('%wc', ch)`  
character *ch* (input)

**Description** See page 45.

## %wd

**Purpose** To display an integer.

**Syntax** `winio@('%wd', v)`  
integer *v* (input)

**Description** See page 45.

## %we

**Purpose** To display a real value.

**Syntax** `winio@('%we', v)`  
double precision *v* (input)

**Description** See page 45.

---

## %wf

**Purpose** To display a real value.

**Syntax** `winio@('%wf', v)`  
double precision *v* (input)

**Description** See page 45.

---

## %wg

**Purpose** To display a real value.

**Syntax** `winio@('%wg', v)`  
double precision *v* (input)

**Description** See page 45.

---

## %wp

**Purpose** To supply the name of a wallpaper bitmap.

**Syntax** `winio@('%wp[bitmap_name]')`

**Modifiers** Grave accent (`) - ensures that, if necessary, the window is expanded to hold at least one copy of the bitmap.

**Description** *bitmap\_name* is a standard character string (can be replaced by @). The bitmap is used as a back drop to the contents of the window and, if necessary, is repeated

horizontally and vertically to fill the space.

**Example** `winio@(%wp[wallpaper])`  
wallpaper BITMAP file.bmp (resource file)

**See also** %bg

---

## %WS

**Purpose** To write a string.

**Syntax** `winio@(%ws, s)`  
`character*(*) s` (input)

**Description** See page 45.

---

## %WW

**Purpose** To change the style of a window.

**Syntax** `winio@(%ww[options])`

**Description** By default, a format window that uses neither %ww nor %sy has a caption bar but does not have: a) system menu, b) a minimise box and c) a maximise box. Such a window has a double frame that is raised.

%ww is used to change the style a window. If %ww is used without any of its options, the resulting window has a caption bar that includes: a) system menu, b) a minimise box and c) a maximise box. Such a window has a single frame that is not raised.

The list of options given below includes options for changing the *border* of the window. This refers to the space that, by default, is provided by ClearWin+ to surround the text and controls within a window (as in %bd). Do not confuse this *border* with the *frame* of the window that is determined by the Windows API style.

A window that uses %ww can be re-sized if a pivot (%pv) is also included.

*options* can be one or more of:

---

|                                                                |                                                                                                                                                                                                                               |
|----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>casts_shadow</code>                                      | This option produces a window which casts a shadow. It is most effective when used in conjunction with <code>no_frame</code> and <code>no_caption</code> .                                                                    |
| <code>no_caption</code>                                        | Omit the caption from the window.                                                                                                                                                                                             |
| <code>no_maxminbox</code>                                      | Omit the maximise and minimise boxes from the window.                                                                                                                                                                         |
| <code>no_maxbox</code>                                         | Omit the maximise box.                                                                                                                                                                                                        |
| <code>no_minbox</code>                                         | Omit the minimise box.                                                                                                                                                                                                        |
| <code>no_sysmenu</code>                                        | Omit the box at the top left which can be used to produce the system menu.                                                                                                                                                    |
| <code>no_border</code>                                         | Omit the blank border which normally surrounds the window contents, but leave the frame itself. Using this format you can, for example, force a tool bar (textual or bit-mapped) to be placed immediately beneath the menu.   |
| <code>thin_border</code>                                       | Produces a half-size white border.                                                                                                                                                                                            |
| <code>no_frame</code>                                          | Omit the window frame and the blank border which normally surrounds the window contents. This option is particularly useful with certain types of child window.                                                               |
| <code>no_edge</code>                                           | This option is like <code>no_frame</code> except that it leaves the default border i.e. an empty space around the edge of the window.                                                                                         |
| <code>naked</code>                                             | Equivalent to <code>no_frame</code> with <code>no_border</code> and <code>no_caption</code> .                                                                                                                                 |
| <code>volatile</code>                                          | The window closes when the focus is lost.                                                                                                                                                                                     |
| <code>inactive</code>                                          | Disables the window.                                                                                                                                                                                                          |
| <code>invisible</code>                                         | Hides the window.                                                                                                                                                                                                             |
| <code>fixed_size</code>                                        | Prevents the user from resizing the window.                                                                                                                                                                                   |
| <code>topmost</code>                                           | Forces the window to stay on top, even if another application gets control. This is sometimes useful when it is necessary to execute another application without the user losing sight of the window of the main application. |
| <code>maximise</code><br>(or <code>maximize</code> )           | Displays the window as a maximised window.                                                                                                                                                                                    |
| <code>minimise</code><br>(or <code>minimize</code> )           | Displays the window as a icon.                                                                                                                                                                                                |
| <code>super_maximise</code><br>( <code>super_maximize</code> ) | Displays only the client area (requires an accelerator key attached to the Escape key that restores the normal window).                                                                                                       |

**independent** Makes the window independent of other windows in the program that are already open. This can be used to create several windows which the user can move between at will and where there is no implied window hierarchy.

---

**Example** `winio@(%ww[no_border, maximise])`

**See also** `%sy`

---

## **%oWX**

**Purpose** To display a hexadecimal value.

**Syntax** `winio@(‘%wx’, v)`  
integer *v* (input)

**Description** See page 45.



# 22.

## Compiling and linking

### Compiler options and directives for Windows

This section describes the compiler options and directives that are provided for the purpose of creating Windows executable programs. Information about other compiler options and directives will be found in the user guide for your Salford compiler.

Using ClearWin+, a Windows application can be created in one of two ways. One approach is to compile the main program using the /WINDOWS command line option and then run the linker separately (other routines that have not been compiled with /WINDOWS may also be linked in). Further details of this approach are given below. The other method is to use one of the compiler command line options /LINK and /LGO and to insert the WINAPP compiler directive into the program code before the main program.

The WINAPP directive takes the form:

WINAPP resource\_file

The /WINDOWS command line option becomes redundant when this directive is used. When used, WINAPP should be the first compiler directive in the file and must appear before the main program. WINAPP can take either no arguments or one argument. If the resource script file is specified, the SRC resource compiler is invoked (see page 296) to compile the resource script. If it is not specified, the application will simply be marked as a *Windows application* rather than a *Console application*.

Some compilers may require a WINAPP directive to have three arguments rather than one when a resource script is specified. In this case two zero values can be placed before the file name. For example:

```
WINAPP 0,0,resource.rc
```

These represent stack and heap size values that are redundant in Win32 applications.

For Win32 applications, the resource file name must have a different stem (i.e. the part before the extension) from all the other source file names. For example, do not use *prog.rc* with *prog.for*. If you did not obey this rule, both files would be compiled into *prog.obj* and one would overwrite the other.

Resources are automatically appended to the executable or DLL being created. However, by default ClearWin+ only searches for resources in the current executable. Calling the subroutine `USE_RESOURCE_LIBRARY@` causes ClearWin+ to also search in a given DLL. The call takes the form:

```
CALL USE_RESOURCE_LIBRARY@('MYLIB')
```

where *mylib.dll* is the name of the DLL.

Fortran 77 routines that use ClearWin+ and/or the Windows API should include the line

```
INCLUDE <windows.ins>
```

unless the routine only refers to `winio@` or `RGB@` (say) which can be declared as `INTEGER` in the routine. Note the use of “`<`” and “`>`” to denote that the default directory for system include files should be searched. The file called *windows.ins* references three other files as follows:

```
INCLUDE <clearwin.ins>
INCLUDE <win32api.ins>
INCLUDE <win32prm.ins>
```

These files are installed into the INCLUDE sub-directory of your chosen installation directory (the default installation directory is *c:\win32app\salford*). Where appropriate, *clearwin.ins* can be used in place of *windows.ins* in your program. This will reduce the amount of processing at compile time.

Fortran 90/95 routines can alternatively make use of the file *mswin.mod* instead of *windows.ins* via the statement

```
USE MSWIN
```

The source file for *mswin.mod* references three other files as follows:

```
MODULE mswin
 USE msw32prm
 USE mswin32
 USE clrwin
END MODULE mswin
```

Where appropriate, *clrwin* can be used in place of *mswin* in your program. This will reduce the amount of processing at compile time.

It is recommended that you use the **IMPLICIT NONE** statement at the start of any routine that uses the *windows.ins* file. Without **IMPLICIT NONE**, a mispelt Windows parameter (some are very long) will not be detected at compile time. Such bugs are very hard to find in the Windows environment.

**Tip**

Of necessity, the three include files and the equivalent module files contain many thousands of lines of declarations in order to provide you with full access to all the **ClearWin+** functions and the Windows API. Once you are familiar with the concepts described in this guide, you may find it helpful to create your own include or module file that contains only those declarations that are used by your programs. If you use **IMPLICIT NONE**, and remove references to *windows.ins* or *msmod*, the compiler will report all usage of undeclared variables and functions. You can then search the relevant include or module files for the missing names and, using a text editor, build up your own include or module file. Note that **ClearWin+** routines (whose names usually end in an "@" character) will appear in a **C\_EXTERNAL** statement in order to provide the compiler with a reference to a routine in the run time library. If you take the time to make your own include or module file, the benefits of much faster compilation will be immediately obvious, especially if your program contains many sub-programs that use these include or module files.

Unlike a *Console application*, a *Windows application* does not create or inherit a console (i.e. a DOS box), and any default input/output is directed to a *ClearWin* window (which can be embedded inside other windows using %cw). Although a simple **ClearWin+** window can be displayed from a Console application, the rules concerning which Windows API calls will work from within a Console application are ill-defined (**ClearWin+** calls all ultimately resolve into API calls). Therefore, it is recommended that all **ClearWin+** applications should be set up as Windows applications using the **WINAPP** directive.

## SLINK commands

If the **/LINK** and **/LGO** compiler command line options are not used, **SLINK** must be called in order to link programs.

If your application uses a resource script (*resource.rc* say) then it must be separately compiled using the Salford Resource Compiler **SRC**. This generates the object file *resource.obj* which can then be loaded in the same way as other object files.

The following example shows the commands required to compile and link a simple Windows application from a file *myapp.for* using **FTN77**:

```
FTN77 MYAPP /WINDOWS
```

```
SRC RESOURCE
```

```
SLINK
LO MYAPP
LO RESOURCE
FILE
```

#### Note

- a) Because *myapp.obj* and *resource.obj* are both used in the **SLINK** commands it is evident that we cannot use *myapp.rc* as the name of the resource.
- b) If there is more than one file of source code in addition to *myapp.for* then the first to be loaded must contain the main program and be marked as a Windows application either by using the **/WINDOWS** compiler option or by including the **WINAPP** directive.
- c) Not more than one resource object can be loaded.
- d) A **SLINK** command line of the form:

```
subsystem windows,major.minor
specifies the Windows GUI style to use. The default is now
subsystem windows,4
```

The former Windows 3.10 subsystem can also be obtained by setting the environment variable

```
LINK_SUBSYSTEM=3
```

## Using the Salford Resource Compiler

If a program uses a resource script and neither of the compiler options **/LGO** and **/LINK** are used to compile and link the program, then the Salford Resource Compiler **SRC** must be used to compile the resource script. An alternative method can also be used with **FTN95** (see below).

When **SRC** is called it creates a permanent compilation of the resource script. With a few exceptions and a few additions, **SRC** uses the same resource script syntax as the Microsoft Resource Compiler **RC**. However, most **ClearWin+** programmers will find that they only use the following kinds of resources:

```
BITMAP
ICON
CURSOR
GIF
HYPERTEXT
SOUND
```

Once a resource script has be written, it should be stored in a file with the .RC extension. The resource can then be compiled with a call to **SRC** of the form:

```
SRC RESOURCE
```

This command takes the file *resource.rc* as input and produces an object file called *resource.obj*. This means that *resource* must not be used as the stem for both a Fortran filename and resources in the same directory and project. The resulting object file can then be linked with other object files using **SLINK** as follows.

```
SLINK
LO MYAPP
LO RESOURCE
FILE
```

## FTN95

Using **FTN95**, it is possible to append resource directives to the end of a program using the **RESOURCES** directive. Everything after this directive (up to the end of the file) is treated as resources and passed to the **SRC** command. The result is combined with the object code produced by **FTN95**. In order to use the **RESOURCES** directive, the **SRC** command must be available on the PATH. All the resources for a program must be supplied in one place.

Here is a simple example:

```
WINAPP
INTEGER i,winio@
i=winio@('%ca[A beautiful bitmap]&')
i=winio@('%bm[mybitmap]%'2n1%cn%`bt[OK]')
END
RESOURCES
mybitmap BITMAP c:\bitmaps\pic1.bmp
```



# 23.

# ClearWin windows

## Introduction

This chapter describes the attributes of a *ClearWin* window. A summary of functions that can be used with *ClearWin* windows is given on page 315. Details of these functions can be found in Chapter 27.

A *ClearWin* window has the attributes of a *Windows* window but has added functionality. In particular a *ClearWin* window can be used in association with the standard Fortran I/O routines READ, WRITE and PRINT.

Note that a *ClearWin* window is not the same as a window created with `winio@` (which is called a *format* window). It is, however, possible to embed a *ClearWin* window in a *format* window (see page 123).

Graphics can be drawn to a *ClearWin* window by using `get_graphics_dc@` and by calling Windows API drawing functions (details are given in the *ClearWin+ User's Supplement*). However, it is recommended that graphics objects should be drawn to a *format* window created using `%gr`, `%pl` or `%og`.

A *ClearWin* window (that is not embedded in a *format* window) can be created by calling `create_window@` (this function returns a standard *Windows* handle). `destroy_window@` is used to kill the window and `clear_window@` clears any I/O text that appears in the window. However, it does not clear graphics that have been drawn to the associated device context. `update_window@` is called to “invalidate” a *ClearWin* window and hence redraw it. Various other routines can be used to control the font of the text and to get and set the handle for default I/O, whilst `open_to_window@` can be used to attach a Fortran unit number to a particular *ClearWin* window.

As an alternative to using a Fortran READ statement, the programmer can access a *ClearWin* window keyboard buffer via the routines: feed\_wkeyboard@, flush\_wkeyboard@, get\_wkey@, get\_wkey1@, and wkey\_waiting@.

**It is strongly recommended that either the routine set\_all\_max\_lines@ or the routine set\_max\_lines@ be used before creating a *ClearWin* window (including default *ClearWin* windows).** These routines are used to limit the amount of memory that is used by a *ClearWin* window. See below for further details.

## Simple programs

Using ClearWin+, simple programs can be compiled unchanged. As soon as standard output is generated which would have gone to the screen, or the program attempts to read from the keyboard, a *ClearWin* window is set up by the ClearWin+ system. Successive lines of output are written to this window (entitled Output), which scrolls when necessary. The window acts like a terminal, so keyboard input is echoed in the window. The window can be scrolled, moved, or resized by the user. The system retains the information output to the window so that WM\_PAINT messages can be handled without any special programmer action. As an example, here is a complete application program to calculate square roots:

```
WINAPP
DOUBLE PRECISION d
DO WHILE(.TRUE.)
 READ *,d
 PRINT *,SQRT(d)
ENDDO
END
```

Notice that this program does not require an explicit exit path. Each time it pauses for input, the system will be prepared to accept a window close request (Alt-F4). When your program exits, its window will remain open until closed by the user in order to ensure that the user has time to read the output.

Because ClearWin+ must store away every line that you send to a window - to cater for possible repaint requests or user scroll operations - it is important to limit the amount of output to manageable proportions. Store is consumed from the heap and is reclaimed when the *ClearWin* window is closed or cleared. It is recommended that the routine set\_all\_max\_lines@ (or set\_max\_lines@) be used to limit the amount of store that is consumed.

In a similar way, DOS applications that contain calls to the Salford graphics library can also be compiled unchanged. In this case ClearWin+ will automatically create a simple *format* window for the drawing.

## Explicit *ClearWin* window creation

When *ClearWin+* creates a *ClearWin* window for you it is created full screen by default. Also, the window uses the Windows system fixed font (referred to as `SYSTEM_FIXED_FONT` in the Windows documentation). In general, it is better to create a *ClearWin* window explicitly so that you have a *handle* to it and you can exert more control. For example, the following would create a half-size window in the centre of the screen:

```
WINAPP
INCLUDE <windows.ins>
INTEGER win,xsize,ysize,xpos,ypos
xsize=clearwin_info@('screen_width')/2
ysize=clearwin_info@('screen_depth')/2
xpos=xsize/4
ypos=ysize/4
win=create_window@('Moon landing',xpos,ypos,xsize,ysize)
...
```

The first *ClearWin* window that you create will be the parent window. When this window is closed the program will terminate.

Once you have created it, you can inform the system that your window is the default *ClearWin* window, thus:

```
CALL set_default_window@(win)
```

A *ClearWin* window can also be attached to a specified Fortran unit, for example:

```
CALL open_to_window@(7,win)
WRITE(7,'(a)') 'Testing'
```

Fortran units connected to *ClearWin* windows are opened for input and output. When you close such a unit, the corresponding window is closed and all the heap storage required to hold the output is returned to the system.

When your program terminates, the *ClearWin* windows that it has created will remain open for viewing, scrolling etc. until closed by the user (e.g. by pressing Alt-F4).

## Text output to *ClearWin* windows

Textual window input/output uses, by default, a Windows font called `SYSTEM_FIXED_FONT` output in black. You are not limited to the use of this font; Windows can write text in a wide variety of fonts and colours. You simply use

appropriate Windows API routines to create one or more fonts and then attach them to a window.

In order to create a new font, it is usually best to start with the parameters defining the system font and modify them to obtain what you require.

For example:

```
WINAPP
INCLUDE <windows.ins>
INTEGER lfheight,lfwidth,lfescapement,
+ lforientation,lfweight,lfitalic,lfunderline,
+ lfstrikeout,lfcharset,lfoutprecision,
+ lfclipprecision,lfquality,lpitchandfamily
CHARACTER*80 lffacename
INTEGER newfont
CALL get_system_font@(lfheight,lfwidth,lfescapement,
+ lforientation,lfweight,lfitalic,lfunderline,
+ lfstrikeout,lfcharset,lfoutprecision,
+ lfclipprecision,lfquality,lpitchandfamily,lffacename)
c--- Alter the default parameters to double the size
c--- and specify italic.
lfheight=2*lfheight
lfwidth =2*lfwidth
lfitalic=1
c--- CreateFont is a Windows API function...
newfont=CreateFont(lfheight,lfwidth,lfescapement,
+ lforientation,lfweight,lfitalic,lfunderline,
+ lfstrikeout,lfcharset,lfoutprecision,
+ lfclipprecision,lfquality,lpitchandfamily,lffacename)

CALL set_default_font@(newfont)
...
```

The default font used by a *ClearWin* window (SYSTEM\_FIXED\_FONT) can be changed by a call to either set\_default\_proportional\_font@ or to set\_default\_font@. These routines will alter the default font used by ClearWin+ on all *ClearWin* windows created subsequent to the call. The original default can be restored by a call to set\_default\_to\_fixed\_font@.

## Getting text from a *ClearWin* window

It is possible to extract the data placed into a *ClearWin* window by calling:

```
INTEGER FUNCTION GET_CLEARWIN_TEXT@(WIN,BUFFER,COPY)
INTEGER WIN, COPY
CHARACTER*(*) BUFFER
```

WIN is the handle of *ClearWin* window in question. To obtain the handle of the default window use:

```
WIN=GET_DEFAULT_WINDOW@()
```

If COPY is zero, the function returns the length of the required buffer but no data is copied. If COPY is 1, the data is also copied to the buffer.

## Obsolete *ClearWin* window functions

The following functions are considered to have been superseded by the introduction of `wini0@` in *ClearWin+*. They are documented in the *ClearWin+ User's Supplement*.

---

|                              |                                                                           |
|------------------------------|---------------------------------------------------------------------------|
| ADD_FORTRAN_WINDOW_CALLBACK@ | Adds a call-back function for a given <i>ClearWin</i> window and message. |
| ADD_TEXT_DESCRIPTOR@         | Adds a text descriptor to a window.                                       |
| ADD_WINDOW_MENU@             | Sets the menu in a given window.                                          |
| ATTACH_FORTRAN_MENU@         | Loads a named menu from the program resource.                             |
| GET_FILENAME@                | Displays a dialog box from which a filename can be selected.              |
| GET_GRAPHICS_DC@             | Returns a device context to a bitmap.                                     |
| MAKE_DIALOG_BOX@             | Creates and display a given dialog box.                                   |
| SET_WIN_ESCAPE@              | Sets the escape character for a specified <i>ClearWin</i> window.         |

---



# 24.

## Using the Windows API

### Calling Windows API routines from Fortran

As the Windows API is based upon C++, it is easier to use the API from a C++ program. However, it is possible to program the API from Fortran but Fortran data structures do not easily map on to the data structures that are used by the Windows API. One way forward, is to employ mixed language programming, keeping your existing Fortran as far as possible unchanged, and using C++ to provide an interface to the Windows API.

Programmers who are not familiar with C++ will probably prefer to avoid learning a new language. If this is the case, the following points should be kept in mind when calling Windows API functions from Fortran.

All Windows API routines which do not take the address of a routine as an argument (or return one as a result) are available for calling from Fortran. Owing to the fact that the Windows API routines are written in C++, it is necessary to use the Salford Fortran statement **STDCALL** to declare access to Win32 API routines. Examples of STDCALL statements can be seen in the file *win32api.ins*.

Each pointer argument to a Windows API routine has been specified as a **WINREF** or **WINSTRING** (the latter in the case of character arguments that need converting to null-terminated C-strings) and the actual argument should be a variable or array to which the pointer refers. The defintion of STDCALL appears below.

## The Fortran STDCALL statement

The STDCALL declaration allows the Fortran programmer to call Win32 API routines written in C++.

The syntax of the declaration for a STDCALL function is as follows:

```
 STDCALL name ['alias'] [(desc , . . .)] [:restype]
```

where:

*name*

is the name by which it will be called in the Fortran program.

*alias*

is the C++ Windows API name (or the required `__stdcall` function name). Note that this appears in single quotes and is *case-sensitive*.

*desc*

is an argument descriptor, and is either REF, VAL, STRING, INSTRING, or OUTSTRING.

STRING, INSTRING and OUTSTRING may be followed optionally by an integer in parentheses. This integer specifies the maximum length for the corresponding argument in the C routine, in each case where the length of the corresponding Fortran character object cannot be determined (i.e. the actual argument is `CHARACTER(*)`). If the integer is not specified, then a default value of 256 (bytes) is assumed for the maximum length of the string.

*restype*

is the type of the function. If this does not appear then the function does not return a result (equivalent to the C type void). Valid types are INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, LOGICAL and STRING. INTEGER, REAL and LOGICAL may be followed by a length specifier of the form “\**n*”. If the function result is declared to be of type STRING then the function result should be assigned to a variable of type CHARACTER.

Some examples of valid STDCALL declarations are:

```
 STDCALL SUB1, SUB2
 STDCALL SUB2 'GetAttr':INTEGER
 STDCALL CSUB3(REF,STRING(20)):STRING
 STDCALL SUB4 'GetSize' (REF,VAL,VAL,INSTRING,OUTSTRING(100))
```

If no argument specifiers are specified, then default argument linkage is assumed. This is as follows:

Arrays:

by reference (i.e. as a pointer)

**INTEGER** and **REAL** scalars:

by value

**LOGICAL**:

by value, as an integer of the appropriate length, 1 representing .TRUE. and 0 representing .FALSE.

**CHARACTER** objects:

Copied to a compiler defined temporary variable. A trailing null is added to the end of the *significant* length of the string (i.e. there are no trailing spaces). The temporary variable is then passed by reference. In addition, if the actual argument is a scalar or array element, the result in the temporary variable is copied back, and padded to the right with blanks if necessary. This is equivalent to the **STRING** linkage descriptor described below.

Where linkage descriptors are specified, the number of arguments in each call must agree with the number of descriptors specified. The various categories of objects which may correspond to particular argument descriptors are as follows:

Numeric and **LOGICAL** scalars:

Value or reference (VAL or REF)

Arrays, externals, dummy procedures:

Reference (REF)

**CHARACTER** objects:

Reference or string (REF, STRING, INSTRING or OUTSTRING)

The three variants of the **STRING** descriptor are as follows:

**STRING**

The corresponding argument is both input and output, and is copied to a temporary variable on entry to the routine (with a trailing null inserted at the end of the significant length), and if the argument is a scalar or array element, is copied back to the actual argument, blank padded to the right if necessary.

**INSTRING**

The corresponding argument is an input argument with respect to the external routine. The argument is only copied to the temporary variable, and not copied back.

**OUTSTRING**

The argument is returned by the external routine. The temporary variable is set up to be the length of the corresponding scalar or array element plus one, or of specified or default (256) length if the corresponding argument is **CHARACTER<sup>\*</sup>(\*)**, but the value is only copied out. Obviously, this descriptor is only appropriate where the actual argument is a scalar or array element.

When it is required to pass a **NULL** pointer to a string, the value 0 (zero) should be used.

Some Windows API functions allow a particular argument to take two different types in different circumstances. For example, an **LPSTR** in some circumstances and an integer in others. This is outside the scope of the **STDCALL** mechanism. If this feature affects you then you should copy the **STDCALL** statement for the relevant API and modify it to have a different Fortran name and argument list, but keeping the same called name.

A common example is the Windows API function **LoadCursor** which is used to load either a cursor defined in the program resource or a predefined system cursor. This has the following definition:

```
HCURSOR WINAPI LoadCursorA(HANDLE hInstance, LPSTR lpCursorName)
```

When used to load a cursor from the program resource, *hInstance* is the instance handle of the application. *lpCursorName* is a character string containing the name of the cursor in the program resource. This form of the function will have the **STDCALL** declaration :

```
STDCALL LOADCURSOR 'LoadCursorA' (VAL, INSTRING):INTEGER*4
```

When used to load a predefined system cursor, the first argument *hInstance* is set to zero and the second argument *lpCursorName* is an integer containing one of a number of predefined values which specifies the cursor to be loaded. This form of the function will have the **STDCALL** declaration :

```
STDCALL LOADCURSOR 'LoadCursorA' (VAL, VAL):INTEGER*4
```

However, having two differing **STDCALL** statements for the same Fortran function is not allowed. The solution is to change the Fortran name. For example

```
STDCALL LOADCURSOR1 'LoadCursorA' (VAL, INSTRING):INTEGER*4
```

```
STDCALL LOADCURSOR2 'LoadCursorA' (VAL, VAL):INTEGER*4
```

The same situation arises with some other functions that have the form as **LoadCursor**, for example **LoadBitmap** and **LoadIcon**.

A further example is given by the Windows API printer **Escape** function which can take many different forms. One of these has the following form:

```
int WINAPI Escape(hdc, GETTECHNOLOGY, NULL, NULL, lpTechnology)
```

*lpTechnology* is an **LPSTR** (long pointers to strings) so the **STDCALL** declaration for this form of the function for use in a Fortran program would be:

```
STDCALL ESCAPE 'Escape' (VAL, VAL, VAL, INSTRING,
+ OUTSTRING):INTEGER*4
```

A second form of this **Escape** function is:

```
int Escape(hdc, SETCOPYCOUNT, sizeof(int),
 lpNumCopies, lpActualCopies)
```

*lpNumCopies*, and *lpActualCopies* are both **LPINT** (long pointers to integers) so the **STDCALL** declaration for this form of the function would be:

```
STDCALL ESCAPE 'Escape' (VAL, VAL, VAL, REF, REF):INTEGER*4
```

As before, having two differing **STDCALL** statements for the same Fortran function is not allowed and the solution is to change the Fortran name. For example

```
STDCALL ESCAPE1 'Escape' (VAL, VAL, VAL,INSTRING,
+ OUTSTRING):INTEGER*4
STDCALL ESCAPE2 'Escape' (VAL, VAL, VAL,REF,REF):INTEGER*4
```

## The Fortran STOP and PAUSE statements

You may wish to trap the **STOP** or **PAUSE** Fortran statements for your own additional processing. These standard Fortran routines call upon other routines named **STOP@** and **PAUSE@** respectively and the user may provide alternative definitions for these routines to over-ride the system defaults.

In order to do this you should provide your own **STOP@** subroutine with the following specification:

```
SUBROUTINE STOP@(MESSAGE)
CHARACTER*(*) MESSAGE
```

When you have completed your ‘clean up’ operation in **STOP@** you should call on the system routine **EXIT@1** as the last instruction within **STOP@** to allow **ClearWin+** to ‘clean up’ before termination (**EXIT@1** has no arguments and does not return to the calling routine).

```
SUBROUTINE EXIT@1()
```

Similarly, you may also provide your own version of the **PAUSE@** routine with the specification:

```
SUBROUTINE PAUSE@(MESSAGE)
CHARACTER*(*) MESSAGE
```

The default **PAUSE@** message action uses the **MessageBox** function to put up a MODAL dialog box captioned ‘Pause’ with a **STOP** icon, a single **OK** button and the pause message which is a argument in the standard **PAUSE** routine. This default action may be modified if the user writes his own **PAUSE@** routine. No call-on (like the call to **EXIT@1** above) is necessary or possible.

A routine called ABORT@ is available for use when the program needs to terminate because of an error condition.

## Using Windows API structures from Fortran

The difficulty of accessing Windows API structures from Fortran can be alleviated by the use of Fortran statement functions and Fortran EQUIVALENCE statements.

For example, suppose the text metrics for the system font are required in order to determine the size of a main window before it is created. This requires finding the height of a single line of text. The Windows API function **GetTextMetrics** returns the metric properties of the system font in a structure, LPTEXTMETRICS. The specific fields of interest are *tmHeight* and *tmExternalLeading*.

Examining the *windows.h* file, it will be seen that the structure length is 29 bytes. In order to represent this in Fortran, any array may be used, provided its length is at least 29 bytes. In the interests of efficiency it is a good idea to keep 4-byte alignment, so an array 32 bytes long might be used to hold the structure. After the call to **GetTextMetrics**, the relevant field values may be extracted by use of statement functions, using the address of the structure as the argument. For example:

```
integer*1 TEXTMETRICS(32)
integer*2 line_hght,tmHeight,tmExternalLeading
integer*4 atm
.
.
.
tmHeight(i)=core2(i+0)
tmExternalLeading(i)=core2(i+8)
.
.
.
got_tm=GetTextMetrics(hwnd,TEXTMETRICS)
atm=loc(TEXTMETRICS)
line_hght=tmHeight(atm)+tmExternalLeading(atm)
```

CORE2 and LOC are Salford Fortran intrinsic functions.

The use of EQUIVALENCE statements is illustrated on page 111.

# 25.

## Programming tips

This chapter contains a collection of coding ideas for experienced users of ClearWin+. The topics presented here are in no particular order and do not fit comfortably in other parts of the manual.

### Tricks with boxes

The %ob format has a number of uses. Consider what happens if you use %3.4ob[invisible]. In this case the grid structure will be invisible, but the contents of the various cells will still be laid out in a rectangular pattern.

**Boxes can be nested, so some or all of the entries in this lattice can also be boxes.**

Invisible boxes can be used to create groupings of controls. For example, suppose you had a format string such as:

```
"%10.3ls%bt[Case 1]%n1%bt[Case 2]%n1%bt[Case 3]%n1%bt[Case 5]"
```

Because the list box is not very deep, it is quite likely that the first one or two buttons would appear to the right of the list box, while the rest would be separated underneath the box. Obviously you could solve the problem by right justifying all the buttons, but that might not be what was required. A better solution is to surround the buttons with an invisible box.

While you are developing a window it may be a good idea to make all boxes visible, so as to understand the layout. When everything is right you can add the invisible option to the %ob format.

## Debugging with an auxiliary terminal

A common problem encountered while developing Windows applications is that temporary debugging output tends to interfere with the appearance of the application. Although ClearWin+ connects the standard output to a large window for debugging purposes, this can be less than satisfactory because the debug window may obscure part of the program under development. A very effective alternative is to connect the RS232 port to an old fashioned 'dumb' terminal (these can be obtained second hand very cheaply) or to a cheap PC (an AT or XT will do - anything with an RS232 port). If you use a PC you will need to execute a command on the PC to read the RS232 port to the screen (e.g. TYPE COM2). Output can be sent to the RS232 terminal even in the most sensitive parts of a Windows program, such as while the program is responding to the WM\_PAINT message (e.g. while a %eb box is requesting re-colouring).

You can use any COM port, but typically COM1 is connected to the mouse and COM2 is free, so we will assume you are using COM2 in what follows. To write to the COM2 port simply open the file COM2 and write to it - these names are reserved by Windows and do not correspond to ordinary file names. It is important to ensure that the COM port has been set up with the same baud rate as the terminal. Typically you should include a suitable MODE command in the *autoexec.bat* file (e.g. MODE COM2:9600,n,8,1). If in doubt as to whether the link is working, try piping a command from a DOS box to the port, e.g. DIR > COM2.

Great care should be taken to remove all debugging output to RS232 ports before shipping an application. Failure to do this may cause interference with modems or other devices attached to your user's PC.

The following routines are used to send output to an auxiliary terminal:

```
SUBROUTINE SET_RS232_COMM_STREAM@(PORT)
INTEGER PORT

SUBROUTINE RS232_PRINT_HEX@(MESSAGE, VALUE)
CHARACTER*(*) MESSAGE
INTEGER VALUE

SUBROUTINE RS232_PRINT_DEC@(MESSAGE, VALUE)
CHARACTER*(*) MESSAGE
INTEGER VALUE
```

PORT is the COM port number which defaults to the value 2 (i.e. COM2). The string MESSAGE is used to identify the integer VALUE when it is displayed on the terminal.

## Recording mouse and keyboard events

A simple method is available to enable mouse and keyboard actions within a ClearWin+ application to be recorded and replayed. This facility should be thought of as a debug/test facility as it cannot be made totally reliable if the circumstances of the playback differ from those of the recording. In particular, mouse positions will differ if playback occurs at a different screen resolution (see below). The format of recorded files may also vary between versions of ClearWin+.

Recording is enabled by calling `START_CLEARWIN_RECORDER@`. This subroutine takes one CHARACTER variable specifying the name of the file to hold the recording. Recorded operations are written to the file immediately (rather than being buffered) to help in situations in which the application aborts. Recording stops when `STOP_CLEARWIN_RECORDING@` is called, or when the application terminates. The file is created as a text file, and here are some typical contents:

```
<LEFT_MOUSE_DOWN,BT008/1,95,24>
<LEFT_MOUSE_UP,BT008/1,95,24>
<LEFT_MOUSE_DOWN,BT005/2,24,13>
<LEFT_MOUSE_UP,BT005/2,24,13>
<LEFT_MOUSE_DOWN,MAIN/1,4,1>
<LEFT_MOUSE_UP,MAIN/1,4,1>
```

Each event occupies one line of the file and starts with a name for the event. The second field names the control receiving the input. For example, the first two lines refer to button number 8 of the first ClearWin+ window. A name in this form is known as a fully qualified control name. Observe that the first two letters of a control name are derived from the format used to create it. Sometimes a user clicks on the main window rather than a control. In this case the ‘control name’ field has the form shown in the last two lines above. The last two fields for mouse operations are the co-ordinates of the mouse relative to the control in question.

Note that this scheme at least means that some mouse operations, such as button clicks, are unaffected by changes in screen resolution. The reason why recorded input does not make explicit reference to Windows handles is that different handle numbers will be supplied by Windows each time a program is run.

The control naming scheme is effective whether or not recording has been enabled. The following two routines can be used to convert between control names and the corresponding Windows handles:

```
INTEGER*4 HANDLE_FROM_CONTROL_NAME@(NAME)
CHARACTER*(*) NAME

CHARACTER*10 CONTROL_NAME_FROM_HANDLE@(H)
INTEGER*4 H
```

These routines will generate incorrect results if they are given bad arguments.

The subroutine `START_CLEARWIN_PLAYBACK@` can be used to play back a recorded file. This subroutine takes one `CHARACTER` variable specifying the name of the file that holds the recording. When play back is completed, normal mouse and keyboard input is resumed.

It is vital that the playback does not deviate from the recording in any way. For example, if your application displays an extra window if some file is present, then that file must be either present for both the recording and playback, or absent for both.

The problem of mouse actions being sensitive to the screen resolution and font sizes is ameliorated somewhat by the following:

- a) Menus, edit boxes, and text arrays store additional information that stores the data in a resolution-insensitive form. This facility may be extended to cover more controls in future.
- b) Mouse co-ordinates are always stored relative to the control in which they occur. This means that, for example, a mouse click at point (40,50) in a graphics area of some fixed resolution will be recorded correctly.

## Breaking into the debugger

Under Win32, pressing `CTRL-BREAK` or `CTRL-SCROLL-LOCK` will break to the debugger if an application is awaiting input from within a `ClearWin+` window. No action is taken if the program is not running under the debugger. This feature is effective with either the normal debugger (`SDBG`) or the machine level debugger (`SUSWT32`).

# 26.

## Library overview

### ClearWin window functions

This section summarises the functions that can be used with *ClearWin* windows. This type of window is described in Chapter 23.

---

|                     |                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------|
| CLEAR_WINDOW@       | Clears a given <i>ClearWin</i> window to its given background colour.                      |
| CREATE_WINDOW@      | Creates and displays a <i>ClearWin</i> window.                                             |
| DESTROY_WINDOW@     | Closes and destroys a displayed <i>ClearWin</i> window.                                    |
| FEED_WKEYBOARD@     | Inserts a key into the keyboard buffer.                                                    |
| FLUSH_WKEYBOARD@    | Empties the keyboard buffer.                                                               |
| GET_CLEARWIN_TEXT@  | Gets text from a <i>ClearWin</i> window.                                                   |
| GET_DEFAULT_WINDOW@ | Returns the handle of the current default <i>ClearWin</i> window for standard Fortran I/O. |
| GET_WKEY@           | Gets a key from the keyboard buffer.                                                       |
| GET_WKEY1@          | Gets a key from the keyboard buffer but does not wait.                                     |
| OPEN_TO_WINDOW@     | Attaches a <i>ClearWin</i> window to a Fortran unit.                                       |
| SET_ALL_MAX_LINES@  | Sets the maximum number of lines to be stored for all <i>ClearWin</i> windows.             |

|                               |                                                                                     |
|-------------------------------|-------------------------------------------------------------------------------------|
| SET_MAX_LINES@                | Sets the maximum number of lines to be stored for a given <i>ClearWin</i> window.   |
| SET_DEFAULT_FONT@             | Sets the default standard I/O font for <i>ClearWin</i> windows.                     |
| SET_DEFAULT_TO_FIZED_FONT@    | Resets the default standard I/O font for <i>ClearWin</i> windows to “System Fixed”. |
| SET_DEFAULT_PROPORIONAL_FONT@ | Sets the default standard I/O font for <i>ClearWin</i> windows to “System”.         |
| SET_DEFAULT_WINDOW@           | Sets the default <i>ClearWin</i> window for standard I/O.                           |
| UPDATE_WINDOW@                | Redraws a <i>ClearWin</i> window.                                                   |
| WIN_COUA@                     | Outputs $n$ characters from a string to a <i>ClearWin</i> window.                   |
| WIN_GETCHAR@                  | Reads a character from the keyboard.                                                |
| WIN_GETLINE@                  | Reads a line of text from the keyboard.                                             |
| WKEY_WAITING@                 | Checks if the keyboard buffer is not empty.                                         |

## Clipboard functions

|                            |                                                                    |
|----------------------------|--------------------------------------------------------------------|
| CLIPBOARD_TO_SCREEN_BLOCK@ | Copies a bitmap on the clipboard to a DIB.                         |
| COPY_FROM_CLIPBOARD@       | Copies data from the clipboard.                                    |
| COPY_TO_CLIPBOARD@         | Copies data to the clipboard.                                      |
| GRAPHICS_TO_CLIPBOARD@     | Copies a <i>drawing surface</i> to the clipboard.                  |
| METAFILE_TO_CLIPBOARD@     | Sends a metafile to the clipboard.                                 |
| PLAY_CLIPBOARD_METAFILE@   | Plays a clipboard metafile on the current <i>drawing surface</i> . |
| SIZEOF_CLIPBOARD_TEXT@     | Gets the size of text on the clipboard.                            |

## File mapping functions

---

|                          |                        |
|--------------------------|------------------------|
| MAP_FILE_FOR_READING@    | Maps a file to memory. |
| MAP_FILE_FOR_READ_WRITE@ | Maps a file to memory. |
| UNMAP_FILE@              | Closes a file map.     |

---

## Format window functions

The following functions are primarily for use with `WIN10@`.

---

|                          |                                                                                                                                                                                        |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ADD_CURSOR_MONITOR@      | Names a call-back function that monitors which window or control has the cursor.                                                                                                       |
| ADD_FOCUS_MONITOR@       | Names a call-back function that monitors which window or control has the focus.                                                                                                        |
| ADD_KEYBOARD_MONITOR@    | Names a call-back function that monitors the use of the keyboard.                                                                                                                      |
| ADD_MENU_ITEM@           | Creates a dynamic menu using %mn.                                                                                                                                                      |
| DISPLAY_POPUP_MENU@      | Activates a popup menu defined using %pm.                                                                                                                                              |
| GET_MOUSE_INFO@          | Obtains the position of the mouse, the mouse buttons, and the keyboard shift keys at the time when the last owner-draw (%^dw) or graphics region (%^gr) call-back function was called. |
| GET_WINDOW_LOCATION@     | Gets the location and size of a given window.                                                                                                                                          |
| MOVE_WINDOW@             | Moves a window given its handle.                                                                                                                                                       |
| PERMIT_ANOTHER_CALLBACK@ | Allows another call-back function to be called.                                                                                                                                        |
| REMOVE_CURSOR_MONITOR@   | Releases a call-back function that monitors which window or control has the cursor.                                                                                                    |
| REMOVE_FOCUS_MONITOR@    | Releases a call-back function that monitors which window or control has the focus.                                                                                                     |
| REMOVE_MENU_ITEM@        | Removes a dynamically attached %mn menu item.                                                                                                                                          |
| REPLY_TO_TEXT_MESSAGE@   | Replies to a message from an application that uses <code>SEND_TEXT_MESSAGE@</code> .                                                                                                   |

---

|                           |                                                                 |
|---------------------------|-----------------------------------------------------------------|
| RESIZE_WINDOW@            | Resizes a window given its window handle.                       |
| SEE_PROPERTYSHET_PAGE@    | Sets the sheet number for %ps.                                  |
| SEE_TREEVIEW_SELECTION@   | Ensures that the current %tv item is visible.                   |
| SEND_TEXT_MESSAGE@        | Sends a message to an application that uses %rm.                |
| SET_CLEARWIN_STYLE@       | Modifies the default styling controls for <i>format</i> windows |
| SET_CONTROL_TEXT_COLOUR@  | Changes the colour of a text control.                           |
| SET_CONTROL_VISIBILITY@   | Shows or hides a control or window.                             |
| SET_HIGHLIGHTED@          | Selects all of the text in a %rd, %rf or %rs edit box.          |
| SET_OLD_RESIZE_MECHANISM@ | Reverts to the old ClearWin+ resizing strategy.                 |

## General functions

|                           |                                                       |
|---------------------------|-------------------------------------------------------|
| ABORT@                    | Used to terminate a program in the event of an error. |
| HIBYTE@                   | To get the top 8 bits of a 16 bit integer.            |
| HIWORD@                   | To get the top 16 bits of a 32 bit integer.           |
| LOBYTE@                   | To get the bottom 8 bits of a 16 bit integer.         |
| LOWORD@                   | To get the bottom 16 bits of a 32 bit integer.        |
| START_PROCESS@            | To start a new process.                               |
| TEMPORARY_YIELD@          | Yields program control.                               |
| WINDOWS_95_FUNCTIONALITY@ | Tests if Windows 95 is in use.                        |
| YIELD_PROGRAM_CONTROL@    | Yields program control.                               |

## Graphics functions (bitmaps,cursors,icons)

|                    |                                             |
|--------------------|---------------------------------------------|
| ADD_GRAPHICS_ICON@ | Places a movable icon in a graphics window. |
| GET_IM_INFO@       | Obtains information from a graphics file.   |

|                       |                                                   |
|-----------------------|---------------------------------------------------|
| MAKE_BITMAP@          | Embeds a bitmap in program code.                  |
| MAKE_ICON@            | Embeds an icon in program code.                   |
| REMOVE_GRAPHICS_ICON@ | Removes an icon created using ADD_GRAPHICS_ICON@. |
| SET_CURSOR_WAITING@   | Produces a temporary hour-glass cursor.           |

See also Graphics functions (device independent bitmap).

## Graphics functions (device context)

The following routines transmit a handle to a device context and are particularly associated with the %gr and %dw formats described in Chapter 15. GET\_BITMAP\_DC@ returns a handle for use with %dw whilst GET\_CURRENT\_DC@ returns a handle for a device context created by using %gr. These routines allow ClearWin+ to be used with Windows API functions and as such are unlikely to be of interest to a majority of ClearWin+ programmers.

|                          |                                                                      |
|--------------------------|----------------------------------------------------------------------|
| ACTIVATE_BITMAP_PALETTE@ | Ensures that the palette is set correctly for the current window.    |
| ATTACH_BITMAP_PALETTE@   | Attaches a palette to a DC.                                          |
| CHANGE_PEN@              | Changes the pen in a device context acquired through GET_BITMAP_DC@. |
| CLEAR_BITMAP@            | Clears a bitmap device context acquired by GET_BITMAP_DC@.           |
| GET_BITMAP_DC@           | Creates a device context for use with %dw.                           |
| GET_CURRENT_DC@          | Gets the current graphical device context.                           |
| RELEASE_BITMAP_DC@       | Releases a bitmap device context acquired by GET_BITMAP_DC@.         |

## Graphics functions (device independent bitmap)

### Using a 24 bit per pixel array

|                    |                                   |
|--------------------|-----------------------------------|
| DISPLAY_DIB_BLOCK@ | Displays an array on the screen.  |
| GET_DIB_BLOCK@     | Reads a BMP file into an array.   |
| GET_DIB_SIZE@      | Gets information from a BMP file. |

---

|                |                               |
|----------------|-------------------------------|
| PUT_DIB_BLOCK@ | Saves an array to a BMP file. |
|----------------|-------------------------------|

---

### Using a handle

|                            |                                            |
|----------------------------|--------------------------------------------|
| CLIPBOARD_TO_SCREEN_BLOCK@ | Copies a bitmap on the clipboard to a DIB. |
| DIB_PAINT@                 | Displays a DIB.                            |
| EXPORT_BMP@                | Exports a DIB to a BMP file.               |
| EXPORT_PCX@                | Exports a DIB to a PCX file.               |
| GET_SCREEN_DIB@            | Copies part of a graphics area.            |
| IMPORT_BMP@                | Reads in a BMP file.                       |
| IMPORT_PCX@                | Reads in a .PCX file.                      |
| PRINT_DIB@                 | Prints a device independent bitmap.        |
| RELEASE_SCREEN_DIB@        | Releases memory for a DIB.                 |

---

See page 145 for further details.

## Graphics functions (font, text)

|                  |                                                                                           |
|------------------|-------------------------------------------------------------------------------------------|
| BOLD_FONT@       | Makes text on a <i>drawing surface</i> bold.                                              |
| CHOOSE_FONT@     | Displays a standard Windows dialog box so that the user can select a font and its colour. |
| DRAW_CHARACTERS@ | Draws text on a <i>drawing surface</i> .                                                  |
| FONT_METRICS@    | Gets the system metrics for the default font and the current <i>drawing surface</i> .     |
| GET_FONT_ID@     | Gets the font identifier for the current <i>drawing surface</i> .                         |
| GET_FONT_NAME@   | Returns the name of a loaded font.                                                        |
| GET_SYSTEM_FONT@ | Gets the attributes of the system font.                                                   |
| GET_TEXT_SIZE@   | Gets the dimensions of a given character string.                                          |
| ITALIC_FONT@     | Makes text on a <i>drawing surface</i> italic.                                            |
| ROTATE_FONT@     | Rotates the font being used on a <i>drawing surface</i> .                                 |
| SCALE_FONT@      | Scales the font being used on a <i>drawing surface</i> .                                  |

|                     |                                                             |
|---------------------|-------------------------------------------------------------|
| SELECT_FONT@        | Selects a font for a <i>drawing surface</i> .               |
| SELECT_FONT_ID@     | Selects a font for a <i>drawing surface</i> .               |
| SIZE_IN_PIXELS@     | Sets the font size for a <i>drawing surface</i> .           |
| SIZE_IN_POINTS@     | Sets the font size for a <i>drawing surface</i> .           |
| UNDERLINE_FONT@     | Underlines text on a <i>drawing surface</i> .               |
| USE_WINDOWS95_FONT@ | Sets the default font to Windows 95 font (Windows 95 only). |

## Graphics functions (lines,fill)

The following routines draw to the current *drawing surface*. The colour value that is used will depend on the selected colour mode for the surface (either RGB mode or VGA mode). See Chapter 15 for details.

|                             |                                                  |
|-----------------------------|--------------------------------------------------|
| CLEAR_SCREEN@               | Clears a <i>drawing surface</i> .                |
| DRAW_BEZIER@                | Draws a Bezier spline.                           |
| DRAW_ELLIPSE@               | Draws an ellipse.                                |
| DRAW_FILLED_ELLIPSE@        | Fills an ellipse.                                |
| DRAW_FILLED_POLYGON@        | Fills a polygon.                                 |
| DRAW_FILLED_RECTANGLE@      | Fills a rectangle.                               |
| DRAW_LINE_BETWEEN@          | Draws a straight graphics line.                  |
| DRAW_POINT@                 | Sets the colour of a pixel..                     |
| DRAW_POLYLINE@              | Draws a number of connected straight lines.      |
| DRAW_RECTANGLE@             | Draws a rectangle.                               |
| FILL_SURFACE@               | Fills a area with a given colour.                |
| FILL_TO_BORDER@             | Fills a area with a given colour.                |
| GET_GRAPHICS_SELECTED_AREA@ | Gets the current screen ‘rubber-band’ rectangle. |
| GET_POINT@                  | Gets the colour of a pixel..                     |
| SET_GRAPHICS_SELECTION@     | Selects a screen ‘rubber-band’ mode.             |
| SET_LINE_STYLE@             | Sets the line style.                             |

---

|                 |                      |
|-----------------|----------------------|
| SET_LINE_WIDTH@ | Sets the line width. |
|-----------------|----------------------|

---

## Graphics functions (metafiles)

A metafile is a device independent representation for graphics images. You need to open a metafile if you use either DO\_COPIES@ or PRINT\_GRAPHICS\_PAGE@ to print a graphics image (an example appears on page 195).

---

|                          |                                                                         |
|--------------------------|-------------------------------------------------------------------------|
| CLOSE_METAFILE@          | Closes a previously opened metafile.                                    |
| DO_COPIES@               | Produces multiple printer copies using a metafile.                      |
| METAFILE_TO_CLIPBOARD@   | Sends a metafile to the clipboard.                                      |
| OPEN_METAFILE@           | Records graphics sequences so that they can be replayed to the printer. |
| PLAY_CLIPBOARD_METAFILE@ | Plays a clipboard metafile on the current <i>drawing surface</i> .      |
| PRINT_GRAPHICS_PAGE@     | Prints a <i>drawing surface</i> using a metafile.                       |

---

## Graphics functions (palette)

---

|                            |                                                                                     |
|----------------------------|-------------------------------------------------------------------------------------|
| GET_MATCHED_COLOUR@        | Gets the closest match on the current <i>drawing surface</i> .                      |
| GET_NEAREST_SCREEN_COLOUR@ | Gets the closest screen colour match from the system palette.                       |
| GET_RGB_VALUE@             | Gets the RGB value of the colour of a pixel on the current <i>drawing surface</i> . |
| HIGH_COLOUR_MODE@          | Tests if the screen is a high colour device.                                        |
| SET_RGB_COLOURS_DEFAULT@   | Changes the global default colour mode.                                             |
| USE_APPROXIMATE_COLOURS@   | Switches off colour matching.                                                       |
| USE_RGB_COLOURS@           | Switches between VGA colour mode and RGB colour mode on a <i>drawing surface</i> .  |

---

The following functions access an internal ClearWin+ palette - a logical palette that is attached to the device context of each *drawing surface*. Palettes are redundant if the

graphics device is a *high colour* device (i.e. has more than 8 bits per pixel). These functions are documented in the *ClearWin+ User's Supplement*.

---

|                        |                              |
|------------------------|------------------------------|
| GET_COLOURS@           | Accesses palette data.       |
| GET_PCL_PALETTE@       | Accesses palette data.       |
| LOAD_PCL_COLOURS@      | Loads standard PCL colours.  |
| LOAD_STANDARD_COLOURS@ | Loads standard VGA colours.  |
| PERFORM_COLOURS@       | Implements a palette change. |
| SET_COLOURS@           | Loads given palette data.    |

---

## Graphics functions (printer,files)

---

|                          |                                                             |
|--------------------------|-------------------------------------------------------------|
| CLOSE_PRINTER@           | To output to a graphics printer or plotter.                 |
| CLOSE_PRINTER_ONLY@      | Closes a graphics printer or plotter without output.        |
| GET_PRINTER_ORIENTATION@ | Gets the printer orientation as portrait or landscape.      |
| NEW_PAGE@                | Provides a new page on the current <i>drawing surface</i> . |
| OPEN_GL_PRINTER@         | Begins OpenGL output to a graphics printer.                 |
| OPEN_GL_PRINTER1@        | Begins OpenGL output to a graphics printer.                 |
| OPEN_PRINTER@            | Begins output to a graphics printer or plotter.             |
| OPEN_PRINTER_TO_FILE@    | Begins output to a file.                                    |
| OPEN_PRINTER1@           | Begins output to a graphics printer or plotter.             |
| OPEN_PRINTER1_TO_FILE@   | Begins output to a file.                                    |
| PRINT_OPENGL_IMAGE@      | Prints an OpenGL image.                                     |
| SELECT_PRINTER@          | Configures and selects the active printer.                  |
| SET_PRINTER_ORIENTATION@ | Sets the printer orientation to portrait or landscape.      |

---

See also Graphics functions (metafiles).

## Graphics functions (screen,regions)

### Regions

|                         |                                                   |
|-------------------------|---------------------------------------------------|
| COPY_GRAPHICS_REGION@   | Copies one <i>drawing surface</i> to another.     |
| CREATE_GRAPHICS_REGION@ | Creates a <i>drawing surface</i> .                |
| DELETE_GRAPHICS_REGION@ | Removes a <i>drawing surface</i> .                |
| GRAPHICS_TO_CLIPBOARD@  | Copies a <i>drawing surface</i> to the clipboard. |
| SCROLL_GRAPHICS@        | Scrolls a <i>drawing surface</i> .                |
| SELECT_GRAPHICS_OBJECT@ | Selects a <i>drawing surface</i> .                |
| WRITE_GRAPHICS_TO_BMP@  | Writes a <i>drawing surface</i> to a file.        |
| WRITE_GRAPHICS_TO_PCX@  | Writes a <i>drawing surface</i> to a file.        |

These are used together with the DIB functions listed on page 319. An alternative set of routines uses the concept of a virtual screen (a legacy from DOS programming). The alternative set is described in the *ClearWin+ User's Supplement*.

### Other routines

|                           |                                                                       |
|---------------------------|-----------------------------------------------------------------------|
| GET_GRAPHICAL_RESOLUTION@ | Gets the width and height of the current <i>drawing surface</i> .     |
| GRAPHICS_WRITE_MODE@      | Selects replace/XOR mode before writing to a <i>drawing surface</i> . |
| PERFORM_GRAPHICS_UPDATE@  | Refreshes a <i>drawing surface</i> 's display.                        |

## Hypertext functions

The following routines can be used in association with the %ht format described in Chapter 19.

|                         |                                                                 |
|-------------------------|-----------------------------------------------------------------|
| ADD_HYPERTEXT@          | Adds text to the hypertext system.                              |
| ADD_HYPERTEXT_RESOURCE@ | Opens a hypertext resource that is included in a resource file. |
| CHANGE_HYPERTEXT@       | Changes the topic in a hypertext control.                       |

## Information functions

The following routines can be used in association with the `WINIO@` function.

|                                   |                                                                       |
|-----------------------------------|-----------------------------------------------------------------------|
| <code>CLEARWIN_FLOAT@</code>      | Interrogates the state of the ClearWin+ environment.                  |
| <code>CLEARWIN_INFO@</code>       | Interrogates the state of the ClearWin+ environment.                  |
| <code>CLEARWIN_STRING@</code>     | Obtains string information from ClearWin+.                            |
| <code>CLEARWIN_VERSION@</code>    | Gets the current ClearWin+ version information.                       |
| <code>SET_CLEARWIN_FLOAT@</code>  | Sets or change a floating point value associated with a named string. |
| <code>SET_CLEARWIN_INFO@</code>   | Sets or change an integer value associated with a named string.       |
| <code>SET_CLEARWIN_STRING@</code> | Sets or change a string value associated with a named string.         |

## Mouse functions

|                                         |                                                          |
|-----------------------------------------|----------------------------------------------------------|
| <code>GET_MOUSE_INFO@</code>            | Gets the mouse position, mouse state and keyboard state. |
| <code>SET_MOUSE_CURSOR_POSITION@</code> | Sets the mouse position relative to a given window.      |

## Movie functions

|                             |                                      |
|-----------------------------|--------------------------------------|
| <code>SHOW_MOVIE@</code>    | To show a video.                     |
| <code>MOVIE_PLAYING@</code> | To test if a video is still playing. |

## Standard dialog functions

|                        |                                                                                           |
|------------------------|-------------------------------------------------------------------------------------------|
| CHOOSE_COLOUR@         | Displays a standard Windows dialog box so that the user can select a colour.              |
| CHOOSE_FONT@           | Displays a standard Windows dialog box so that the user can select a font and its colour. |
| DEFINE_FILE_EXTENSION@ | Allows application registry under Windows 95.                                             |
| GET_FILTERED_FILE@     | Accesses the standard 'open' dialog box.                                                  |
| SET_OPEN_DIALOG_PATH@  | Sets the initial directory before displaying the standard file open dialog box.           |

See also standard call-back functions (Chapter 20).

## Sound functions

These functions require a sound card such as Sound Blaster. Note that simple WAV resources can be played using the 'SOUND' standard call-back function.

|                      |                                                                          |
|----------------------|--------------------------------------------------------------------------|
| CLOSE_CD_TRAY@       | Closes an open CD-drive tray.                                            |
| CLOSE_WAV_FILE@      | Closes a WAV file opened by OPEN_WAV_FILE_READ@ or OPEN_WAV_FILE_WRITE@. |
| GET_TRACK_LENGTH@    | Returns the length of a CD track.                                        |
| OPEN_CD_TRAY@        | Opens a CD-drive tray.                                                   |
| OPEN_WAV_FILE_READ@  | Opens a WAV file for reading in sections.                                |
| OPEN_WAV_FILE_WRITE@ | Opens a WAV file for reading in sections.                                |
| PLAY_AUDIO_CD@       | Plays the audio CD from the current position.                            |
| PLAY_SOUND@          | Sends a sample to the audio output device.                               |
| PLAY_SOUND_RESOURCE@ | Sends a resource SOUND to the current audio device.                      |
| RECORD_SOUND@        | Records sound from the sound input device.                               |
| SET_CD_POSITION@     | Sets the position of the CD read head to TRACK + MILLISECONDS (offset).  |

|                        |                                          |
|------------------------|------------------------------------------|
| SET_SOUND_SAMPLE_RATE@ | Sets the sampling speed.                 |
| SOUND_PLAYING@         | Tests for ongoing sound output.          |
| SOUND_RECORDING@       | Tests if a recording is being made.      |
| SOUND_SAMPLE_RATE@     | Returns the current sound sampling rate. |
| STOP_AUDIO_CD@         | Stops audio playback from the CD.        |
| WAV_FILE_READ@         | Reads a WAV file as a samples wave-form. |
| WAV_FILE_WRITE@        | Stores any recorded data.                |
| WRITE_WAV_FILE@        | Stores any recorded data.                |

## Useful API functions

The following API functions are simple to use and are occasionally useful in ClearWin+ programs. Details can be found in Microsoft SDK documentation.

|                  |                                                                                                                                                   |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| BringWindowToTop | Brings the specified window to the top of the “Z order”.                                                                                          |
| ClipCursor       | Confines the mouse cursor to a given rectangle. Use CORE4(0) to represent the NULL pointer. Use an integer array to represent the RECT structure. |
| CopyFile         | Copies one file to another.                                                                                                                       |
| CreateProcess    | For simplicity, use START_PROCESS@                                                                                                                |
| EnableWindow     | Enables/disables a window.                                                                                                                        |
| FindWindow       | Finds the handle of a window from its class name and/or its title.                                                                                |
| GetDeskTopWindow | Gets the window handle of the screen.                                                                                                             |
| GetWindowLong    | Gets window parameters such as style and extended style.                                                                                          |
| GetWindowText    | Gets the title of a window.                                                                                                                       |
| IsIconic         | Tests if a window is minimised.                                                                                                                   |
| IsWindowVisible  | Tests if a window is visible.                                                                                                                     |
| IsZoomed         | Tests if a window is maximised.                                                                                                                   |
| MoveWindow       | Changes a window’s position and dimensions.*                                                                                                      |
| SetWindowLong    | Sets window parameters such as style and extended style.                                                                                          |

|               |                                                       |
|---------------|-------------------------------------------------------|
| SetWindowPos  | Changes a window's position, dimensions and state.*   |
| SetWindowText | Sets the title of a window.                           |
| ShellExecute  | Opens or prints a file (e.g. accesses a website).     |
| ShowWindow    | Changes a windows state: visible, minimised, etc.     |
| WinExec       | Starts another application (see also START_PROCESS@). |
| WinHelp       | Displays a Windows help file.                         |

\* See also GET\_WINDOW\_LOCATION@ and its associated notes.

# 27.

## Library reference

---

### ABORT@

**Purpose** To terminate a program in the event of an error.

**Syntax** SUBROUTINE ABORT@( )

**Description** Normally when a program terminates - e.g. by executing a STOP statement, or by encountering the END statement in the main program - any windows that are open are left open until the user closes them and call-back functions continue to be called to service the open windows.

Sometimes, if for example a serious error is detected, it may be necessary to terminate the program immediately in which case the subroutine ABORT@ should be called.

---

### ACTIVATE\_BITMAP\_PALETTE@

**Purpose** To ensure that the palette is set correctly for the current window.

**Syntax** SUBROUTINE ACTIVATE\_BITMAP\_PALETTE@( DC )

INTEGER DC

**Description** This subroutine restores the current palette that the device context is using. This is usually handled by Clearwin+.

**See also** ATTACH\_BITMAP\_PALETTE@, CLEAR\_BITMAP@, CREATE\_BITMAP@,  
GET\_BITMAP\_DC@

## ADD\_ACCELERATOR@

**Purpose** To associate an accelerator key with a given window.

**Syntax**

```
SUBROUTINE ADD_ACCELERATOR@(W, KEY_NAME, F)
 INTEGER W
 CHARACTER*(*) KEY_NAME
 EXTERNAL F
```

**Description** This associates the callback function F with the named key (see the definition of %ac on page 88 for details of the names of keys). W is the handle of the window which you can obtain using %hw when the window is created. If a previous accelerator definition for this key in this window exists then it is removed.

**Notes** You could use this routine if you were writing a configurable editor and you wished to change the accelerator definitions of the surrounding window dynamically.

The call-back function cannot be a string representing a standard call-back.

**See also** %ac, REMOVE\_ACCELERATOR@

---

## ADD\_CURSOR\_MONITOR@

**Purpose** To name a call-back function that monitors which window or control has the cursor.

**Syntax**

```
SUBROUTINE ADD_CURSOR_MONITOR@(CB_FUNC)
 EXTERNAL CB_FUNC
```

**Description** A general mechanism is available to enable programmers to monitor the window/control that has the cursor (or the focus, see ADD\_FOCUS\_MONITOR@).

CB\_FUNC is the name of a routine that is either a function returning an integer, or a SUBROUTINE. The return value (if any) is not used. Cursor call-back functions can use the CLEARWIN\_INFO@ parameter 'CURSOR\_WINDOW' to determine the window handle of the control or window which has the cursor.

- Notes**
- Window handles can be obtained using %hw or %lc.
  - Although window handles are unique at any given time, handles can (and often do) get re-used after a window has been destroyed.

**See also** REMOVE\_CURSOR\_MONITOR@, ADD\_FOCUS\_MONITOR@

## ADD\_FOCUS\_MONITOR@

- Purpose** To name a call-back function that monitors which control has the focus.
- Syntax**
- ```
SUBROUTINE ADD_FOCUS_MONITOR@( CB_FUNC )
EXTERNAL CB_FUNC
```
- Description** See ADD_CURSOR_MONITOR@. Focus call-back functions use the CLEARWIN_INFO@ parameter 'FOCUSSED_WINDOW'.
- Notes** The CLEARWIN_INFO@ parameter used here is not the same as 'FOCUS_WINDOW' which returns the handle of the WINIO@ window which has focus rather than the control within the window that has the focus.
- See also** REMOVE_FOCUS_MONITOR@

ADD_GRAPHICS_ICON@

- Purpose** To place a movable icon or bitmap in a %gr graphics region.
- Syntax**
- ```
INTEGER FUNCTION ADD_GRAPHICS_ICON@(NAME, X, Y,
+ WIDTH, DEPTH)
CHARACTER*(*) NAME
INTEGER X, Y, WIDTH, DEPTH
```
- Description** A %gr graphics region may have an icon resource ‘attached’ to it. The icon can be moved freely around the graphics window with the mouse. Control of the icon is maintained by the call-back function attached to the %gr window with the caret modifier (^).
- The NAME argument is that used for the resource that is included in your program. The X and Y values, initially hold the location the image is draw to. They also hold the new position when the image is moved. The CLEARWIN\_INFO@ parameter 'DRAGGED\_ICON' is set to the value of the handle of the moved icon. The handle is initially obtained as the returned from the function. The call-back function may modify the X and Y values if required. The purpose of this may be to ‘lock’ the image onto the horizontal or vertical plane by continuously setting either the X or Y variables to a constant value.
- When the icon is dropped CLEARWIN\_INFO@ parameter 'DROPPED\_ICON' is set to the handle of the icon. A graphics icon can be removed with a call to REMOVE\_GRAPHICS\_ICON@.

The WIDTH and HEIGHT parameters should be set at zero except when the icon image occupies less than the possible 32x32 icon area e.g. 16x16. A small icon can be constructed by starting at the top-left corner and filling the remaining unused area with the ‘transparent’ colour. If you do not specify the correct size of the icon then the unused area will be detected by the mouse.

**Return value** Returns a non-zero value for success or zero for failure.

**See also** REMOVE\_GRAPHICS\_ICON@

---

## ADD\_HYPERTEXT@

**Purpose** To add text to the hypertext system.

**Syntax** SUBROUTINE ADD\_HYPERTEXT@( BUFFER, SIZE, HTEXTNAME )  
CHARACTER\*(\*) BUFFER, HTEXTNAME  
INTEGER SIZE

**Description** ADD\_HYPERTEXT@ adds, to the hypertext resource base, a hypertext document containing one or more hypertext topics. Note that this routine does not take a copy of the hypertext data so you must ensure that the memory is not changed or discarded before any hypertext processing is complete. The format code %ht is used to display the hypertext resource. Resources may be included in the resource section of your program/resource script.

**See also** ADD\_HYPERTEXT\_RESOURCE@

---

## ADD\_HYPERTEXT\_RESOURCE@

**Purpose** To open a hypertext resource that is included in a resource file.

**Syntax** SUBROUTINE ADD\_HYPERTEXT\_RESOURCE@( NAME )  
CHARACTER\*(\*) NAME

**Description** The resource file must be added to combine the hypertext document with the program using it. The resource filename should be placed on the WINAPP line after the stack and heap size declarations.

See %ht format and the browse example code included with the ClearWin+ release.

**See also** ADD\_HYPERTEXT@

**Example**    WINAPP 50000, 50000, 'MYPROG.RC'

Where *myprog.rc* is a list of resources.

For example:

```
HYPsomeinfo HYPERTEXT "info1.htm"
```

## ADD\_KEYBOARD\_MONITOR@

**Purpose** To name a call-back function that monitors the use of the keyboard.

**Syntax**    SUBROUTINE ADD\_KEYBOARD\_MONITOR@( CB\_FUNC )  
              EXTERNAL CB\_FUNC

**Description** See ADD\_CURSOR\_MONITOR@. Keyboard call-back functions use the CLEARWIN\_INFO@ parameter 'KEYBOARD\_KEY'.

**Notes** This routine replaces GET\_WKEY@ and GET\_WKEY1@ and should be used in preference to these routines. Calls to these routines will fail if ADD\_KEYBOARD\_MONITOR@ has been called.

**See also** REMOVE\_KEYBOARD\_MONITOR@

## ADD\_MENU\_ITEM@

**Purpose** To create a dynamic menu using %mn.

**Syntax**    SUBROUTINE ADD\_MENU\_ITEM@( HANDLE, NAME, ENABLE,  
              + CHECK, CALLBACK\_FUNCTION)  
              INTEGER HANDLE, ENABLE, CHECK, CALLBACK\_FUNCTION  
              CHARACTER(\*) NAME

**Description** By using %mn, menu items can be added at run time. This is of greatest use when you wish to show a file history or a list of open windows. Items are added one at a time to the end of the pop down menu that contains the handle.

In %mn (see page 81) an asterisk (\*) must be added where a text description would previously have been placed and a integer argument, instead of a call-back, must be provided to hold the HANDLE.

For example:

```
i=winio@('%mn[&Window[*]]&', HANDLE)
```

HANDLE can then be passed to ADD\_MENU\_ITEM@.

NAME is the new string to put into the menu i.e. the new menu item. ENABLE controls the enabling of the item (1 for enabled, zero for disabled, i.e. greyed) and the CHECK will add or remove a tick character (1 for checked, zero for unchecked). A call-back function must be provided so that ClearWin+ can act upon a menu selection. A separator is obtained if NAME is input as CHAR(0).

**Example** See page 83.

**See also** REMOVE\_MENU\_ITEM@, %mn

## ATTACH\_BITMAP\_PALETTE@

**Purpose** To attach a palette to a DC.

**Syntax**

```
INTEGER ATTACH_BITMAP_PALETTE@(HDC, P)
INTEGER HDC, P
```

**Description** To attach a colour palette to a device context. This should not normally be needed as SET\_COLOURS@ provides sufficient flexibility.

**Return value** On success 1 will be returned. -1 indicates that the screen type cannot provide the palette requested due to too few palette entries available. A zero indicates that the display is in full colour mode and therefore does not use a palette.

**See also** ACTIVATE\_BITMAP\_PALETTE@

## BOLD\_FONT@

**Purpose** To change the current *drawing surface* text to/from bold.

**Syntax**

```
SUBROUTINE BOLD_FONT@(ACTIVE)
INTEGER ACTIVE
```

**Description** When using the function DRAW\_CHARACTERS@ on a *drawing surface* with any font selected (other than a Hershey font), a call to this routine with a value of one will make any further text output bold. A further call with a value of zero will reverse the effect.

**See also** ROTATE\_FONT@, SCALE\_FONT@, SELECT\_FONT@, ITALIC\_FONT@,  
UNDERLINE\_FONT@

---

## CHANGE\_HYPERTEXT@

**Purpose** To change the topic in a hypertext control.

**Syntax** SUBROUTINE CHANGE\_HYPERTEXT@( HWND, TOPICNAME )  
INTEGER HWND  
CHARACTER(\*) TOPICNAME

**Description** TOPICNAME is the name of the topic to select (as in <DOC name = "doc\_name">) and HWND is the handle of the main window returned by %hw.

---

---

## CHANGE\_PEN@

**Purpose** To change the pen in a device context acquired through GET\_BITMAP\_DC@.

**Syntax** INTEGER FUNCTION CHANGE\_PEN@(HDC,PEN\_STYLE,WIDTH,COLOUR)  
INTEGER HDC,PEN\_STYLE,WIDTH,COLOUR

**Description** The pen associated with a device context acquired through GET\_BITMAP\_DC@ can be changed using the Windows API function **SelectObject**. However, when using the CHANGE\_PEN@ function, ClearWin+ keeps track of which pens have been created, reuses them where possible and finally deletes them automatically. The style, width, and colour arguments are as supplied to the Windows API function **CreatePen**.

**Return value** Always returns 1.

---

---

## CHOOSE\_COLOUR@

**Purpose** To display a standard Windows dialog box so that the user can select a colour.

**Syntax** INTEGER CHOOSE\_COLOUR@( )

**Return value** This function returns the RGB value of the colour selected by the user or -1 if the operation is cancelled. The user can use the dialog to define custom colours that are available if the dialog is opened again before the program terminates (default values are restored when the program restarts).

**See also** CHOOSE\_FONT@

---

## CHOOSE\_FONT@

**Purpose** To display a standard Windows dialog box so that the user can select a font, its attributes and its colour.

**Syntax** SUBROUTINE CHOOSE\_FONT@( RBGCOLOUR )  
INTEGER RBGCOLOUR

**Description** This function displays a dialog box that allows the user to select a font, its attributes (bold, italic, underline, strikeout) and its colour. This becomes the font for the current *drawing surface*. RBGCOLOUR is used for both input and output. On input it is the colour that is initially selected in the dialog box. On output it is the RGB value selected by the user.

If this font is to be restored after a change of font then you should call GET\_FONT\_ID@ before the change and then SELECT\_FONT\_ID@ to restore the font.

**See also** GET\_FONT\_ID@, SELECT\_FONT\_ID@, CHOOSE\_COLOUR@

---

## CLEAR\_BITMAP@

**Purpose** To clear a bitmap device context acquired by GET\_BITMAP\_DC@.

**Syntax** SUBROUTINE CLEAR\_BITMAP@( HDC )  
INTEGER HDC

**Description** The bitmap associated with the given device context is cleared to the current Windows background

**See also** GET\_BITMAP\_DC@, RELEASE\_BITMAP\_DC@, %dw

---

## CLEAR\_SCREEN@

**Purpose** To clear the *drawing surface*.

**Syntax** SUBROUTINE CLEAR\_SCREEN@

**Description** CLEAR\_SCREEN@ clears the current *drawing surface* to the default background.

**See also** DRAW\_FILLED\_RECTANGLE@.

---

## CLEAR\_WINDOW@

**Purpose** To clear a given *ClearWin* window to its given background colour.

**Syntax** SUBROUTINE CLEAR\_WINDOW@(W)  
INTEGER W

**Description** All text written to the window will be cleared and all call-back requests deleted. This is effectively ‘clear screen’ and leaves the window in a state as if it had just been created. W is the window handle returned by CREATE\_WINDOW@.

---

## CLEARWIN\_FLOAT@

**Purpose** To access ClearWin+ parameters.

**Syntax** DOUBLE PRECISION FUNCTION CLEARWIN\_FLOAT@( PARAM )  
CHARACTER\*(\*) PARAM

**Description** Parameters are specified as case insensitive character strings. They may be system or user-defined values.

**Return value** Returns the value of the specified floating point parameter.

**See also** SET\_CLEARWIN\_FLOAT@

---

## CLEARWIN\_INFO@

**Purpose** To access ClearWin+ parameters.

**Syntax** INTEGER FUNCTION CLEARWIN\_INFO@( PARAM )  
CHARACTER\*(\*) PARAM

**Description** Parameters are specified as case insensitive character strings. They may be system or user-defined values.

The following parameters are defined:

ACTION\_X  
Provides the X value of the control that has generated the call-back.

**ACTION\_Y**

Provides the Y value of the control that has generated the call-back.

**ACTIVE\_EDIT\_BOX**

Returns the address of the EDIT\_INFO array of the currently active %eb box.

**CALL\_BACK\_WINDOW**

This option provides the call-back routine with the handle of the window generating the call-back.

**CURSOR\_WINDOW**

Used with ADD\_CURSOR\_MONITOR@. Gives the handle of the control which currently has the cursor.

**DRAGGED\_ICON**

Is set to the value of the handle for the dragged icon, indicating that a icon is currently being dragged (see ADD\_GRAPHICS\_ICON@).

**DROPPED\_COUNT**

Used with the %dr call-back function to return the number of files dropped.

**DROPPED\_CURRENT**

Used with the %dr call-back function to return the number of the current file being processed (in the range from 1 to the value supplied by DROPPED\_COUNT).

**DROPPED\_ICON**

Is set to the value of the handle for the recently dropped icon (see ADD\_GRAPHICS\_ICON@).

**EDIT\_KEY**

Used with %eb to get the ASCII value of the last key pressed.

**FOCUS\_WINDOW**

This parameter will return the Window handle of the window with focus, or zero if no window of the application has the focus ( see %hw page 121).

**FOCUSSED\_WINDOW**

Used with ADD\_FOCUS\_MONITOR@. Gives the handle of the control which currently has the keyboard focus.

**GAINING\_FOCUS**

Returns 1 when an (%eb) edit box gains the focus.

**GIF\_MOUSE\_X**

Returns the pixel co-ordinate of a mouse click in a %gi image relative to the top left corner of the image.

**GIF\_MOUSE\_Y**

Returns the pixel co-ordinate of a mouse click in a %gi image relative to the top left corner of the image.

**GRAPHICS\_DEPTH**

Provides the ‘Y axis’ value for the graphics window, vital if the window is resized at any point.

**GRAPHICS\_DC**

Set during the call-back when a %dw[user\_resize] graphics area is re-sized (or when it is first created). The call-back can use this to re-draw the image.

**GRAPHICS\_HDC**

This provides the device context of the *drawing surface*. It will return zero if there is no current *drawing surface*. Using this handle you can write additional information to a *drawing surface* using API functions. The handle points to a bitmap so it is also possible to extract information from this area. Do not use this handle with %gr if metafile\_resize has been specified. If you use YIELD\_PROGRAM\_CONTROL@, or if you return from a call-back there is a possibility that the *drawing surface* will be resized (replacing its handle) or closed. It is therefore important to obtain the handle again in such circumstances. In general, obtain the handle, perform the actions you require, and discard the handle.

**GRAPHICS\_RESIZING**

This returns the value one if the window has been resized making it necessary to interrogate GRAPHICS\_DEPTH and GRAPHICS\_WIDTH.

**GRAPHICS\_WIDTH**

Provides the ‘X axis’ value for the graphics window, vital if the window is resized at any point.

**GRAPHICS\_MOUSE\_FLAGS**

Provides the flags associated with a mouse event in a graphics (%gr) or owner-draw (%dw) control.

The flags should be considered as a set of bits thus:

|            |    |                                |
|------------|----|--------------------------------|
| MK_LBUTTON | 1  | Left mouse button depressed    |
| MK_LBUTTON | 2  | Right mouse button depressed   |
| MK_SHIFT   | 4  | Keyboard shift key depressed   |
| MK_CONTROL | 8  | Keyboard control key depressed |
| MK_MBUTTON | 16 | Middle mouse button depressed  |

**GRAPHICS\_MOUSE\_X**

Provide the X co-ordinate of a mouse event in a graphics (%gr) or owner-draw (%dw) control.

**GRAPHICS\_MOUSE\_Y**

Provides the Y co-ordinate of a mouse event in a graphics (%gr) or owner-draw (%dw) control.

**INTERMEDIATE\_SCROLL**

Used with %hx and %vx to determine if the callback was called whilst the *thumb* is moving or after its release. Returns the value 1 if the *thumb* is moving, otherwise zero.

**LATEST\_FORMATTED\_WINDOW**

This is set to the Window handle of the most recently created formatted window. This handle should be used with care. In particular, if you make explicit Windows API calls using this handle, these may interfere with the action of `winio@`.

**LATEST\_VARIABLE**

Holds the address of a variable in a call-back attached to %rd, %rf or %rs.

**LATEST\_WINDOW**

Same as LATEST\_FORMATTED\_WINDOW.

**LISTBOX\_ITEM\_SELECTED**

Used in a call-back function for a list box. Returns 1 if the item was “selected” by double clicking on an item in the extended list; returns zero if the item was “moved to” by using a single click.

**LOSING\_FOCUS**

Returns 1 when an edit box (%eb) loses its focus.

**MESSAGE\_HWND**

This is set to the HWND parameter in a call-back from the %mg format.

**MESSAGE\_LPARAM**

This is set to the LPARAM parameter in a call-back from the %mg format.

**MESSAGE\_WPARAM**

This is set to the WPARAM parameter in a call-back from the %mg format.

**OPENGL\_DEPTH**

Returns the handle of the device context for the current %og region.

**OPENGL\_DEVICE\_CONTEXT**

Returns the pixel depth of a resized OpenGL (%og) region.

**OPENGL\_WIDTH**

Returns the pixel width of a resized OpenGL (%og) region.

**PIXELS\_PER\_H\_UNIT**

Translates from device dependant units used by %aw etc. to absolute pixel values

**PIXELS\_PER\_V\_UNIT**

Translates from device dependant units used by %aw etc. to absolute pixel values

**PRINTER\_COLLATE etc.**

For printer parameters see page 192.

**SCREEN\_DEPTH**

Returns the vertical screen resolution. Note that the available area of the largest possible window (the client area) will be normally be a little smaller than this because of the caption, border, etc..

**SCREEN\_WIDTH**

Returns the horizontal screen resolution.

**SHEET\_NO**

Holds the value of the current top-most property sheet (%ps). Initially it is 1.

**SPIN\_CONTROL\_USED**

This is set whilst in a call-back for a variable modified using a spin control.

**TEXT\_ARRAY\_CHAR etc.**

Parameters used with %tx are detailed on page 103.

**TREEVIEW\_ITEM\_SELECTED**

Returns the value one when an item has been selected from a treeview control (%tv) by means of a double mouse click.

**Return value** Returns the value of the specified integer parameter.

**See also** SET\_CLEARWIN\_INFO@

## CLEARWIN\_STRING@

**Purpose** To obtain string information from ClearWin+.

**Syntax** CHARACTER\*(\*) FUNCTION CLEARWIN\_STRING@( STR )  
CHARACTER\*(\*) STR

**Description** Parameters are specified as case insensitive character strings. They may be system or user-defined values.

ClearWin+ maintains a set of system strings that contain information from the system. The following have been defined:

**CALLBACK\_REASON**

See the end of this list.

CURRENT\_TEXT\_ITEM etc.

See Chapter 19.

**CURRENT\_MENU\_ITEM**

This option will return a string containing the current menu item selected. It is employed when using dynamic menus that have a shared call-back (see page 81).

**DROPPED\_FILE**

Is a string containing the full path and filename of the object dropped into a format window using %dr.

**CALLBACK\_REASON**

Whenever a call-back function is invoked, this string can be used to determine the reason for the call. The following list of reasons (upper case letters only) may be extended in the future, so programs should be designed to ignore reasons which they do not recognise.

|                        |                                                                              |
|------------------------|------------------------------------------------------------------------------|
| 'ACCELERATOR'          | Call associated with %ac.                                                    |
| 'BUTTON_PRESS'         | A button has been pressed.                                                   |
| 'COLOURING'            | User_colour event for edit box (%eb).                                        |
| 'DATA_ALTERATION'      | The contents of %eb, %rd (etc.) box have been altered.                       |
| 'DIRTY'                | Indicates an OpenGL window is in need of re-painting.                        |
| 'DRAG_AND_DROP'        | Call in response to dropping a file onto the window.                         |
| 'ENUMERATED_PARAMETER' | An enumerated parameter (%ep) has been selected from within a parameter box. |
| 'FILE_OPEN'            | Call from 'FILE_OPNR' or 'FILE_OPENW'.                                       |
| 'GAINING_FOCUS'        | Edit box gets input focus.                                                   |
| 'HYPERTEXT_CHANGE'     | Called when hypertext is being changed.                                      |
| 'HYPERTEXT_LINK'       | Call-back from activating a hypertext link in %ht.                           |
| 'ITEM_DOUBLE_CLICKED'  | Listbox item double clicked.                                                 |
| 'ITEM_SELECTED'        | Listbox item selected.                                                       |
| 'KEY_DOWN'             | A keyboard key has been pressed.                                             |
| 'LOSING_FOCUS'         | Edit box loses input focus.                                                  |
| 'MENU_SELECTION'       | A menu item has been invoked.                                                |

|                        |                                                                       |
|------------------------|-----------------------------------------------------------------------|
| 'MESSAGE_HOOK'         | Call of %mg related function.                                         |
| 'MOUSE_LEFT_CLICK'     | Left button down.                                                     |
| 'MOUSE_LEFT_RELEASE'   | Left button up.                                                       |
| 'MOUSE_DOUBLE_CLICK'   | Left double click.                                                    |
| 'MOUSE_MIDDLE_CLICK'   | Middle button down.                                                   |
| 'MOUSE_MIDDLE_RELEASE' | Middle button up.                                                     |
| 'MOUSE_MOVE'           | Mouse movement.                                                       |
| 'MOUSE_RIGHT_CLICK'    | Right button down.                                                    |
| 'MOUSE_RIGHT_RELEASE'  | Right button up.                                                      |
| 'RESIZE'               | Called when a control is being re-sized.                              |
| 'USER_PARAMETER'       | A user parameter (%up) has been selected from within a parameter box. |
| 'TIMER'                | Call associated with %dl.                                             |
| 'WINDOW_CLOSE'         | Call in response to %cc.                                              |
| 'WINDOW_STARTUP'       | Call in response to %sc.                                              |
| '+'                    | Called as part of a compound call-back.                               |
| '?'                    | Not currently in a call-back.                                         |

**Return value** Returns the value of the specified string parameter.

**See also** SET\_CLEARWIN\_STRING@

## CLEARWIN\_VERSION@

**Purpose** Gets the current ClearWin+ version information.

**Syntax** INTEGER FUNCTION CLEARWIN\_VERSION@()

**Return value** The minor version number is stored in the lower byte of the returned value and the major version number is stored in the second byte.

### Example

```
WINAPP
INCLUDE <windows.ins>
```

```

INTEGER version, i
version=clearwin_version@@()
i=winio@('%ca[Version information]&')
i=winio@('%n\nMajor %wd minor%wd%n\n',
+ version/256,AND(version,255))
END

```

## **CLIPBOARD\_TO\_SCREEN\_BLOCK@**

**Purpose** To copy a bitmap on the clipboard to a DIB.

**Syntax** INTEGER FUNCTION CLIPBOARD\_TO\_SCREEN\_BLOCK@( HDIB )  
INTEGER HDIB

**Description** This function copies a bitmap from the clipboard and returns a handle to the device independent bitmap that it creates. This handle can then be used, for example, with DIB\_PAINT@, PRINT\_DIB@ and EXPORT\_BMP@.

**Return value** A return value of zero indicates a failure (e.g. the current clipboard data is of the wrong type).

**See also** GET\_SCREEN\_BIB@, RELEASE\_SCREEN\_DIB@, IMPORT\_BMP@

## **CLOSE\_CD\_TRAY@**

**Purpose** To close an open CD-drive tray.

**Syntax** SUBROUTINE CLOSE\_CD\_TRAY@()

**Description** Simply by calling this routine any open CD drive with a mechanical drawer will close.

**See also** OPEN\_CD\_TRAY@, PLAY\_AUDIO\_CD@, SET\_CD\_POSITION@, STOP\_AUDIO\_CD@

**Example**

```

WINAPP
INTEGER i,ctrl,winio@
EXTERNAL cb_open,cb_close
ctrl=1
i=winio@('%ca[CD Tray Control]\n&')
i=winio@('%^bt[OPEN] ^bt[CLOSE]\n',cb_open,cb_close,ctrl)
END
INTEGER FUNCTION cb_open()
include <windows.ins>

```

```

CALL open_cd_tray@()
cb_open=2
END
INTEGER FUNCTION cb_close()
INCLUDE <windows.ins>
CALL close_cd_tray@()
cb_close=2
END

```

## CLOSE\_METAFILE@

**Purpose** To close a previously opened metafile.

**Syntax** INTEGER CLOSE\_METAFILE@( HANDLE )
 INTEGER HANDLE

**Description** If HANDLE is zero then the current *drawing surface* is used otherwise a handle to an existing *drawing surface* should be supplied.

**See also** OPEN\_METAFILE@, METAFILE\_TO\_CLIPBOARD@, PLAY\_CLIPBOARD\_METAFILE@

## CLOSE\_PRINTER@

**Purpose** To output to a graphics printer or plotter.

**Syntax** INTEGER FUNCTION CLOSE\_PRINTER@( HANDLE )
 INTEGER HANDLE

**Description** See OPEN\_PRINTER@. If HANDLE is zero, the current printer is closed.

**Return value** Returns 1 for success or zero for failure.

**See also** DO\_COPIES@, OPEN\_PRINTER@, SELECT\_PRINTER@

## CLOSE\_PRINTER\_ONLY@

**Purpose** To close a graphics printer or plotter without output.

**Syntax** INTEGER FUNCTION CLOSE\_PRINTER\_ONLY@( HANDLE )
 INTEGER HANDLE

**Description** Use this function instead of CLOSE\_PRINTER@ if you want to abort a printing process without output. See OPEN\_PRINTER@.

**Return value** Returns 1 for success or zero for failure.

**See also** CLOSE\_PRINTER@

## CLOSE\_WAV\_FILE@

**Purpose** To close a WAV file opened by OPEN\_WAV\_FILE\_READ@ or OPEN\_WAV\_FILE\_WRITE@.

**Syntax**

```
SUBROUTINE CLOSE_WAV_FILE@(HANDLE)
INTEGER HANDLE
```

**Description** This routine closes a WAV file opened for reading or writing. It is particularly important to call this routine to close a file open for writing before the program terminates.

## COPY\_FROM\_CLIPBOARD@

**Purpose** To copy data from the clipboard.

**Syntax**

```
INTEGER FUNCTION COPY_FROM_CLIPBOARD@(BUFFER, NUM, TYPE)
CHARACTER*(*) BUFFER
INTEGER NUM, TYPE
```

**Description** Use this routine to copy anything from the clipboard. The following data types can exist in the clipboard, though not all are directly supported by Clearwin+:

|                    |                                             |
|--------------------|---------------------------------------------|
| CF_BITMAP          | Bitmap data                                 |
| CF_DIB             | A BITMAPINFO structure followed by a bitmap |
| CF_DIF             | Data interchange format                     |
| CF_DSPBITMAP       | Private data stored in a bitmap format      |
| CF_DSPMETAFILEPICT | Private data stored in metafile format      |
| CF_DSPETEXT        | Private data stored in text format          |
| CF_METAFILEPICT    | The data is in metafile format              |
| CF_OEMTEXT         | The data is in OEM text format              |
| CF_OWNERDISPLAY    | The data is a private format                |
| CF_PALETTE         | A palette                                   |

|         |                                |
|---------|--------------------------------|
| CF_RIFF | Resource interchange format    |
| CF_SYLK | Microsoft symbolic link format |
| CF_TEXT | Text format                    |
| CF_TIFF | Tag image file format          |
| CF_WAVE | Wave format                    |

The BUFFER will contain NUM bytes copied from the clipboard. To determine the length of CF\_TEXT or CF\_OEMTEXT the routine SIZEOF\_CLIPBOARD\_TEXT@ can be called.

**Return value** Returns 1 for success or zero for failure.

**See also** COPY\_TO\_CLIPBOARD@, CLIPBOARD\_TO\_SCREEN\_BLOCK@,  
SIZEOF\_CLIPBOARD\_TEXT@

#### Example

```
INTEGER addr,len,i
CHARACTER buffer*1024
len=sizeof_clipboard_text()
IF(len.GT.0)THEN
 CALL get_storage(addr,len)
 IF(addr.GT.0)THEN
 i=copy_from_clipboard(addr,len, CF_TEXT)
 CALL move@(buffer,CCORE1(addr),len)
 ENDIF
ENDIF
```

## COPY\_GRAPHICS\_REGION@

**Purpose** To copy screen blocks in *drawing surfaces* defined by %gr etc..

**Syntax** INTEGER FUNCTION COPY\_GRAPHICS\_REGION@(
+ DEST\_GR, DX, DY, DWIDTH, DHEIGHT,
+ SRC\_GR, SX, SY, SWIDTH, SHEIGHT, COPY\_MODE )
INTEGER DEST\_GR, DX, DY, DWIDTH, DHEIGHT,
+ SRC\_GR, SX, SY, SWIDTH, SHEIGHT, COPY\_MODE

**Description** You must have at least one *drawing surface* open (see for example %`gr on page 136). DEST\_GR is the handle of the destination *drawing surface* and SRC\_GR is the source *drawing surface*, however, the handles specified can be the same. If you set either or both to zero then the current *drawing surface* is assumed for the source and/or destination.

`DX`, `DY`, `DWIDTH` and `DHEIGHT` specify the destination rectangular region.

`SX`, `SY`, `SWIDTH` and `SHEIGHT` specify the source rectangular region.

If `DWIDTH = SWIDTH` and `DHEIGHT = SHEIGHT` then a normal copy will occur. If there are any differences then the image will be ‘stretched’ accordingly.

`COPY_MODE` defines the method of copying. Windows defines 255 different copy methods (ROPs). The most useful are described below:

| Hex Value | Name        | Logical Description                         |
|-----------|-------------|---------------------------------------------|
| CC0020    | SRCCOPY     | <i>source</i> (most common - a direct copy) |
| 8800C6    | SRCAND      | <i>source and destination</i>               |
| 660046    | SRCINVERT   | <i>source xor destination</i>               |
| 440328    | SRCERASE    | <i>source and not destination</i>           |
| EE0086    | SRCPAINT    | <i>source or destination</i>                |
| 330008    | NOTSRCCOPY  | <b>not</b> <i>source</i>                    |
| 1100A6    | NOTSRCERASE | <b>not</b> ( <i>source or destination</i> ) |
| BB0226    | MERGEPAINT  | <b>not</b> <i>source or destination</i>     |
| 550009    | DSTINVERT   | <b>not</b> <i>destination</i>               |
| 000042    | BLACK       | 0                                           |
| FF0062    | WHITE       | 1                                           |

**Return value** Returns 1 for success or zero for failure.

**Notes** Under Win32 it is possible to stretch a *drawing surface* when copying from the screen to a printer but not every printer supports this mode.

**See also** `SCROLL_GRAPHICS@`, `SELECT_GRAPHICS_OBJECT@`, `CREATE_GRAPHICS_REGION@`, `DELETE_GRAPHICS_REGION@`

---

## **COPY\_TO\_CLIPBOARD@**

**Purpose** To copy data to the clipboard.

**Syntax** `INTEGER FUNCTION COPY_TO_CLIPBOARD@( BUFFER, NUM, TYPE )`

CHARACTER\*(\*) BUFFER  
INTEGER NUM, TYPE

**Description** NUM contents of BUFFER will be placed into the Windows clipboard and will be of the TYPE specified. For a list of the available data types see the COPY\_FROM\_CLIPBOARD@ function.

**Return value** Returns 1 on success or zero on failure.

**See also** COPY\_FROM\_CLIPBOARD@, CLIPBOARD\_TO\_SCREEN\_BLOCK@,  
GRAPHICS\_TO\_CLIPBOARD@, SIZEOF\_CLIPBOARD\_TEXT@

---

## **CREATE\_BITMAP@**

**Purpose** To create a bitmap from bitmap data.

**Syntax** INTEGER FUNCTION CREATE\_BITMAP@( PTR )  
INTEGER PTR

**Description** Creates a bitmap from a pointer to data that is in the same format as a .BMP file. When the program is terminated the bitmap will be deleted. The function returns a handle that can be used with %`bm or in Windows API functions that require a HBITMAP handle.

**Return value** Returns a handle to the new bitmap. A zero return indicates failure.

**See also** MAKE\_BITMAP@

---

## **CREATE\_CURSOR@**

**Purpose** To create a cursor from data in a .CUR file.

**Syntax** INTEGER FUNCTION CREATE\_CURSOR@( PTR )  
INTEGER PTR

**Description** Creates a cursor from a pointer to data that is in the same format as a .CUR file. When the program is terminated the cursor will be deleted. The function returns a handle that can be used, for example, with %`cu or in Windows API functions that require a HCURSOR handle.

**Return value** Returns a handle to the new cursor. A zero return indicates failure.

## CREATE\_GRAPHICS\_REGION@

**Purpose** To create an internal (off-screen) *drawing surface*.

**Syntax**

```
INTEGER FUNCTION CREATE_GRAPHICS_REGION@(HANDLE,
+ WIDTH, HEIGHT)
 INTEGER HANDLE, WIDTH, HEIGHT
```

**Description** This routine is used to create internal *drawing surfaces*. For example, it can be used to display part of a large bitmap. Areas of the buffer can be copied into a %gr region (for example) using COPY\_GRAPHICS\_REGION@. If you make extensive use of this function it is recommended that you delete unwanted surfaces with a call to DELETE\_GRAPHICS\_REGION@.

HANDLE is an input value, chosen by the programmer, that must be unique. It is used to identify the surface in calls to related functions. WIDTH, HEIGHT specify the size in pixels of the region.

The colour mode for the surface will be determined by the default colour mode or by the colour mode of the current %gr *drawing surface*.

**Return value** Returns 1 for success, otherwise zero.

**See also** DELETE\_GRAPHICS\_REGION@, COPY\_GRAPHICS\_REGION@, SCROLL\_GRAPHICS@

## CREATE\_WINDOW@

**Purpose** To create and display a *ClearWin* window.

**Syntax**

```
INTEGER FUNCTION CREATE_WINDOW@(TITLE,X,Y,XW,YW)
 INTEGER X,Y,XW,YW
 CHARACTER*(*) TITLE
```

**Description** Creates a window of width XW, height YW and title TITLE at position (X, Y).

Do not confuse this function with the similarly named routine in the Windows API called **CreateWindow**.

**Return value** Returns a standard Windows handle.

## **DEFINE\_FILE\_EXTENSION@**

**Purpose** Allows application registry under Windows 95.

**Syntax**

```
SUBROUTINE DEFINE_FILE_EXTENSION@(EXTNAME, PATH,
+ DESCR, ICON_IDX, NEW)
CHARACTER*129 NAME, PATH, DESCRIPTION
INTEGER ICON_INDEX, NEW OPTION
```

**Description** Under Windows 95 it is possible to register an application with the system so that if a data file is opened via Explorer your application will be called to process it.

The EXTNAME variable is a string that contains the extension, the PATH must contain the full path and program name. A text description should be supplied in DESCRIPTION. The ICON\_INDEX selects the icon to be used by Windows. If you specify -1 no icon is used otherwise the relevant icon is used. i.e. if you have four icons in your resource, by placing a value of 2 in ICON\_INDEX the second icon resource will be used. NEW OPTION option should be set to a non zero value to activate the file type addition to Windows 95.

For example:

```
CHARACTER*129 pname
CALL get_program_name@(pname)
k=define_file_extension@('.ICO',pname,
+ 'Icon file editor',0,1)
```

You can obtain the file name passed to your program, by Windows 95 as if you were examining the arguments on the command line.

For example:

```
...
CHARACTER*129 cmnam@,filetoopen
filetoopen=cmnam@()
...
```

GET\_PROGRAM\_NAME@ and CMNAM@ are Salford library functions.

---

## **DELETE\_GRAPHICS\_REGION@**

**Purpose** To remove an internal *drawing surface*.

**Syntax** INTEGER FUNCTION DELETE\_GRAPHICS\_REGION@( HANDLE )  
INTEGER HANDLE

**Description** This routine will remove from memory a *drawing surface* that has previously been defined by CREATE\_GRAPHICS\_REGION@. HANDLE is the value used in an earlier call to CREATE\_GRAPHICS\_REGION@.

**Return value** Returns 1 for success, otherwise zero.

**See also** CREATE\_GRAPHICS\_REGION@, COPY\_GRAPHICS\_REGION@, SCROLL\_GRAPHICS@

---

## **DESTROY\_WINDOW@**

**Purpose** To close and destroy a displayed *ClearWin* window.

**Syntax** SUBROUTINE DESTROY\_WINDOW@(W)  
INTEGER W

**Description** Destroys a window specified by W and any child windows that may be owned by it. Fonts owned by the window are not destroyed. W is the window handle returned by CREATE\_WINDOW@.

---

## **DIB\_PAINT@**

**Purpose** To display a device independent bitmap (DIB).

**Syntax** INTEGER FUNCTION DIB\_PAINT@( HORIZ, VERT, HDIB,  
+ FUNCTION, MODE )  
INTEGER HORIZ, VERT, HDIB, FUNCTION, MODE

**Description** Displays the DIB with handle HDIB on the current device with HORIZ and VERT relative displacement (they may be negative). FUNCTION selects the type of logical restore operation with respect to the previous screen :

- |   |                       |
|---|-----------------------|
| 0 | REPLACE former pixel  |
| 1 | AND with former pixel |
| 2 | OR with former pixel  |

### 3 XOR with former pixel

MODE specifies if

- 0 the dib palette should be used,
  - 1 the current palette should be used and the image not dithered,
  - 2 dithering should be used.

Before you can call this function the DIB must first be loaded using GET\_SCREEN\_BLOCK@, IMPORT\_BMP@, IMPORT\_PCX@, or CLIPBOARD\_TO\_SCREEN\_BLOCK@.

**Return value** Returns 1 if the function is successful otherwise zero.

## DISPLAY DIB BLOCK@

**Purpose** To display a device independent bitmap.

**Syntax** SUBROUTINE DISPLAY\_DIB\_BLOCK@( X,Y,PARRAY,AW,AH,AX,AY,W,H,  
+ FUNCTION,MODE,ERCODE )  
CHARACTER PARRAY(3,AW,AH)  
INTEGER X,Y,AW,AH,AX,AY,W,H,FUNCTION,MODE,ERCODE

**Description** All of the arguments are input values except for ERCODE which returns the error status.

This routine transfers a rectangular block of size  $W \times H$  from the array PARRAY at position (AX, AY) to the position (X, Y) on the current graphics device (e.g. a %gr window or a printer). FUNCTION and MODE are the same as in DIB\_PAINT@ and will normally be set to zero. The variable ERCODE is returned as zero if the process is successful.

**See also** GET\_IM\_INFO@, GET\_DIB\_SIZE@, GET\_DIB\_BLOCK@, PUT\_DIB\_BLOCK@, PRINT\_DIB@

**Example** See page 145

**DISPLAY POPUP MENU@**

**Purpose** To activate a popup menu defined using %pm.

**Syntax** SUBROUTINE DISPLAY\_POPUP\_MENU@()

**Description** This subroutine will activate a pre-defined popup menu. It can be used when the right mouse button is pressed over a %gr *drawing surface* that has FULL\_MOUSE\_INPUT activated (see page 136).

## DO\_COPIES@

**Purpose** To produce multiple printer copies of a graphics image using metafiles.

**Syntax**

```
INTEGER DO_COPIES@(HANDLE, NUM)
INTEGER HANDLE, NUM
```

**Description** Sends NUM copies to the printer. A metafile must have been created with a call to OPEN\_METAFILE@ and also closed by CLOSE\_METAFILE@ before DO\_COPIES@ can be used. The CLEARWIN\_INFO@ parameter PRINTER\_COPIES will return the number of copies the user has selected.

The call to CLOSE\_PRINTER@ will have automatically sent one copy to the printer.

**See also** OPEN\_PRINTER@, CLOSE\_PRINTER@

**Example** See page 195.

## DRAW\_BEZIER@

**Purpose** To draw a Bezier spline.

**Syntax**

```
SUBROUTINE DRAW_BEZIER@(X, Y, N, COLOUR)
INTEGER X(N),Y(N),N,COLOUR
```

**Description** X and Y are arrays giving the co-ordinates of the N knots and COLOUR is the colour value using the current *colour mode*.

## DRAW\_CHARACTERS@

**Purpose** To draw text on the current *drawing surface*.

**Syntax**

```
SUBROUTINE DRAW_CHARACTERS@(STR,IH,IV,ICOL)
CHARACTER*(*) STR
INTEGER IH,IV,ICOL
```

**Description** DRAW\_CHARACTERS@ draws text, using the current font and attributes, at the point (IH, IV). These co-ordinates represent the bottom left-hand corner of the first character. STR contains the character string to be drawn. The text is positioned to the nearest pixel. ICOL provides the colour for the text that appears on the existing background using the current *colour mode*.

**See also** SELECT\_FONT@, SCALE\_FONT@, UNDERLINE\_FONT@, ITALIC\_FONT@, BOLD\_FONT@, SIZE\_IN\_PIXELS@, SIZE\_IN\_POINTS@, ROTATE\_FONT@, GET\_FONT\_NAME@, CHOOSE\_FONT@.

## DRAW\_ELLIPSE@

**Purpose** To draw an ellipse on the current *drawing surface*.

**Syntax** SUBROUTINE DRAW\_ELLIPSE@(IXC,IYC,IA,IB,ICOL)  
INTEGER IXC,IYC,IA,IB,ICOL

**Description** DRAW\_ELLIPSE@ draws an ellipse with centre at (IXC, IYC), with horizontal semi-axis IA, vertical semi-axis IB and colour ICOL using the current *colour mode*. This routine can be used to produce a circle on the *drawing surface* if the axes are scaled appropriately.

**See also** DRAW\_FILLED\_ELLIPSE@

## DRAW\_FILLED\_ELLIPSE@

**Purpose** To fill an ellipse on the current *drawing surface*.

**Syntax** SUBROUTINE DRAW\_FILLED\_ELLIPSE@(IXC,IYC,IA,IB,ICOL)  
INTEGER IXC,IYC,IA,IB,ICOL

**Description** DRAW\_FILLED\_ELLIPSE@ fills an ellipse with centre at (IXC, IYC), with horizontal semi-axis IA, vertical semi-axis IB and colour ICOL using the current *colour mode*. This routine can be used to fill a circle if the axes are scaled appropriately.

**See also** DRAW\_ELLIPSE@

## DRAW\_FILLED\_POLYGON@

**Purpose** To fill a polygon on the current *drawing surface*.

**Syntax**

```
SUBROUTINE DRAW_FILLED_POLYGON@(IX,IY,N,ICOL)
 INTEGER IX(N),IY(N)
 INTEGER N,ICOL
```

**Description** DRAW\_FILLED\_POLYGON@ draws a straight line from (IX(1), IY(1)) to (IX(2), IY(2)), and continues until (IX(N), IY(N)) and fills the enclosed region. ICOL specifies the fill colour value using the current *colour mode*.

**See also** DRAW\_POLYLINE@

## DRAW\_FILLED\_RECTANGLE@

**Purpose** To fill a rectangle on the current *drawing surface*.

**Syntax**

```
SUBROUTINE DRAW_FILLED_RECTANGLE@(IX1,IY1,IX2,IY2,ICOL)
 INTEGER IX1,IY1,IX2,IY2,ICOL
```

**Description** DRAW\_FILLED\_RECTANGLE@ fills a rectangle in colour ICOL using the current *colour mode*. (IX1, IY1) and (IX2, IY2) are opposite corners.

**See also** DRAW\_RECTANGLE@

## DRAW\_LINE\_BETWEEN@

**Purpose** To draw a straight line on the current *drawing surface*.

**Syntax**

```
SUBROUTINE DRAW_LINE_BETWEEN@(IX1,IY1,IX2,IY2,ICOL)
 INTEGER IX1,IY1,IX2,IY2,ICOL
```

**Description** DRAW\_LINE\_BETWEEN@ draws a line with colour value ICOL (using the current *colour mode*) from (IX1, IY1) to (IX2, IY2). The coordinates are pixel numbers relative to the top left of the *drawing surface*.

**See also** DRAW\_POLYLINE@

## DRAW\_POINT@

**Purpose** To set a single pixel colour on the current *drawing surface*.

**Syntax** SUBROUTINE DRAW\_POINT@(IH,IV,ICOL)  
INTEGER IH,IV,ICOL

**Description** DRAW\_POINT@ sets the pixel at (IH,IV) on the current *drawing surface* to colour value ICOL using the current *colour mode*.

**See also** GET\_POINT@, GET\_RGB\_VALUE@

---

## DRAW\_POLYLINE@

**Purpose** To draw a number of connected straight lines.

**Syntax** SUBROUTINE DRAW\_POLYLINE@(IX,IY,N,ICOL)  
INTEGER IX(N),IY(N)  
INTEGER N,ICOL

**Description** DRAW\_POLYLINE@ draws a straight line from (IX(1), IY(1)) to (IX(2), IY(2)), and continues until (IX(N), IY(N)). That is, it joins N points with straight lines. ICOL specifies the colour value using the current *colour mode*. For a closed polygon, simply set the last pair of coordinates equal to the first pair.

**See also** DRAW\_LINE\_BETWEEN@, DRAW\_FILLED\_POLYGON@

---

## DRAW\_RECTANGLE@

**Purpose** To draw a rectangle on the current *drawing surface*.

**Syntax** SUBROUTINE DRAW\_RECTANGLE@(IX1,IY1,IX2,IY2,ICOL)  
INTEGER IX1,IY1,IX2,IY2,ICOL

**Description** DRAW\_RECTANGLE@ draws a rectangle in colour ICOL using the current *colour mode*. (IX1, IY1) and (IX2, IY2) are opposite corners.

**See also** DRAW\_FILLED\_RECTANGLE@

## EXPORT\_BMP@

**Purpose** To exports a device independent bitmap to a .BMP file.

**Syntax** SUBROUTINE EXPORT\_BMP@( HDIB, FILENAME, ERROR )  
INTEGER DIB\_HANDLE,ERROR  
CHARACTER\*(\*) FILENAME

**Description** Writes a DIB to a bitmap file specified by FILENAME. HDIB is a handle returned for example by GET\_SCREEN\_DIB@, CLIPBOARD\_TO\_SCREEN\_BLOCK@ or IMPORT\_BMP@.

ERROR returns one of the following:

|   |                         |
|---|-------------------------|
| 0 | success                 |
| 1 | unable to open file     |
| 3 | unable to write to file |

**See also** IMPORT\_BMP@, WRITE\_GRAPHICS\_TO\_BMP@, DIB\_PAINT@, PRINT\_DIB@

---

## EXPORT\_PCX@

**Purpose** To export a device independent bitmap to a .PCX file.

**Syntax** SUBROUTINE EXPORT\_PCX@( HDIB, FILENAME, ERROR )  
INTEGER DIB\_HANDLE,ERROR  
CHARACTER\*(\*) FILENAME

**Description** Writes a DIB to a PCX file specified by FILENAME. HDIB is a handle returned for example by GET\_SCREEN\_DIB@, CLIPBOARD\_TO\_SCREEN\_BLOCK@ or IMPORT\_PCX@.

ERROR returns one of the following:

|   |                         |
|---|-------------------------|
| 0 | success                 |
| 1 | unable to open file     |
| 3 | unable to write to file |

**See also** IMPORT\_PCX@, WRITE\_GRAPHICS\_TO\_PCX@, DIB\_PAINT@, PRINT\_DIB@

## FEED\_WKEYBOARD@

- Purpose** To insert a key into a *ClearWin* window keyboard buffer.
- Syntax** SUBROUTINE FEED\_WKEYBOARD@( KEY , ERROR )  
INTEGER KEY , ERROR
- Description** This routine is only for use in a *ClearWin* window (e.g. with %cw). KEY is the ASCII code for the key to be inserted. Currently ERROR always returns zero for success.
- See also** FLUSH\_WKEYBOARD@, GET\_WKEY@, GET\_WKEY1@, WKEY\_WAITING@

## FILL\_SURFACE@

- Purpose** To fill an area of the current *drawing surface* with a given colour.
- Syntax** SUBROUTINE FILL\_SURFACE@( XSTART, YSTART, SURF\_COL  
+ FILL\_COL )  
INTEGER XSTART, YSTART, SURF\_COL, FILL\_COL
- Description** XSTART, YSTART are the co-ordinates of the focus, FILL\_COL is the colour to fill with and SURF\_COL is the current colour of the region to fill. The colour values relate to the current *colour mode*.
- See also** FILL\_TO\_BORDER@

## FILL\_TO\_BORDER@

- Purpose** To fill an area of the current *drawing surface* with a given colour.
- Syntax** SUBROUTINE FILL\_TO\_BORDER@( XSTART, YSTART, FILL\_COL  
+ STOP\_COL )  
INTEGER XSTART, YSTART, FILL\_COL, STOP\_COL
- Description** XSTART, YSTART are the co-ordinates of the focus, FILL\_COL is the colour to fill with and STOP\_COL is the colour of the closed boundary of the area to fill. The colour values relate to the current *colour mode*
- See also** FILL\_SURFACE@

## **FLUSH\_WKEYBOARD@**

**Purpose** To empty a *ClearWin* window keyboard buffer.

**Syntax** SUBROUTINE FLUSH\_WKEYBOARD@( )

**Description** This routine is only for use in a *ClearWin* window (e.g. with %cw).

**See also** FEED\_WKEYBOARD@, GET\_WKEY@, GET\_WKEY1@, WKEY\_WAITING@

---

## **FONT\_METRICS@**

**Purpose** To get the system metrics for the default font and the current *drawing surface*.

**Syntax** SUBROUTINE FONT\_METRICS@( METRICS )  
INTEGER METRICS(20)

**Description** The array is returned with values for the following 20 system metrics:  
Height, Ascent, Descent, InternalLeading, ExternalLeading, AveCharWidth,  
MaxCharWidth, Weight, Overhang, DigitizedAspectX, DigitizedAspectY,  
FirstChar, LastChar, DefaultChar, BreakChar, Italic, Underlined, StruckOut,  
PitchAndFamily, CharSet.

---

## **GET\_BITMAP\_DC@**

**Purpose** To obtain the device context of a bitmap which may be written to using the Windows API graphics functions and used in a %dw format.

**Syntax** INTEGER FUNCTION GET\_BITMAP\_DC@( H\_BITS, V\_BITS )  
INTEGER H\_BITS,V\_BITS

**Description** The bitmap is created of size H\_BITS × V\_BITS pixels. The device context can be destroyed by RELEASE\_BITMAP\_DC@, but will be returned to the system at normal program termination.

RELEASE\_BITMAP\_DC@ should be used when multiple bitmaps are created and when they need not be saved.

**Return value** The function returns a handle to the device context.

**See also** CLEAR\_BITMAP@, RELEASE\_BITMAP\_DC@, %dw

## **GET\_CLEARWIN\_TEXT@**

**Purpose** To get text from a *ClearWin* window.

**Syntax** INTEGER FUNCTION GET\_CLEARWIN\_TEXT@( WIN,BUFFER,COPY )  
 INTEGER WIN,COPY  
 CHARACTER\*(\*) BUFFER

**Description** WIN is the handle of *ClearWin* window in question. To obtain the handle of the default window use:

```
WIN=GET_DEFAULT_WINDOW@()
```

**Return value** If COPY is zero, the function returns the length of the required buffer but no data is copied. If COPY is 1, the data is also copied to the buffer.

## **GET\_CURRENT\_DC@**

**Purpose** To get the device context of the current *drawing surface*.

**Syntax** SUBROUTINE GET\_CURRENT\_DC@( DC )  
 INTEGER DC

**See also** SELECT\_GRAPHICS\_OBJECT@

## **GET\_DEFAULT\_WINDOW@**

**Purpose** To return the handle of the current default *ClearWin* window for standard Fortran I/O.

**Syntax** INTEGER FUNCTION GET\_DEFAULT\_WINDOW@()

## **GET\_DIB\_BLOCK@**

**Purpose** To read a BMP format file into a character array.

**Syntax** SUBROUTINE GET\_DIB\_BLOCK@( FILENAME, PARRAY, AW, AH, ADX,  
 + ADY, W, H, DX, DY, ERCODE )  
 CHARACTER\*(\*) FILENAME  
 CHARACTER PARRAY(3,AW,AH)

```
INTEGER AW,AH,ADX,ADY,W,H,DX,DY,ERCODE
```

**Description** All of the arguments are input values except for PARRAY which contains the returned data and ERCODE which returns the error status.

FILENAME is the name of the BMP file.

AW and AH are dimensions of PARRAY.

ADX and ADY are offsets used when writing the data to PARRAY.

W and H are the width and height of the image data to be copied.

DX and DY are offsets used when reading the data from the file.

The first dimension of PARRAY is used to store the red, green and blue values for each pixel. ERCODE will return:

|    |                |
|----|----------------|
| 0  | SUCCESS        |
| 2  | READ ERROR     |
| 4  | MEMORY ERROR   |
| 5  | NOT A BMP FILE |
| 10 | BAD FILE       |
| 13 | BAD NUMBER     |
| 16 | OTHER ERROR    |

**See also** GET\_DIB\_SIZE@, PUT\_DIB\_BLOCK@, DIB\_BLOCK\_PAINT@

**Example** See page 145.

## GET\_DIB\_SIZE@

**Purpose** To get information from a BMP data file.

**Syntax**

```
SUBROUTINE GET_DIB_SIZE@(FILENAME, WIDTH, HEIGHT, NBBP,
+ ERCODE)
CHARACTER*(*) FILENAME
INTEGER WIDTH, HEIGHT, NBBP, ERCODE
```

**Description** This routine is similar to GET\_IM\_INFO@ but returns the number of bits per pixel NBBP as follows:

| NBBP | Number of colours |
|------|-------------------|
| 1    | 2                 |
| 4    | 16                |
| 8    | 256               |
| 24   | 16MB              |

**See also** GET\_DIB\_BLOCK@, PUT\_DIB\_BLOCK@, DIB\_BLOCK\_PAINT@

**Example** See page 145.

## GET\_FILTERED\_FILE@

**Purpose** To access the standard ‘open’ dialog box.

**Syntax** SUBROUTINE GET\_FILTERED\_FILE@(TITLE, FILE, PATH,  
+ FILTERNAMES, FILTERSPECS, NFILTERS, MUSTEXIST )  
CHARACTER(\*) TITLE, FILE, PATH, FILTERNAMES,  
+ FILTERSPECS  
INTEGER NFILTERS, MUSTEXIST

**Description** This function can be used instead of %fs and %ft in cases where the filters need to be set dynamically. The arguments are as follows.

|              |                                                                                |
|--------------|--------------------------------------------------------------------------------|
| TITLE:       | The caption for the dialog box.                                                |
| FILE:        | The full path returned by the dialog box.                                      |
| PATH:        | The input path.                                                                |
| FILTERNAMES: | An array of strings for the description of file filters.                       |
| FILTERSPECS: | An array of strings for the corresponding filters.                             |
| NFILTERS     | The number of file filters.                                                    |
| MUSTEXIST:   | An input value that is set to 1 if the file to be selected must already exist. |

**Notes** After this routine has been called, CLEARWIN\_INOF@(‘FILTER\_ITEM\_SELECTED’) provides the index of the file filter that was selected by the user. This can be called, for example, to detect if the file was saved using a different file extension.

**Example**

```

WINAPP
INTEGER i,winio@,open_func
EXTERNAL open_func
i=winio@('%ca[Image info]&')
i=winio@('%mn[&File[&Open,E&xit]]',open_func,'EXIT')
END
C-----
INTEGER FUNCTION open_func()
INCLUDE <windows.ins>
CHARACTER file*129,format*12,title*20,path*129
INTEGER NFILTERS
PARAMETER(NFILTERS=2)
CHARACTER*20 filtname(NFILTERS),filtspec(NFILTERS)
title='Open Bitmap File'
file=' '
path='c:\windows'
filtname(1)='Bitmap files'
filtspec(1)='*.bmp'
filtname(2)='All files'
filtspec(2)='*.*'
CALL get_filtered_file@(title,file,path,filtname,filtspec,
+
NFILTERS,1)
.
.
.
open_func=1
END

```

**GET\_FONT\_ID@**

**Purpose** To get the current font identifier.

**Syntax** SUBROUTINE GET\_FONT\_ID@( ID )
  
INTEGER ID

**Description** ID is returned as the identifier of the font for the current *drawing surface*.

**See also** CHOOSE\_FONT@, SELECT\_FONT\_ID@

## GET\_FONT\_NAME@

**Purpose** To return the name of the loaded font.

**Syntax** SUBROUTINE GET\_FONT\_NAME@( NAME, NUM )  
 CHARACTER\*(\*) NAME  
 INTEGER NUM

**Description** The name of an installed font is returned in the NAME parameter. NUM references the installed fonts and should start at 1. If a call is made and there is no associated font to the number supplied an empty string will be returned.

**Example**

```
INCLUDE <windows.ins>
CHARACTER*20 name
DO k=1,3000
 CALL get_font_name@(name,k)
 IF(name.EQ.' ')STOP
 PRINT *,name
ENDDO
END
```

## GET\_GRAPHICAL\_RESOLUTION@

**Purpose** To determine the width and height of the current *drawing surface*.

**Syntax** SUBROUTINE GET\_GRAPHICAL\_RESOLUTION@( WIDTH, HEIGHT )  
 INTEGER WIDTH, HEIGHT

**Description** WIDTH and HEIGHT are set to the values for the current *drawing surface*.

## GET\_GRAPHICS\_SELECTED\_AREA@

**Purpose** To get the current %gr ‘rubber-band’ selection.

**Syntax** SUBROUTINE GET\_GRAPHICS\_SELECTED\_AREA@( X1,Y1,X2,Y2 )  
 INTEGER X1, Y1, X2, Y2

**Description** The four parameters are set to the values of the current graphics selection which is defined by the ‘rubber-band’ line activated with a call to SET\_GRAPHICS\_SELECTION@. The selection mode must be set to either 1 or 2 before

this routine can be used.

**See also** SET\_GRAPHICS\_SELECTION@, %gr

**Example** See page 143.

## GET\_IM\_INFO@

**Purpose** To obtain information from a graphics file.

**Syntax**

```
 SUBROUTINE GET_IM_INFO@(FILENAME, WIDTH, HEIGHT
 + NB_COLOURS, NB_IMAGES, FORMAT, ERROR)
 CHARACTER*(*) FILENAME,FORMAT
 INTEGER WIDTH(n), HEIGHT(n), NB_COLOURS(n)
 INTEGER NB_IMAGES, ERROR
```

**Description** This subroutine accesses the information that is contained within a .BMP or .PCX file supplied in FILENAME. The relevant data is returned in the supplied parameters. The WIDTH and HEIGHT will contain the dimension of the image. WIDTH, HEIGHT and NB\_COLOURS are arrays. NB\_IMAGES is returned as the number of images and n should set at a value which is always greater than or equal to NB\_IMAGES. FORMAT will be returned as a single character string, either 'BMP' or 'PCX'.

ERROR will return:

|    |                |
|----|----------------|
| 0  | SUCCESS        |
| 2  | READ ERROR     |
| 5  | NOT A BMP FILE |
| 10 | BAD FILE       |
| 15 | UNKNOWN FORMAT |

**See also** GET\_DIB\_SIZE@

## GET\_MATCHED\_COLOUR@

**Purpose** To find the closest RGB colour match for the current *drawing surface*.

**Syntax**

```
 INTEGER FUNCTION GET_MATCHED_COLOUR@(COLOUR)
 INTEGER COLOUR
```

**Description** GET\_MATCHED\_COLOUR@ selects the RGB palette colour that is the best match to the

supplied RGB COLOUR on the current *drawing surface*. An error condition arises if RGB colour mode is not selected on the current *drawing surface*. If there is no current *drawing surface* then the function returns its input value. Likewise, the function returns its input value if the region is a printer or if the graphics device is a *high colour* device (i.e. more than 8 bits per pixel).

**Return value** Returns the closest match for the current *drawing surface*.

**See also** GET\_NEAREST\_SCREEN\_COLOUR@, USE\_RGB\_COLOURS@

**Example** See page 155.

## GET\_MOUSE\_INFO@

**Purpose** To obtain the position of the mouse, the mouse buttons, and the keyboard shift keys at the time when the last owner-draw (%^dw) or graphics region (%^gr) call-back function was called.

**Syntax** SUBROUTINE GET\_MOUSE\_INFO@( X, Y, FLAGS )  
INTEGER X,Y,FLAGS

**Description** This routine should be called immediately on entry to the call-back function. It does not make sense to call this function in other contexts. The X and Y values are pixel positions relating to the bitmap associated with the format code (%gr or %dw). Pixel co-ordinates are measures from (0,0) at the top left corner of the area. FLAGS contains the following flags (the parameters will be found in *windows.ins*):

|            |    |                                |
|------------|----|--------------------------------|
| MK_LBUTTON | 1  | Left mouse button depressed    |
| MK_RBUTTON | 2  | Right mouse button depressed   |
| MK_SHIFT   | 4  | Keyboard shift key depressed   |
| MK_CONTROL | 8  | Keyboard control key depressed |
| MK_MBUTTON | 16 | Middle mouse button depressed  |

**Notes** CLEARWIN\_INFO@ can be used instead. These values are returned using GRAPHICS\_MOUSE\_FLAGS, GRAPHICS\_MOUSE\_X and GRAPHICS\_MOUSE\_Y.

## GET\_NEAREST\_SCREEN\_COLOUR@

**Purpose** To find the closest colour match for the screen from the system palette.

**Syntax** INTEGER FUNCTION GET\_NEAREST\_SCREEN\_COLOUR@( COLOUR)  
INTEGER COLOUR

**Return value** Returns the RGB colour from the Windows system palette that will be displayed when the specified colour value is used on the screen (the value returned by the Windows API function **GetNearestColor**).

**Notes** You can use this function to provide a colour for a *drawing surface* if you want to override the colour matching carried out automatically by ClearWin+. See page 156 for further details.

**See also** GET\_MATCHED\_COLOUR@

---

## GET\_POINT@

**Purpose** To get a pixel colour on the current *drawing surface*.

**Syntax** SUBROUTINE GET\_POINT@(IH,IV,ICOL)  
INTEGER IH,IV,ICOL

**Description** GET\_POINT@ gets the colour value ICOL using the current *colour mode* of the pixel at (IH, IV) on the current *drawing surface*. The colour value is given in terms of the current *colour mode* of the surface.

**See also** GET\_RGB\_VALUE@, DRAW\_POINT@

---

## GET\_PRINTER\_ORIENTATION@

**Purpose** To get the printer orientation as portrait or landscape.

**Syntax** INTEGER GET\_PRINTER\_ORIENTATION@( )

**Return vlaue** Returns 0 for portrait and 1 for landscape.

## **GET\_RGB\_VALUE@**

**Purpose** To get the RGB value of the colour of a pixel on the current *drawing surface*.

**Syntax** SUBROUTINE GET\_RGB\_VALUE@( HOR, VER, VALUE )  
INTEGER HOR,VER,VALUE

**Description** The horizontal and vertical co-ordinates locate the pixel on the current *drawing surface*. An RGB value is always returned irrespective of the current *colour mode* for the surface.

**See also** GET\_NEAREST\_SCREEN\_COLOUR@, SET\_COLOURS@, USE\_RGB\_COLOURS@

## **GET\_SCREEN\_DIB@**

**Purpose** To copy part of a graphics area.

**Syntax** SUBROUTINE GET\_SCREEN\_DIB@( HDIB, X1, Y1, X2, Y2)  
INTEGER HDIB, X1, Y1, X2, Y2

**Description** HDIB is returned as the handle of a new device context that contains a copy of part of the current graphics area. X1, Y1 are input as the pixel co-ordinates of the top left hand corner of the area to copy whilst X2, Y2 are the corresponding co-ordinates of the bottom right hand corner.

**See also** DIB\_PAINT@, PRINT\_DIB@, EXPORT\_BMP@, RELEASE\_SCREEN\_DIB@

## **GET\_SYSTEM\_FONT@**

**Purpose** To get the attributes of the system font.

**Syntax** SUBROUTINE GET\_SYSTEM\_FONT@(LFHEIGHT,LFWIDTH,  
+ LFESCAPEMENT,LFORIENTATION,LFWEIGHT,  
+ LFITALIC,LFUNDERLINE,LFSTRIKEOUT,LFCHARSET,  
+ LFOUTPRECISION,LFCLIPPRECISION,LFQUALITY,  
+ LFPITCHANDFAMILY,LPFACENAME)  
INTEGER LFHEIGHT,LFWIDTH, LFESCAPEMENT,  
+ LFORIENTATION,LFWEIGHT,LFITALIC,  
+ LFUNDERLINE,LFSTRIKEOUT,LFCHARSET,  
+ LFOUTPRECISION,LFCLIPPRECISION,LFQUALITY,

```
+ LFPITCHANDFAMILY
CHARACTER(*) LPFACENAME
```

**Description** These attributes may be used to build new fonts by modification, i.e. scaling, italicisation etc.. Modified parameters can then be supplied to the Windows API function **CreateFont**. See the Windows API documentation of **CreateFont** for further details.

## GET\_TEXT\_SIZE@

**Purpose** To get the dimensions of a graphics character string.

**Syntax**

```
SUBROUTINE GET_TEXT_SIZE@(STR, WIDTH, DEPTH)
CHARACTER(*) STR
INTEGER WIDTH,DEPTH
```

**Description** This routine returns the pixel WIDTH and DEPTH of a given character string STR when drawn on the current *drawing surface*. Trailing spaces will be included so either use a substring notation or terminate the significant characters by CHAR(0).

## GET\_TRACK\_LENGTH@

**Purpose** To get the length of a CD track.

**Syntax**

```
INTEGER FUNCTION GET_TRACK_LENGTH@(TRACK)
INTEGER TRACK
```

**Description** Call this routine to obtain the length of an audio CD's track in milliseconds. If the track length returned is zero then this track does not exist. This result can be used to determine the number of audio tracks available. The first track number is 1 rather than zero.

**Return value** Returns the required track length or zero on failure.

**See also** CLOSE\_CD\_TRAY@, SET\_CD\_POSITION@, PLAY\_AUDIO\_CD@, STOP\_AUDIO\_CD@

## GET\_WINDOW\_LOCATION@

**Purpose** To get the location and size of a given window.

**Syntax**

```
SUBROUTINE GET_WINDOW_LOCATION@(HANDLE, X, Y, WIDTH,
+ HEIGHT)
 INTEGER HANDLE, X, Y, WIDTH, HEIGHT
```

**Description** `HANDLE` is a window handle obtained, for example, by using `%hw` or `%lc`. Other arguments are returned with the pixel coordinates and dimensions of the window.

**Notes** For main windows the `X,Y` location is in absolute screen coordinates. For child windows attached by `%aw` to a frame, the coordinates are relative to the top left corner of the frame. For controls and other child (`%ch`) windows `X` and `Y` are relative to the parent window.

This routine (together with the associated move and resize routines) can be used on windows created using `WIN10@` or on other windows if the window handle is available. They are particularly useful to control the placement of MDI child windows.

Moving individual controls and un-framed child windows may be unwise as it can cause controls to overlap on the screen.

Windows will be partially or totally concealed if they are moved outside the area of the screen or the enclosing window or frame (in the case of child windows). This is normal and can be used if desired.

Some interesting effects can be obtained by using these functions. For example, suppose that a child window (which might contain a large graphics region) were attached to a frame that was too small to contain it. By calling `MOVE_WINDOW@` (e.g. in response to a scroll call-back function) with negative values of `X` and/or `Y`, the contents of the window could be scrolled within the frame.

The API function `BringWindowToTop` may also be useful when re-arranging windows.

**See also** `MOVE_WINDOW@`, `RESIZE_WINDOW@`

---

## GET\_WKEY@

**Purpose** To get a key from a *ClearWin* window keyboard buffer.

**Syntax**

```
INTEGER FUNCTION GET_WKEY@()
```

**Description** This routine is only for use in a *ClearWin* window (e.g. with `%cw`). If necessary, it

waits for the user to press a key on the keyboard. However, the routine will yield to other processes.

**Notes** ADD\_KEYBOARD\_MONITOR@ provides a better mechanism for handling keyboard input.

**Return value** Returns the ASCII code for the first key and removes it from the buffer.

**See also** FEED\_WKEYBOARD@, FLUSH\_WKEYBOARD@, GET\_WKEY1@, WKEY\_WAITING@

## GET\_WKEY1@

**Purpose** To get a key from a *ClearWin* window keyboard buffer.

**Syntax** INTEGER FUNCTION GET\_WKEY1@( )

**Description** This routine is only for use in a *ClearWin* window (e.g. with %cw). It is like GET\_WKEY@ but immediately returns zero if the buffer is empty.

**Notes** ADD\_KEYBOARD\_MONITOR@ provides a better mechanism for handling keyboard input.

**Return value** Returns the ASCII code for the first key and removes it from the buffer. If the buffer is empty it returns zero.

**See also** FEED\_WKEYBOARD@, FLUSH\_WKEYBOARD@, GET\_WKEY@, WKEY\_WAITING@

## GRAPHICS\_TO\_CLIPBOARD@

**Purpose** To allow the interchange of graphics between *drawing surfaces* and other Windows applications.

**Syntax** INTEGER FUNCTION GRAPHICS\_TO\_CLIPBOARD@(X1,Y1,X2,Y2)  
INTEGER X1, Y1, X2, Y2

**Description** Places a rectangular area of a *drawing surface* on to the clipboard so that other Windows programs can use the image.

X1, Y1 is the top left corner of the area.

X2, Y2 is the lower right corner of the area.

**Return value** Returns 1 for success and zero on failure.

**See also** COPY\_FROM\_CLIPBOARD@, COPY\_TO\_CLIPBOARD@,

CLIPBOARD\_TO\_SCREEN\_BLOCK@

---

## GRAPHICS\_WRITE\_MODE@

**Purpose** To select replace/XOR mode before writing to the current *drawing surface*.

**Syntax** SUBROUTINE GRAPHICS\_WRITE\_MODE@(MODE)  
INTEGER MODE

**Description** This routine sets the graphics write mode for the current *drawing surface* depending on the value of MODE. Any value other than 3 will force all subsequent graphics output to replace existing pixels. A value of 3 will cause the output to be X0Red with existing pixels.

---

## HIBYTE@

**Purpose** To get the top 8 bits of a 16 bit integer.

**Syntax** INTEGER\*1 FUNCTION HIBYTE@(WORD)  
INTEGER\*2 WORD

**Notes** This is the ClearWin+ Fortran equivalent of the Microsoft C macro with the same name.

---

## HIGH\_COLOUR\_MODE@

**Purpose** To test if the screen is a high colour device.

**Syntax** LOGICAL FUNCTION HIGH\_COLOUR\_MODE@( )

**Description** A high colour device is one that has more than 8 bits per pixel (i.e. more than 256 colours). A high colour device does not make use of colour palettes. If you want to use a wide range of colours whilst avoiding the complexities of managing a palette, one solution is to restrict the usage to desktops operating in high colour mode. Thus you can use this function to test if the desktop is set to operate with more than 256 colours per pixel and to send a error message and close down on failure.

**Return value** Returns .TRUE. for success.

## HIWORD@

**Purpose** To get the top 16 bits of a 32 bit integer.

**Syntax** INTEGER\*2 FUNCTION HIWORD@(DWORD)  
INTEGER DWORD

**Description** Windows API functions sometimes return two values in a long integer with one value in the top 16 bits and one value in the bottom 16 bits. **GetTextExtent** does this for example, returning the height of the text in the top 16 bits and the width in the bottom 16 bits.

**Notes** This is the Fortran ClearWin+ equivalent of the Microsoft C macro with the same name.

---

## IMPORT\_BMP@

**Purpose** To read in a BMP file.

**Syntax** INTEGER FUNCTION IMPORT\_BMP@( FILENAME, ERROR )  
CHARACTER(\*) FILENAME  
INTEGER ERROR

**Description** Reads in a BMP file and returns a DIB handle to the image. The handle can then be used with DIB\_PAINT@, PRINT\_DIB@ and EXPORT\_BMP@. ERROR returns one of the following:

|    |                     |
|----|---------------------|
| 0  | success             |
| 2  | unable to read file |
| 4  | insufficient memory |
| 5  | not a BMP file      |
| 10 | unable to open file |

**Return value** Returns a handle to the image or zero on failure.

**See also** GET\_SCREEN\_BIB@, RELEASE\_SCREEN\_DIB@, IMPORT\_PCX@,  
CLIPBOARD\_TO\_SCREEN\_BLOCK@

## **IMPORT\_PCX@**

**Purpose** To read in a .PCX file.

**Syntax** INTEGER FUNCTION IMPORT\_PCX@( FILENAME, ERROR )  
 CHARACTER\*(\*) FILENAME  
 INTEGER ERROR

**Description** Reads in a PCX file and returns a DIB handle to the image that is compatible with windows bitmaps allowing it be used with DIB\_PAINT@, PRINT\_DIB@ and EXPORT\_BMP@. ERROR returns one of the following:

|    |                     |
|----|---------------------|
| 0  | success             |
| 2  | unable to read file |
| 4  | insufficient memory |
| 6  | not a PCX file      |
| 10 | unable to open file |

**Return value** Returns a handle to the image or zero on failure.

**See also** GET\_SCREEN\_BIB@, RELEASE\_SCREEN\_DIB@, IMPORT\_BMP@,  
 CLIPBOARD\_TO\_SCREEN\_BLOCK@

## **ITALIC\_FONT@**

**Purpose** To set/unset italic font style for text on the current *drawing surface*.

**Syntax** SUBROUTINE ITALIC\_FONT@( ITAL )  
 INTEGER ITAL

**Description** This routine sets or resets the italic property of the current *drawing surface*. Any subsequent calls to the DRAW\_CHARACTERS@ routine will reflect the change.

**See also** ROTATE\_FONT@, SCALE\_FONT@, SELECT\_FONT@, UNDERLINE\_FONT@,  
 BOLD\_FONT@

## **LOBYTE@**

**Purpose** To get the bottom 8 bits of a 16 bit integer.

**Syntax** INTEGER\*1 FUNCTION LOBYTE@(WORD)  
 INTEGER\*2 WORD

**Notes** This is the ClearWin+ Fortran equivalent of the Microsoft C macro with the same name.

## LOWORD@

**Purpose** To get the bottom 16 bits of a 32 bit integer.

**Syntax**

```
INTEGER*2 FUNCTION LOWORD@(DWORD)
 INTEGER DWORD
```

**Notes** This is the Fortran ClearWin+ equivalent of the Microsoft C macro with the same name.

## MAKE\_BITMAP@

**Purpose** To embed a bitmap in program code.

**Syntax**

```
INTEGER FUNCTION MAKE_BITMAP@(BMP_DATA, XRES, YRES)
 CHARACTER*YRES BMP_DATA(XRES)
 INTEGER XRES, YRES
 PARAMETER XRES,YRES
```

**Description** Bitmaps are usually included in a program statically via resources or dynamically with Windows API calls. This routine is provided so that bitmaps may be included in the source code as text data and then translated into bitmap format. A (HBITMAP) handle to a bitmap is returned that can be used for example with %`bm. This provides a speedy method of designing simple graphics that would otherwise require image editing software.

The character arrays may include the following characters:

|           |                     |
|-----------|---------------------|
| - (minus) | Black               |
| b         | Dark blue           |
| r         | Dark red            |
| g         | Dark green          |
| y         | Dark yellow (brown) |
| m         | Dark magenta        |
| c         | Dark cyan           |
| w         | Grey                |
| l         | Light grey          |
| B         | Blue                |
| R         | Red                 |
| G         | Green               |

|   |         |
|---|---------|
| Y | Yellow  |
| M | Magenta |
| C | Cyan    |
| W | White   |

**Return value** Returns a bitmap handle to the new bitmap.

**See also** MAKE\_ICON@

## MAKE\_ICON@

**Purpose** To embed an icon in program code.

**Syntax** INTEGER FUNCTION MAKE\_ICON@( ICON\_DATA )  
CHARACTER\*32 ICON\_DATA(32)

**Description** This routine allows an icon to be embedded in the program code. A 32x32 character array must be defined to accommodate the icon data. The characters can be defined as any of the following:

|           |                     |
|-----------|---------------------|
| - (minus) | Black               |
| b         | Dark blue           |
| r         | Dark red            |
| g         | Dark green          |
| y         | Dark yellow (brown) |
| m         | Dark magenta        |
| c         | Dark cyan           |
| w         | Grey                |
| B         | Blue                |
| R         | Red                 |
| G         | Green               |
| Y         | Yellow              |
| M         | Magenta             |
| C         | Cyan                |
| W         | White               |

If any other character is used then a transparent colour is assumed which will allow the background to show through. A (HICON) handle to the icon is returned which may be used for example with %`ic. The icon is automatically discarded at program termination.

**Return value** Returns a handle to the new icon.

**See also** MAKE\_BITMAP@

**Example**

```

INCLUDE <windows.ins>
CHARACTER*32 icon(32)
INTEGER hicon
hicon=make_icon@(&ICON)
DATA icon(1)/* RRRRRRRRRRRRRRRRRRRRRRRRRRRRR */
DATA icon(2)/* RYYBBYYBBYYBBYYBBYYBBYYBBYYR */
...
...
DATA icon(30)/* RYYBBYYBBYYBBYYBBYYBBYYR */
DATA icon(31)/* RYYBBYYBBYYBBYYBBYYBBYYR */
DATA icon(32)/* RRRRRRRRRRRRRRRRRRRRRRRRRRR */
k=winio@('%ca[Make icon]%',&ICON)
END

```

**MAP\_FILE\_FOR\_READ\_WRITE@****Purpose** To map a file to memory (Win32 only).**Syntax** INTEGER FUNCTION MAP\_FILE\_FOR\_READ\_WRITE@( FILENAME, SIZE )
CHARACTER\*(\*) FILENAME
INTEGER SIZE**Description** See MAP\_FILE\_FOR\_READING@.

In MAP\_FILE\_FOR\_READ\_WRITE@ the variable SIZE can be input as zero, in which case it will be returned as the original size of the file. Otherwise, the mapped memory region will be set to SIZE bytes and the file will be extended if necessary. If the file does not exist it will be created. In this routine the contents of the file can be changed.

**Return value** Returns the starting address of the mapped view or zero on failure.**See also** MAP\_FILE\_FOR\_READING@, UNMAP\_FILE@**MAP\_FILE\_FOR\_READING@****Purpose** To map a file to memory (Win32 only).**Syntax** INTEGER FUNCTION MAP\_FILE\_FOR\_READING@( FILENAME, SIZE )
CHARACTER\*(\*) FILENAME
INTEGER SIZE

**Description** When a file is memory mapped, a region of memory is defined so that accesses to that memory effectively access the disk file. A file may be opened for reading using MAP\_FILE\_FOR\_READING@.

FILENAME is the name of the file to be opened and SIZE is returned as the size of the file.

Use the various Salford CORE intrinsic functions to access the contents of the file. Any attempt to alter the contents of the file will cause an error.

**Return value** Returns the starting address of the mapped view or zero on failure

**Notes** To call a routine with a mapped file as an argument use the form MYSUB(CCORE1(PTR)) where PTR is the value returned by MAP\_FILE\_FOR\_READING@.

**See also** MAP\_FILE\_FOR\_READ\_WRITE@, UNMAP\_FILE@

## METAFILE\_TO\_CLIPBOARD@

**Purpose** To send a metafile to the clipboard.

**Syntax** INTEGER FUNCTION METAFILE\_TO\_CLIPBOARD@( )

**Description** This function can be used after a call to OPEN\_METAFILE@ followed by calls to graphics drawing routines. It causes the image to be stored on the clipboard in metafile format. Alternatively, if the %gr option metafile\_resize is used, this function can be used to send the resulting metafile to the clipboard.

**Return value** Returns 1 for success or zero for failure.

**Notes** If a background colour is specified for %gr (e.g. %gr[black]), this information is not recorded in the metafile.

**See also** OPEN\_METAFILE@, CLOSE\_METAFILE@, PLAY\_CLIPBOARD\_METAFILE@

## MOVE\_WINDOW@

**Purpose** To move a window.

**Syntax** SUBROUTINE MOVE\_WINDOW@( HANDLE, X, Y)  
INTEGER HANDLE, X, Y

**Description** HANDLE is typically a value obtained by using %hw or %lc. X and Y are input as the new location of the window in pixel coordinates.

See GET\_WINDOW\_LOCATION@ and its associated notes.

**See also** RESIZE\_WINDOW@

## MOVIE\_PLAYING@

**Purpose** To test if a video is still playing.

**Syntax** INTEGER FUNCTION MOVIE\_PLAYING@( )

**Return value** MOVIE\_PLAYING@ returns an integer that is the Window handle of the movie window (or zero if no movie is playing). This handle can be passed to the standard API function **DestroyWindow** to force the movie to end.

## NEW\_PAGE@

**Purpose** To provide a new page on the current *drawing surface*.

**Syntax** SUBROUTINE NEW\_PAGE@

**Description** The effect of this routine depends on the type of the current *drawing surface*:

| Device  | Effect                                                                         |
|---------|--------------------------------------------------------------------------------|
| screen  | clears the screen area (identical to CLEAR_SCREEN@)                            |
| printer | prints the image and then blanks the bitmap (an alternative to CLOSE_PRINTER@) |

## OPEN\_CD\_TRAY@

**Purpose** To open a CD-drive tray.

**Syntax** SUBROUTINE OPEN\_CD\_TRAY@( )

**Description** Simply by calling this routine any closed CD drive with a mechanical drawer will open.

**See also** CLOSE\_CD\_TRAY@, SET\_CD\_POSITION@, PLAY\_AUDIO\_CD@, STOP\_AUDIO\_CD@

## **OPEN\_GL\_PRINTER@**

**Purpose** To display the Open Printer dialog box and attach it to OpenGL.

**Syntax** INTEGER FUNCTION OPEN\_GL\_PRINTER@(ID,ATTRIB,WIDTH,HEIGHT)  
INTEGER ID,ATTRIB,WIDTH,HEIGHT

**Description** ID is the user assigned printer identification number.

ATTRIB is a variable that describes the buffering that is required and is a combination of the following flags.

GL\_DOUBLE\_BIT  
GL\_STENCIL\_BIT  
GL\_ACCUM\_BIT  
GL\_DEPTH16\_BIT  
GL\_DEPTH32\_BIT

These flags may be added together.

When the function returns, WIDTH and HEIGHT provide the width and height of the printer page in pixels.

Note that ID and ATTRIB are input values whilst WIDTH and HEIGHT are output values.

**Return value** This function returns 1 if the user selected a device or zero if the “CANCEL” button was selected.

**See also** OPEN\_GL\_PRINTER1@, PRINT\_OPENGL\_IMAGE@

## **OPEN\_GL\_PRINTER1@**

**Purpose** To attach a printer to OpenGL.

**Syntax** INTEGER FUNCTION OPEN\_GL\_PRINTER1@(ID,ATTRIB,WIDTH,HEIGHT)  
INTEGER ID,ATTRIB,WIDTH,HEIGHT

**Description** This function is similar to OPEN\_GL\_PRINTER@ but it does not display the Open Printer dialog box. Instead it uses either the default printer or the printer selected in an earlier call to OPEN\_GL\_PRINTER@.

ID is the user assigned printer identification number.

ATTRIB is a variable that describes the buffering that is required and is a combination of the following flags.

GL\_DOUBLE\_BIT

```
GL_STENCIL_BIT
GL_ACCUM_BIT
GL_DEPTH16_BIT
GL_DEPTH32_BIT
```

These flags may be added together.

When the function returns, `WIDTH` and `HEIGHT` provide the width and height of the printer page in pixels.

Note that `ID` and `ATTRIB` are input values whilst `WIDTH` and `HEIGHT` are output values.

**Return value** This function returns 1 for success, otherwise zero.

**See also** `OPEN_GL_PRINTER@`, `PRINT_OPENGL_IMAGE@`

## **OPEN\_METAFILE@**

**Purpose** To record graphics sequences.

**Syntax** `INTEGER FUNCTION OPEN_METAFILE@( H )`  
`INTEGER H`

**Description** Opens a metafile for the current graphics area. A metafile is used to store graphics information using a device independent format that can be printed, replayed or copied to the clipboard.

**Return value** Returns 1 for success or zero on failure.

**See also** `CLOSE_METAFILE@`, `METAFILE_TO_CLIPBOARD@`,  
`PLAY_CLIPBOARD_METAFILE@`, `DO_COPIES`, `PRINT_GRAPHICS_PAGE@`

## **OPEN\_PRINTER@**

**Purpose** To begin output to a graphics printer or plotter.

**Syntax** `INTEGER FUNCTION OPEN_PRINTER@( HANDLE )`  
`INTEGER HANDLE`

**Description** `OPEN_PRINTER@` generates a standard “Open Printer” dialog box from which the user can select a graphics printer or plotter device for subsequent output. If a device is successfully selected then subsequent calls to Salford graphics routines are be written

to this device. The printer or plotter is activated when CLOSE\_PRINTER@ (see page 345) or the Salford graphics routine NEW\_PAGE@ is called (after calling this routine, it may be necessary to reset the text attributes).

The handle is supplied by the programmer and is used in conjunction with SELECT\_GRAPHICS\_OBJECT@.

**Return value** This function returns 1 if the user selected a device or zero if the “CANCEL” button was selected.

**Notes** This routine can be used in conjunction with the %gr format code that is used to draw to the screen using Salford graphics routines (see page 136). It can also be used independently of WINIO@.

The standard call-back function GPRINTER\_OPEN can also be used to produce graphics output.

**See also** CLOSE\_PRINTER@, DO\_COPIES@, SELECT\_PRINTER@, OPEN\_PRINTER1@

## OPEN\_PRINTER\_TO\_FILE@

**Purpose** To cause the Windows printer device driver to begin graphics output to a file instead of a physical printer.

**Syntax**

```
INTEGER FUNCTION OPEN_PRINTER_TO_FILE@(HANDLE, FILENAME)
CHARACTER(*) FILENAME
INTEGER HANDLE
```

**Description** This function is similar to OPEN\_PRINTER@ but the output is sent to the specified file.

**Return value** Returns 1 if the user selected a device or zero if the “CANCEL” button was selected.

## OPEN\_PRINTER1@

**Purpose** To begin output to a graphics printer or plotter.

**Syntax**

```
INTEGER FUNCTION OPEN_PRINTER1@(HANDLE)
INTEGER HANDLE
```

**Description** OPEN\_PRINTER1@ does not generate a standard “Open Printer” dialog box which means that printing can resume to the current printer after calls to OPEN\_PRINTER@ and CLOSE\_PRINTER@. This avoids the need to repeatedly select the same printer.

Subsequent calls to Salford graphics routines are written to this device. The printer or plotter is activated when CLOSE\_PRINTER@ (see page 345) or the routine NEW\_PAGE@ is called (after calling this routine, it may be necessary to reset the text attributes).

The handle is supplied by the programmer and is used in conjunction with SELECT\_GRAPHICS\_OBJECT@.

**Return value** This function returns 1 for success, otherwise zero.

**Notes** This routine can be used in conjunction with the %gr format code that is used to draw to the screen using Salford graphics routines (see page 136). It can also be used independently of WINIO@.

The standard call-back function GPRINTER\_OPEN can also be used to produce graphics output.

**See also** CLOSE\_PRINTER@, DO\_COPIES@, SELECT\_PRINTER@

## OPEN\_PRINTER1\_TO\_FILE@

**Purpose** To cause the Windows printer device driver to begin graphics output to a file instead of a physical printer.

**Syntax**

```
INTEGER FUNCTION OPEN_PRINTER1_TO_FILE@(HANDLE, FILENAME)
CHARACTER*(*) FILENAME
INTEGER HANDLE
```

**Description** This function is similar to OPEN\_PRINTER1@ but the output is sent to the specified file.

**Return value** Returns 1 for success, otherwise zero.

## OPEN\_TO\_WINDOW@

**Purpose** To attach a *ClearWin* window to a Fortran unit.

**Syntax**

```
SUBROUTINE OPEN_TO_WINDOW@(UNIT,W)
INTEGER UNIT,W
```

**Description** Attaches a previously created *ClearWin* window with handle W to Fortran unit UNIT. This routine enables Fortran I/O requests (for example READ, WRITE and PRINT) to be directed to the specified window.

## OPEN\_WAV\_FILE\_READ@

**Purpose** To open a WAV file for reading in sections.

**Syntax**

```
INTEGER FUNCTION OPEN_WAV_FILE_READ@(FILE,N_SAMPLES,
+ SAMPLE_RATE)
CHARACTER*(*) FILE
INTEGER N_SAMPLES,SAMPLE_RATE
```

**Description** This routine opens a simple (uncompressed) .WAV file so that its contents can be read with a series of calls to WAV\_FILE\_READ@. The number of samples and the sample rate are returned by the function.

**Return value** The return value is the handle to be passed to subsequent sound function calls.

---

## OPEN\_WAV\_FILE\_WRITE@

**Purpose** To open a WAV file for reading in sections.

**Syntax**

```
INTEGER FUNCTION OPEN_WAV_FILE_READ@(FILE,N_CHANNELS,
+ SAMPLE_RATE)
CHARACTER*(*) FILE
INTEGER N_CHANNELS,SAMPLE_RATE
```

**Description** This routine clears and opens a .WAV file to receive sound output as a series of calls to WAV\_FILE\_WRITE@. The number of channels and the sample rate must be supplied to this function. The file will contain sound stored at 16 bits per sample.

Unlike WRITE\_WAV\_FILE@, files written in this way can be written in sections, and may therefore be of effectively unlimited size.

**Return value** The return value is the handle to be passed to subsequent sound function calls.

---

## PERFORM\_GRAPHICS\_UPDATE@

**Purpose** To refresh the current *drawing surface* display.

**Syntax**

```
SUBROUTINE PERFORM_GRAPHICS_UPDATE@()
```

**Description** The current *drawing surface*, defined by %gr, will be refreshed with every call to this routine. Under normal operations a call to a graphics function will not produce an

immediate update, rather ClearWin+ will wait until you have finished your current sequence of graphics calls from within your call-back. This is not usually a problem as it ensures that only the final results are displayed.

**See also** [SELECT\\_GRAPHICS\\_OBJECT@](#)

---

## PERMIT\_ANOTHER\_CALLBACK@

**Purpose** To allow another call-back function to be called.

**Syntax** SUBROUTINE PERMIT\_ANOTHER\_CALLBACK@( )

**Description** Usually if a window invokes a call-back function (in response to a menu, say) most of the standard controls are inhibited from raising further call-backs. This action is normally desirable as it prevents unfortunate complications if the user's mouse 'bounces' on a button.

The subroutine PERMIT\_ANOTHER\_CALLBACK@ allows a call-back function to permit an additional call-back. This routine will only permit one additional call-back, but it can be called as many times as desired, each time enabling a further call-back. When using this routine it is assumed that a call will also be made to a routine such as TEMPORARY\_YIELD@ to permit the processing of further messages.

**Notes** Many programs that seem to require this facility may be presenting the user with a non-standard interface and should be re-structured. For example, suppose a window offers a menu item which will start a lengthy calculation which the user may wish to abort. The usual way to handle this is to display a small additional window with a Cancel button and possibly a progress bar. Since this additional window is distinct from the window which contained the original menu item, there is no need to call the above routine.

---

## PLAY\_AUDIO\_CD@

**Purpose** To play the audio CD from the current position.

**Syntax** INTEGER FUNCTION PLAY\_AUDIO\_CD( DURATION )  
INTEGER DURATION

**Description** Plays an audio CD starting at the current location which can be set with a call to SET\_CD\_POSITION@. If duration is any valid number greater than 0 it will play for that amount of time in milliseconds. If the duration is less than 0 it will play that number of tracks i.e. a value of -4 will play the next four tracks.

**Return value** Always returns 1.

**See also** CLOSE\_CD\_TRAY@, OPEN\_CD\_TRAY@, SET\_CD\_POSITION@, STOP\_AUDIO\_CD@, GET\_TRACK\_LENGTH@

**Example**

```
call set_cd_position@(1, 0)
call play_audio_cd@(-1)
```

## PLAY\_CLIPBOARD\_METAFILE@

**Purpose** To play a clipboard metafile on the current *drawing surface*.

**Syntax** INTEGER PLAY\_CLIPBOARD\_METAFILE@( COL )
  
INTEGER COL

**Description** This can be used after a call to METAFILE\_TO\_CLIPBOARD@. COL is used to set the background colour using the current *colour mode*.

**Return value** The value 1 is returned for success and zero for failure.

**See also** OPEN\_METAFILE@, CLOSE\_METAFILE@

## PLAY\_SOUND@

**Purpose** To send a sample to the audio output device.

**Syntax** INTEGER FUNCTION PLAY\_SOUND@( LEFT, RIGHT, SAMPLES )
  
INTEGER\*2 LEFT(SAMPLES), RIGHT(SAMPLES)
  
INTEGER SAMPLES
  
PARAMETER SAMPLES

**Description** LEFT and RIGHT are arrays that contain the 16 bit sample data. The number of samples is SAMPLES. You should check the values returned by SOUND\_PLAYING@ and SOUND\_RECORDING@ so that a call is not made to this routine until the sound device is idle.

**Return value** Returns 1 for success and a zero on failure

**See also** WRITE\_WAVE\_FILE@, PLAY\_SOUND\_RESOURCE@, RECORD\_SOUND@, SOUND\_PLAYING@, SOUND\_RECORDING@, SELECT\_SAMPLING\_RATE@

---

## PLAY\_SOUND\_RESOURCE@

**Purpose** To send a resource SOUND to the current audio device.

**Syntax** INTEGER FUNCTION PLAY\_SOUND\_RESOURCE@( NAME )  
CHARACTER\*80 NAME

**Description** This routine plays any resource that is of type SOUND. Care should be taken not to include samples (.WAV files) that are more than a few seconds for two reasons. 1) the sample occupies memory and 2) the routine is synchronous and will halt all other output until the sample has completed playback. It is necessary to supply a valid resource name to this routine.

For example:

Include in the resource section of your program:

```
MYWAVE SOUND "dogbark.wav"
```

In the code section:

```
res=PLAY_SOUND_RESOURCE@('MYWAVE')
```

**Return value** Returns 0 for success, otherwise -1.

**See also** WRITE\_WAVE\_FILE@, PLAY\_SOUND@, RECORD\_SOUND@, SOUND\_PLAYING@,  
SOUND\_RECORDING@, SELECT\_SAMPLING\_RATE@

---

## PRINT\_DIB@

**Purpose** To print a device independent bitmap.

**Syntax** INTEGER FUNCTION PRINT\_DIB@( ID, PR\_X, PR\_Y,  
+ PR\_WIDTH, PR\_HEIGHT, HDIB, X, Y, WIDTH, HEIGHT )  
INTEGER ID, PR\_X, PR\_Y,  
+ PR\_WIDTH, PR\_HEIGHT, HDIB, X, Y, WIDTH, HEIGHT

**Description** ID is the handle of the printer as supplied to OPEN\_PRINTER@.

PR\_X, PR\_Y is the printer offset position to start printing (using the printer resolution).

PR\_WIDTH, PR\_HEIGHT represent the size of the area to print (using the printer resolution).

HDIB is the handle of the DIB returned by IMPORT\_BMP@, IMPORT\_PCX@, GET\_SCREEN\_DIB@ or CLIPBOARD\_TO\_SCREEN\_BLOCK@.

X, Y is the offset position from which to start reading the bitmap

WIDTH, HEIGHT represent the size of the bitmap area to print.

**Return value** Returns 1 for success or zero for failure.

**See also** DIB\_PAINT@, RELEASE\_SCREEN\_DIB@, EXPORT\_BMP@

---

## PRINT\_GRAPHICS\_PAGE@

**Purpose** To print a metafile graphics page.

**Syntax** SUBROUTINE PRINT\_GRAPHICS\_PAGE@

**Description** The current metafile image is printed to the graphics printer destination. The page is not cleared and can still be drawn to.

**See also** OPEN\_METAFILE@, DO\_COPIES@

---

## PRINT\_OPENGL\_IMAGE@

**Purpose** To print an OpenGL image.

**Syntax** INTEGER PRINT\_OPENGL\_IMAGE@(ID, LEAVE\_OPEN)  
INTEGER ID  
LOGICAL LEAVE\_OPEN

**Description** ID is the user-defined identification number previously used with OPEN\_GL\_PRINTER@ or OPEN\_GL\_PRINTER1@.

LEAVE\_OPEN = .TRUE. specifies that the current print job is to be left open as there are more drawings to be sent. LEAVE\_OPEN = .FALSE. specifies that the printer connection with this ID is to be closed.

**See also** OPEN\_GL\_PRINTER@, OPEN\_GL\_PRINTER1@

---

## PRINTER\_DIALOG\_OPTIONS@

**Purpose** To configure the standard printer dialog box.

**Syntax**

```

SUBROUTINE PRINTER_DIALOG_OPTIONS@(
 + NO_PRINT_TO_FILE, NO_PAGE_NUMBERS, NO_SELECTION,
 + NO_DEFAULT_WARNING, SET_SOME_PAGES, SET_SELECTION,
 + SET_COLLATE, SET_ALL_PAGES, SET_PRINT_TO_FILE)
 INTEGER
 + NO_PRINT_TO_FILE, NO_PAGE_NUMBERS, NO_SELECTION,
 + NO_DEFAULT_WARNING, SET_SOME_PAGES, SET_SELECTION,
 + SET_COLLATE, SET_ALL_PAGES, SET_PRINT_TO_FILE

```

**Description** The following table describes the effect of setting an argument value to 1 rather than zero.

|                    |                                              |
|--------------------|----------------------------------------------|
| NO_PRINT_TO_FILE   | Hides the ‘Print to file’ check box.         |
| NO_PAGE_NUMBERS    | Disables the ‘Page numbers’ radio button.    |
| NO_SELECTION       | Disables the ‘Print selection’ radio button. |
| NO_DEFAULT_WARNING | Disables ‘No default printer’ warning.       |
| SET_SOME_PAGES     | ‘Page rage’ radio button is set.             |
| SET_SELECTION      | ‘Selection’ radio button is set.             |
| SET_COLLATE        | ‘Collate copies’ check box is checked.       |
| SET_ALL_PAGES      | ‘Print all pages’ radio button is set.       |
| SET_PRINT_TO_FILE  | ‘Print to file’ check box is checked.        |

The following CLEARWIN\_INFO@ strings hold some of the selected options after the printer dialog box has closed.

|                       |                                     |
|-----------------------|-------------------------------------|
| ‘PRINTER_SELECTION’   | ‘Print selection’ was set.          |
| ‘PRINTER_COLLATE’     | ‘Print collate copies’ was checked. |
| ‘PRINTER_PAGENUMBERS’ | ‘Print page numbers’ was set.       |

These values will be either 1 or zero.

**Note** You can deduce the ‘print all pages’ radio button selection by ‘PRINTER\_SELECTION’ and ‘PRINTER\_PAGENUMBERS’ both being equal to zero.

**See also** Selecting a Printer, page 190.

## PUT\_DIB\_BLOCK@

**Purpose** To save a DIB block in a BMP file (24 bits per pixel).

**Syntax**

```
ROUTINE PUT_DIB_BLOCK@(FILENAME, PARRAY, AW, AH, ADX,
+ ADY, W, H, NBPP, ERCODE)
CHARACTER*(*) FILENAME
CHARACTER PARRAY(3,AW,AH)
INTEGER AW,AH,ADX,ADY,W,H,NBPP,ERCODE
```

**Description** All of the arguments are input values except for ERCODE which returns the error status.

FILENAME is the name of the BMP file.

AW and AH are dimensions of PARRAY.

ADX and ADY are offsets used when reading the data from PARRAY.

W and H are the width and height of the image data to be copied.

PARRAY is the same as in GET\_DIB\_BLOCK@.

NBPP is currently not used but should be set to 24 because currently this routine always writes out BMP files with 24 bits per pixel.

ERROR will return:

|    |                    |
|----|--------------------|
| 0  | SUCCESS            |
| 1  | ERROR ON FILE OPEN |
| 3  | ERROR ON WRITE     |
| 13 | BAD NUMBER         |

**See also** GET\_DIB\_SIZE@, GET\_DIB\_BLOCK@, DIB\_BLOCK\_PAINT@

**Example** See page 145.

## RECORD\_SOUND@

**Purpose** To record sound from the sound input device.

**Syntax**

```
INTEGER RECORD_SOUND@(LEFT, RIGHT, SAMPLES, FLAG)
INTEGER*2 LEFT(SAMPLES), RIGHT(SAMPLES)
INTEGER SAMPLES, FLAG
PARAMETER SAMPLES
```

**Description** LEFT and RIGHT are arrays of INTEGER\*2 16 bit data . They will be filled with sound data so enough memory should be allocated to them which should be the same value as

SAMPLES. FLAG is a return value that is currently unused. You should check the SOUND\_PLAYING@ and SOUND\_RECORDING@ results so that a call is not made to this routine until the sound device is idle.

**Return value** Returns 1 for success or zero on failure.

**See also** WRITE\_WAVE\_FILE@, PLAY\_SOUND@, SOUND\_PLAYING@, SOUND\_RECORDING@, SELECT\_SAMPLING\_RATE@, PLAY\_SOUND\_RESOURCE@

## RELEASE\_BITMAP\_DC@

**Purpose** To release a bitmap device context acquired by GET\_BITMAP\_DC@.

**Syntax** SUBROUTINE RELEASE\_BITMAP\_DC@(HDC)  
INTEGER HDC

**Description** The device context and its associated bitmap are released. This function must not be called whilst a window that uses the device context is still active. HDC is the value returned by a previous call to GET\_BITMAP\_DC@.

**See also** GET\_BITMAP\_DC@, CLEAR\_BITMAP@, %dw

## RELEASE\_SCREEN\_DIB@

**Purpose** To release the memory for a DIB.

**Syntax** SUBROUTINE RELEASE\_SCREEN\_DIB@( HDIB )  
INTEGER HDIB

**Description** HDIB is input as the handle of a device independent bitmap created for example by GET\_SCREEN\_DIB@., IMPORT\_BMP@, or CLIPBOARD\_TO\_SCREEN\_BLOCK@.

## REMOVE\_ACCELERATOR@

**Purpose** To remove an accelerator key created using %ac or ADD\_ACCELERATOR@.

**Syntax** SUBROUTINE REMOVE\_ACCELERATOR@( W, KEY\_NAME )  
INTEGER W  
CHARACTER\*(\*) KEY\_NAME

**Description** Removes the key with name KEY\_NAME from the window with handle W. See ADD\_ACCELERATOR@ for further details.

**See also** %ac, ADD\_ACCELERATOR@

---

## REMOVE\_CURSOR\_MONITOR@

**Purpose** To release a call-back function that monitors which window or control has the cursor.

**Syntax** SUBROUTINE REMOVE\_CURSOR\_MONITOR@( CB\_FUNC )  
EXTERNAL CB\_FUNC

**Description** See ADD\_CURSOR\_MONITOR@.

---

## REMOVE\_FOCUS\_MONITOR@

**Purpose** To release a call-back function that monitors which window or control has the focus.

**Syntax** SUBROUTINE REMOVE\_FOCUS\_MONITOR@( CB\_FUNC )  
EXTERNAL CB\_FUNC

**Description** See ADD\_FOCUS\_MONITOR@.

---

## REMOVE\_GRAPHICS\_ICON@

**Purpose** To remove an icon from a %gr *drawing surface*.

**Syntax** SUBROUTINE REMOVE\_GRAPHICS\_ICON( HANDLE )  
INTEGER HANDLE

**Description** HANDLE is the value returned by an earlier call to ADD\_GRAPHICS\_ICON@.

**See also** ADD\_GRAPHICS\_ICON@

## REMOVE\_KEYBOARD\_MONITOR@

**Purpose** To release a call-back function that monitors the use of the keyboard.

**Syntax** SUBROUTINE REMOVE\_KEYBOARD\_MONITOR@( CB\_FUNC )  
EXTERNAL CB\_FUNC

**Description** See ADD\_KEYBOARD\_MONITOR@.

---

## REMOVE\_MENU\_ITEM@

**Purpose** Removes a dynamically attached %mn menu item.

**Syntax** SUBROUTINE REMOVE\_MENU\_ITEM@( HANDLE )  
INTEGER HANDLE

**Description** Use this routine to remove the last menu item that was previously been attached with a call to ADD\_MENU\_ITEM@ (see page 333).

**See also** ADD\_MENU\_ITEM@

---

## REPLY\_TO\_TEXT\_MESSAGE@

**Purpose** To reply to a message from an application that uses SEND\_TEXT\_MESSAGE@.

**Syntax** SUBROUTINE REPLY\_TO\_TEXT\_MESSAGE@( REPLY )  
CHARACTER\*(\*) REPLY

**Description** This routine can be used within a call-back function associated with %rm. The message and reply can be up to 255 characters long.

**See also** %nc, %rm, SEND\_TEXT\_MESSAGE@

---

## RESIZE\_WINDOW@

**Purpose** To resize a window given its window handle.

**Syntax** SUBROUTINE RESIZE\_WINDOW@( HANDLE, WIDTH, HEIGHT )  
INTEGER HANDLE, WIDTH, HEIGHT

**Description** HANDLE is typically a value obtained by using %hw or %lc. WIDTH and HEIGHT are input as the new size of the window in pixels.

See GET\_WINDOW\_LOCATION@ and its associated notes.

**See also** MOVE\_WINDOW@

## RGB@

**Purpose** To pack 8-bit colour values into a 32-bit integer.

**Syntax**

```
INTEGER FUNCTION RGB@(RED, GREEN, BLUE)
INTEGER RED, GREEN, BLUE
```

**Description** This function is equivalent to the Windows API macro called RGB. It takes colour values in the range 0 to 255 and packs them into the 24 least significant bits of a 32 bit integer. GREEN is shifted left 8 bits and BLUE is shifted left 16 bits. RGB values are used in drawing routines when RGB *colour mode* is adopted.

**Return value** Returns the packed RGB values.

## ROTATE\_FONT@

**Purpose** To rotate the font being used on the current *drawing surface*.

**Syntax**

```
SUBROUTINE ROTATE_FONT@(ROT)
DOUBLE PRECISION ROT
```

**Description** This routine will rotate the selected font about the bottom left-hand corner of the text. The rotation is anti-clockwise and ROT is specified in degrees.

**See also** ITALIC\_FONT@, SCALE\_FONT@, SELECT\_FONT@, UNDERLINE\_FONT@, BOLD\_FONT@

## SCALE\_FONT@

**Purpose** To scale the font being used on the current *drawing surface*.

**Syntax**

```
SUBROUTINE SCALE_FONT@(SIZE)
DOUBLE PRECISION SIZE
```

**Description** Rescales the font being used on the current *drawing surface*.

**See also** ROTATE\_FONT@, SELECT\_FONT@, UNDERLINE\_FONT@, ITALIC\_FONT@, BOLD\_FONT@

## SCROLL\_GRAPHICS@

**Purpose** To scroll an area of the current *drawing surface*.

**Syntax**

```
ROUTINE SCROLL_GRAPHICS@(DX, DY, LEFT, TOP,
+ RIGHT, BOTTOM, SWITCH, COLOUR)
INTEGER DX, DY, LEFT, TOP, RIGHT, BOTTOM, SWITCH,
+ COLOUR
```

**Description** This routine allows you to scroll an area the *drawing surface*. The direction of scrolling is set by the two variables DX and DY. They represent the displacement the image will be copied to. For DX, values less than 0 will move the image left and values greater than 0 will move the image right by the relevant number of pixels. For DY, values less than 0 will move the image up and values greater than 0 will move the image down by the relevant number of pixels.

The LEFT, TOP, RIGHT, BOTTOM arguments define the area to be moved. This can be the whole surface or a sub region within the whole.

SWITCH can either be set to 0 or 1. If it is set to 1 then the area that becomes invalid is set to the COLOUR specified by the final argument (using the current *colour mode*). If it is 0 then the invalid region is left unchanged. You should note that this is a copy routine not a move and the region of the screen that is not copied will not be automatically modified.

**See also** SELECT\_GRAPHICS\_OBJECT@, COPY\_GRAPHICS\_REGION@, CREATE\_GRAPHICS\_REGION@, DELETE\_GRAPHICS\_REGION@

## SEE\_PROPERTYPAGE@

**Purpose** To set the sheet number for %ps.

**Syntax**

```
ROUTINE SEE_PROPERTYPAGE@(SHEET_N)
INTEGER SHEET_N
```

**Description** See page 265.

## **SEE\_TREEVIEW\_SELECTION@**

**Purpose** To ensure that the current %tv item is visible.

**Syntax** SUBROUTINE SEE\_TREEVIEW\_SELECTION@( POINT )  
INTEGER POINT

**Description** This routine ensures that the treeview (defined by %tv page 68) is opened showing the current selection. The argument is the current selection integer (not the array of descriptions). This function may be of most use if called via a call-back attached to %sc, thus providing a once only update.

Note that you should ensure that all the appropriate nodes in the hierarchy are marked as expanded ('E') in order to make the selected item visible.

## **SELECT\_FONT@**

**Purpose** To select a new font for a *drawing surface*.

**Syntax** SUBROUTINE SELECT\_FONT@( FONTNAME )  
CHARACTER\*(\*) FONTNAME

**Description** FONTNAME becomes the font for the current *drawing surface*. The default size is not changed. The font can be set to any currently enabled Windows fonts (see the relevant Windows font manager). Care should be taken when selecting fonts since fonts that are available on one machine may not be available on another.

FONTNAME could be a user-added Windows font or 'SYSTEM FIXED FONT' (this font is used for output to a *ClearWin* window) or 'SYSTEM FONT' (this is the Windows SYSTEM\_FONT parameter or, under Windows 95, the DEFAULT\_GUI\_FONT parameter and is used in some *ClearWin+* controls).

**Notes** The default *Charset* value (see FONT\_METRICS@) for Courier New is ANSI\_CHARSET. By using /OEM after the name of the font ('Courier New/OEM') it is possible to choose OEM\_CHARSET.

**See also** ROTATE\_FONT@, SCALE\_FONT@, ITALIC\_FONT@, UNDERLINE\_FONT@,  
BOLD\_FONT@

## **SELECT\_FONT\_ID@**

**Purpose** To set the font for the current *drawing surface*.

**Syntax** INTEGER FUNCTION SELECT\_FONT\_ID@( ID )  
INTEGER ID

**Description** ID is a value returned by an earlier call to GET\_FONT\_ID@.

**Return value** Returns 1 for success or zero on failure.

**See also** CHOOSE\_FONT@, GET\_FONT\_ID@

## **SELECT\_GRAPHICS\_OBJECT@**

**Purpose** To select the current *drawing surface*.

**Syntax** INTEGER FUNCTION SELECT\_GRAPHICS\_OBJECT@( HANDLE )  
INTEGER HANDLE

**Description** *Drawing surfaces* are created by %gr, OPEN\_PRINTER@. etc and CREATE\_GRAPHICS\_REGION@. HANDLE is a value that you supply to %`gr, or OPEN\_PRINTER@ etc.. ClearWin+ drawing routines are applied to the current *drawing surface*. Use this function if you have created more than one *drawing surface* and you want to draw on a surface that is not the current one.

**Return value** The value 1 is returned for success or zero for failure (invalid handle).

## **SELECT\_PRINTER@**

**Purpose** To configure and select the active printer.

**Syntax** INTEGER FUNCTION SELECT\_PRINTER@( DEV, PORT )  
CHARACTER\*(\*) DEV, PORT

**Description** A call to SELECT\_PRINTER@ will provide a standard Windows dialog box from which the user is able to select and configure any of the attached printer devices.

On return DEV will contain the name of the printer device that has been selected and PORT will contain the name of the port it is connected to.

A character array of size 20, in each case, should be sufficient to hold the returned

data.

**Return value** Returns 1 for success or zero if the user clicked on the CANCEL button.

**See also** OPEN\_PRINTER@, CLOSE\_PRINTER@

**Example** PORT LPT1: , COM3: , FAX: ...

---

## SEND\_TEXT\_MESSAGE@

**Purpose** To send a message to an application that uses %rm.

**Syntax** INTEGER FUNCTION SEND\_TEXT\_MESSAGE@( CLASS\_NAME,  
+ MESSAGE, REPLY)  
CHARACTER(\*) CLASS\_NAME,MESSAGE,REPLY

**Description** This function sends a text string of up to 255 characters to a window with the given class name. The recipient window will normally use the %nc format to give its class a specific name. The recipient will also use the %rm format (which takes one call-back argument) to supply a call-back function to handle the message. The call-back function can call CLEARWIN\_STRING@('MESSAGE\_TEXT') to obtain the message. To send a reply, the call-back function should call REPLY\_TO\_TEXT\_MESSAGE@. Since the message and reply can be up to 255 characters long, they can hold considerable amounts of information.

- Notes**
- a) Every message requires by its very nature a context switch from the transmitter application to the receiver. Under Win32 this is quite slow (particularly under Windows 95), therefore large amounts of information are best transmitted in a file, sending just a short message to transmit the file name. Bearing in mind that file accesses are normally cached, this method can be quite efficient.
  - b) One use for inter-process communication is particularly common under Win32. Applications written to run in 16-bit Windows can only have one copy executing at once. However, it is possible to start several copies of a Win32 application. Often the user does not really want several independent copies of your application running simultaneously. Suppose, for example that a MDI editor MYEDIT is already running and the user issues the command:

MYEDIT filename

What is probably required is to open the file in the editor that is already running. To achieve this it is first necessary to uniquely identify the main window of your application. This is done using the %nc format to give the window a distinctive class name. The window should also use the %rm format to supply a routine to handle messages. Before the main window is created the program should call

`SEND_TEXT_MESSAGE@` to send the name of the file to another copy of the editor if one is running. If this function returns 1 for success the current program can be terminated in the knowledge that the file has been transmitted to another active version of the application.

**Return value** Returns 1 for success, otherwise zero if no window with the given class name was found.

## **SET\_ALL\_MAX\_LINES@**

**Purpose** To set the maximum number of lines to be stored for all *ClearWin* windows.

**Syntax** `SUBROUTINE SET_ALL_MAX_LINES@(N)`  
`INTEGER N`

**Description** This routine is the same as `SET_MAX_LINES@` but applies to all *ClearWin* windows.

## **SET\_CD\_POSITION@**

**Purpose** To set the position of the CD read head to TRACK + MILLISECONDS (offset).

**Syntax** `INTEGER SET_CD_POSITION@( TRACK, MILLISECONDS )`  
`INTEGER TRACK, MILLISECONDS`

**Description** The CD read head will be moved to the track plus the number of specified milliseconds into that track. The first track number is 1 rather than 0.

**Return value** Returns 1 for success or zero on failure.

**See also** `CLOSE_CD_TRAY@`, `OPEN_CD_TRAY@`, `PLAY_AUDIO_CD@`, `STOP_AUDIO_CD@`, `GET_TRACK_LENGTH`

## **SET\_CLEARWIN\_FLOAT@**

**Purpose** To set or change a floating point value associated with a named string.

**Syntax** `SUBROUTINE SET_CLEARWIN_FLOAT@( STRING, VALUE )`  
`CHARACTER*(*) STRING`  
`DOUBLE PRECISION VALUE`

**Description** This routine will create and assign a value to a private named parameter that can be

retrieved using `CLEARWIN_FLOAT@`. User defined parameters can be very useful to communicate information between modules in a program (especially between DLL's under Win32). This function cannot be used to set system `CLEARWIN_FLOAT@` values.

**See also** `CLEARWIN_FLOAT@`

---

## **SET\_CLEARWIN\_INFO@**

**Purpose** To set or change an integer value associated with a named string.

**Syntax** `SUBROUTINE SET_CLEARWIN_INFO@( STRING, VALUE )`  
`CHARACTER*(*) STRING`  
`INTEGER VALUE ;`

**Description** This function will create and assign a value to a private named parameter that can be retrieved using `CLEARWIN_INFO@`. User defined parameters can be very useful to communicate information between modules in a program (especially between DLL's under Win32). This function cannot be used to set system `CLEARWIN_INFO@` values except for the printer dialog parameters described on page 194.

**See also** `CLEARWIN_INFO@`

---

## **SET\_CLEARWIN\_STRING@**

**Purpose** To set or change a string value associated with a named string.

**Syntax** `SUBROUTINE SET_CLEARWIN_STRING( STRING, VALUE )`  
`CHARACTER*(*) STRING, VALUE`

**Description** This function will create and assign a value to a private named parameter that can be retrieved using `CLEARWIN_STRING@`. User defined parameters can be very useful to communicate information between modules in a program (especially between DLL's under Win32). This function should not be used to set system `CLEARWIN_STRING@` values except for:

`PRINTER_DOCUMENT`

This is the name of the document that is about to be printed. Its default value is “ClearWin+ Output”.

**See also** `CLEARWIN_STRING@`

## SET\_CLEARWIN\_STYLE@

**Purpose** Modifies the default styling controls for *format* (WIN10@) windows.

**Syntax** INTEGER FUNCTION SET\_CLEARWIN\_STYLE@( NEWSTYLE )  
INTEGER NEWSTYLE

**Description** The default style for a *format* window is (WS\_OVERLAPPEDWINDOW + WS\_HSCROLL + WS\_VSCROLL). This can be changed to any valid combination of Windows styles contained in the *windows.ins* file so that any subsequent windows have a new style.

|                      |                     |
|----------------------|---------------------|
| WS_OVERLAPPED        | WS_POPUP            |
| WS_CHILD             | WS_CLIPSIBLINGS     |
| WS_CLIPCHILDREN      | WS_VISIBLE          |
| WS_DISABLED          | WS_MINIMIZE         |
| WS_MAXIMIZE          | WS_CAPTION          |
| WS_BORDER            | WS_DLGFRA ME        |
| WS_VSCROLL           | WS_HSCROLL          |
| WS_SYSMENU           | WS_THICKFRAME       |
| WS_MINIMIZEBOX       | WS_MAXIMIZEBOX      |
| WS_GROUP             | WS_TABSTOP          |
| WS_OVERLAPPEDWINDOW  | WS_POPUPWINDOW      |
| WS_CHILDWINDOW       | WS_EX_DLGMODALFRAME |
| WS_EX_NOPARENTNOTIFY |                     |

**Return value** The previous settings are returned and should be stored if the effect you wish to generate is only temporary.

## SET\_CONTROL\_TEXT\_COLOUR@

**Purpose** To change the text colour of a control after a window/dialog has been created.

**Syntax** SUBROUTINE SET\_CONTROL\_TEXT\_COLOUR@(HANDLE,COLOUR)  
INTEGER HANDLE, COLOUR

**Description** HANDLE is the window handle of the control obtained by using %lc. COLOUR is the RGB colour value (see RGB@).

**Notes** Unless the colour is to be varied you can set the text colour of a control using %tc before the control when the window is created, possibly followed by another %tc to reset it after the control has been specified.

A call to SET\_CONTROL\_TEXT\_COLOUR@ can be made in the callback function for the control in question. This means, for example, that it is possible to construct a control that displays text in red for negative values and black for positive values.

---

## SET\_CONTROL\_VISIBILITY@

**Purpose** To show or hide a control or window.

**Syntax** SUBROUTINE SET\_CONTROL\_VISIBILITY@(HANDLE,SHOW)  
INTEGER HANDLE,SHOW

**Description** HANDLE is the window handle obtained by using %lc or %hw. SHOW is set to 1 to show the window or 0 to hide it. This routine provides the same functionality as the API function **ShowWindow** but also performs certain necessary housekeeping internal to ClearWin+. It is therefore preferred to **ShowWindow**.

---

## SET\_CURSOR\_WAITING@

**Purpose** To produce a temporary hour-glass cursor.

**Syntax** SUBROUTINE SET\_CURSOR\_WAITING@( WAIT )  
INTEGER WAIT

**Description** If WAIT is set to a non-zero value, the routine changes the cursor to an hour-glass with immediate effect. A zero value then causes the previous cursor to be restored, again with immediate effect.

---

## SET\_DEFAULT\_FONT@

**Purpose** To set the default standard I/O font for *ClearWin* windows.

**Syntax** SUBROUTINE SET\_DEFAULT\_FONT@(HFONT)  
INTEGER HFONT

**Description** This routine is not suitable for *format* windows. It sets the default standard I/O font for *ClearWin* windows to HFONT. HFONT must be the handle to an existing font which can for example be obtained by calling the **CreateFont** Windows API function.

**Note** Any font changes will affect only those *ClearWin* windows subsequently created. If you wish to change the default font for all *ClearWin* windows then this can only be

achieved by calling one of the ClearWin+ set-font functions before any windows are created or any standard I/O is performed.

---

## SET\_DEFAULT\_PROPORIONAL\_FONT@

**Purpose** To set the default standard I/O font for *ClearWin* windows to “System”.

**Syntax** SUBROUTINE SET\_DEFAULT\_PROPORIONAL\_FONT@()

**Description** This routine is not suitable for *format* windows. The font is proportionally spaced. You can revert to a mono spaced font by using SET\_DEFAULT\_TO\_FIXED\_FONT@.

---

## SET\_DEFAULT\_TO\_FIXED\_FONT@

**Purpose** To reset the default standard I/O font for *ClearWin* windows to “System Fixed”.

**Syntax** SUBROUTINE SET\_DEFAULT\_TO\_FIXED\_FONT@()

**Description** This routine is not suitable for *format* windows. The font is mono-spaced.

---

## SET\_DEFAULT\_WINDOW@

**Purpose** To set the default *ClearWin* window for standard I/O.

**Syntax** INTEGER FUNCTION SET\_DEFAULT\_WINDOW@(W)  
INTEGER W

**Description** All future standard I/O requests will be directed to this window. W is a handle created by CREATE\_WINDOW@.

**Return value** Returns the handle of the previous default window.

---

## SET\_GRAPHICS\_SELECTION@

**Purpose** To select a ‘rubber-band’ mode for a %gr region.

**Syntax** SUBROUTINE SET\_GRAPHICS\_SELECTION@( MODE )  
INTEGER MODE

**Description** MODE can be any of the following:

- |   |              |
|---|--------------|
| 0 | No selection |
| 1 | Rectangle    |
| 2 | Line         |

This routine is used when mouse input is processed within a %gr *drawing surface*. It remains local to the current *drawing surface* selected with SELECT\_GRAPHICS\_OBJECT@.

Mode 1 makes a box appear whenever the left mouse button is pressed. The corner of the box will stay anchored to that point until the mouse button is released. The other corner will follow the mouse cursor around the *drawing surface* until the mouse button is released. By processing the mouse state and the released information it is possible to determine the co-ordinates of the selected region.

Mode 2 is similar to mode 1 but a straight line joins the first point to the second whilst the left mouse button is held down.

Mode 0 deactivates the box and line high-lighting mechanism. All the lines are drawn on to and removed from the display with an XOR write which prevents the destruction of any underlying graphics.

It is important to note that you should only call this routine once for each line mode change required.

**See also** GET\_GRAPHICS\_SELECTED\_AREA@

**Example** See page 143.

## SET\_HIGHLIGHTED@

**Purpose** To select all of the text in a %rd, %rf or %rs edit box.

**Syntax** SUBROUTINE SET\_HIGHLIGHTED@(HANDLE)  
INTEGER HANDLE

**Description** HANDLE is the window handle of the edit box obtained by using %lc.

**See also** %rd, %rf, %rs

---

## SET\_LINE\_STYLE@

**Purpose** To set the style for a line on the current *drawing surface*.

**Syntax** SUBROUTINE SET\_LINE\_STYLE@( VALUE )  
INTEGER VALUE

**Description** This routine changes the line style on the current *drawing surface*. VALUE is one of the following predefined Windows constants (available from the file *windows.ins*):

---

|                |                                                                                                                                      |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------|
| PS_SOLID       | solid line                                                                                                                           |
| PS_DASH        | dashed line                                                                                                                          |
| PS_DOT         | dotted line                                                                                                                          |
| PS_DASHDOT     | alternating dashes and dots                                                                                                          |
| PS_DASHDOTDOT  | alternating dashes and double dots                                                                                                   |
| PS_NULL        | pen is invisible                                                                                                                     |
| PS_INSIDEFRAME | the dimensions of a figure are reduced so that it fits entirely in the bounding rectangle, taking into account the width of the pen. |

---

**Notes** PS\_INSIDEFRAME can be ORed with one of the other values.

PS\_SOLID, PS\_NULL and PS\_INSIDEFRAME are only available with a line width greater than 1.

PS\_NULL can be used to remove the black border when printing a ‘filled polygon’, for example, on a black and white printer.

**See also** SET\_LINE\_WIDTH@

---

## SET\_LINE\_WIDTH@

**Purpose** To set the line width on the current *drawing surface*.

**Syntax** SUBROUTINE SET\_LINE\_WIDTH@( VALUE )  
INTEGER VALUE

**Description** Changes the pixel thickness of a line when drawing on a *drawing surface*. The default value is 1.

**See also** SET\_LINE\_STYLE@

---

## SET\_MAX\_LINES@

**Purpose** To set the maximum number of lines to be stored for a given *ClearWin* window.

**Syntax** SUBROUTINE SET\_MAX\_LINES@(W,N)  
INTEGER W,N

**Description** Sets a maximum to the number of lines of output ( $N > 50$ ) that is to be stored by the *ClearWin* window with handle  $W$ . Before storing line  $N+1$ , line 1 is deleted etc..

---

## SET\_MOUSE\_CURSOR\_POSITION@

**Purpose** To set the position of the mouse with respect to a given window.

**Syntax** SUBROUTINE SET\_MOUSE\_CURSOR\_POSITION@( W, X, Y )  
INTEGER W, X, Y

**Description**  $W$  is the handle of window,  $X$  is the x position relative to the window, and  $Y$  is the y position relative to the window. If  $W$  is zero, the positioning is relative to the top left-hand corner of the screen itself.

---

## SET\_OLD\_RESIZE\_MECHANISM@

**Purpose** To revert to the old *ClearWin+* resizing strategy.

**Syntax** SUBROUTINE SET\_OLD\_RESIZING\_MECHANISM@( )

**Description** In the original *ClearWin+* specification you could create windows which would re-size without any control changing its dimensions. This happened if you used %ww but either did not use a pivot (%pv), or applied a pivot to a control which could not use it.

To obtain the old specification you should call SET\_OLD\_RESIZE\_MECHANISM@. The new scheme is described on page 94.

## SET\_OPEN\_DIALOG\_PATH@

**Purpose** To set the initial directory before displaying the standard file open dialog box.

**Syntax** SUBROUTINE SET\_OPEN\_DIALOG\_PATH@( PATH )

**See also** %fs, %ft, GET\_FILTERED\_FILE@

---

## SET\_PRINTER\_ORIENTATION@

**Purpose** To set the printer orientation to portrait or landscape.

**Syntax** SUBROUTINE SET\_PRINTER\_ORIENTATION@( V )  
INTEGER V

**Description** This routine can be called before OPEN\_PRINTER@ and OPEN\_PRINTER1@ or before using one of the standard call-back functions 'PRINTER\_SELECT', 'PRINTER\_OPEN', or 'PRINTER\_OPEN1'.

Set V = 0 for portrait and V = 1 for landscape.

---

## SET\_RGB\_COLOURS\_DEFAULT@

**Purpose** To set the global default to RGB colour mode.

**Syntax** SUBROUTINE SET\_RGB\_COLOURS\_DEFAULT@( BOOL )  
INTEGER BOOL

**Description** Currently by default ClearWin+ operates in VGA *colour mode* in which colours are represented by palette index values. RGB *colour mode* is recommended for new programs and this routine can be used to change the default. USE\_RGB\_COLOURS@ is similar but relates to a given *drawing surface*. See page 136 for further details.

Use an input value of 1 to change the default to RGB *colour mode*.

**See also** USE\_RGB\_COLOURS@, %gr[rgb\_colours]

## SET\_SOUND\_SAMPLE\_RATE@

**Purpose** To set the sampling speed.

**Syntax** SUBROUTINE SET\_SOUND\_SAMPLE\_RATE@( RATE )  
INTEGER RATE

**Description** The sampling rate is the frequency at which the sound sample will be played back or recorded. Do not change this value whilst SOUND\_PLAYING@ or SOUND\_RECORDING@ return a value of 1. The maximum value that is permissible is dependant of the sound device being used and you should consult the relevant technical notes supplied with it. Common Windows values are maximum 44100 (44.1KHz), midrange 22050 (22.05khz) and minimum 11025 (11.025khz) - also the default. It should also be noted that there will be an absolute minimum. There is no sense in recording at anything less than 8000 (Hz), which is approximately the same sample rate as a telephone connection, as the recording quality would be too low for practical use.

When selecting a suitable value it should also be noted that the sample rate must be set to twice that of the maximum frequency you wish to record, this is to preserve quality and reduce alias distortion.

For example:

If the maximum frequency to be recorded is 10khz (10000) then the sample rate must be set to 20Khz (20000).

Note that this will require a large of amount storage.

**See also** WRITE\_WAVE\_FILE@, PLAY\_SOUND@,  
SOUND\_PLAYING@,PLAY\_SOUND\_RESOURCE@, SOUND\_RECORDING@,  
SOUND\_RECORDING@,SOUND\_SAMPLE\_RATE@

---

## SHOW\_MOVIE@

**Purpose** To show a video.

**Syntax** SUBROUTINE SHOW\_MOVIE@(FILE,OPT)  
INTEGER OPT  
CHARACTER\*(\*) FILE

**Description** SHOW\_MOVIE@ opens the given .AVI file and shows it in a window. If OPT is zero, the user must click the play button to start the movie. If OPT is 1, the movie starts immediately. Do not call this routine with a file which does not exist - if necessary check in advance.

When SHOW\_MOVIE@ returns, the movie window will be displayed and will be under the user's control. The MOVIE\_PLAYING@ function can be used to determine when the user finally closes the window. If a second call to SHOW\_MOVIE@ is made before a previous movie has finished playing, Clearwin+ will wait until the first movie completes.

Movie files can include a sound track, which will be played if a sound card is installed in your PC.

---

## **SIZE\_IN\_PIXELS@**

**Purpose** To set the font size in pixels.

**Syntax** SUBROUTINE SIZE\_IN\_PIXELS@( HEIGHT, WIDTH )  
INTEGER HEIGHT, WIDTH

**Description** This routine allows you to specify the height and width of a font in pixels.

**See also** SIZE\_IN\_POINTS@

---

## **SIZE\_IN\_POINTS@**

**Purpose** To set the font size in points.

**Syntax** SUBROUTINE SIZE\_IN\_POINTS@( HEIGHT, WIDTH )  
INTEGER HEIGHT, WIDTH

**Description** Sets the HEIGHT and WIDTH of a font that is being used on a *drawing surface*.

**See also** SIZE\_IN\_PIXELS@, SELECT\_FONT@

---

## **SIZEOF\_CLIPBOARD\_TEXT@**

**Purpose** To get the size of the text held in the clipboard.

**Syntax** INTEGER FUNCTION SIZEOF\_CLIPBOARD\_TEXT@()

**Description** This routine returns the length of the Windows clipboard text array (CF\_TEXT or CF\_OEMTEXT format). The returned length includes a null (CHAR(0)) character that terminates the string.

**Return value** Returns the length of the string or zero on failure.

**See also** COPY\_FROM\_CLIPBOARD@

---

## SOUND\_PLAYING@

**Purpose** To test for ongoing sound output.

**Syntax** INTEGER FUNCTION SOUND\_PLAYING@()

**Description** This function returns 1 while there is sound output and zero when there is none. It is most useful when used to ‘join’ several sound samples together by detecting when one has finished so that the next may be started.

**Return value** Returns 1 if sound is playing otherwise zero.

**See also** WRITE\_WAVE\_FILE@, RECORD\_SOUND@, PLAY\_SOUND@,  
PLAY\_SOUND\_RESOURCE@, SOUND\_RECORDING@, SOUND\_SAMPLE\_RATE@  
SELECT\_SAMPLING\_RATE@

---

## SOUND\_RECORDING@

**Purpose** To test if a recording is being made.

**Syntax** INTEGER FUNCTION SOUND\_RECORDING@()

**Return value** Returns 1 while the sound system is recording and 0 when it is idle.

**See also** WRITE\_WAVE\_FILE@, RECORD\_SOUND@, SOUND\_PLAYING@,  
SOUND\_SAMPLE\_RATE@, PLAY\_SOUND@, SELECT\_SAMPLING\_RATE@,  
PLAY\_SOUND\_RESOURCE@

---

## SOUND\_SAMPLE\_RATE@

**Purpose** To return the current sound sampling rate.

**Syntax** INTEGER FUNCTION SOUND\_SAMPLE\_RATE@()

**Return value** Returns the value supplied to SET\_SOUND\_SAMPLE\_RATE@ or the default sample rate.

**See also** WRITE\_WAVE\_FILE@, RECORD\_SOUND@, PLAY\_SOUND@,

---

PLAY\_SOUND\_RESOURCE@, SOUND\_RECORDING@, SELECT\_SAMPLING\_RATE@

---

## **START\_PROCESS@**

**Purpose** To create a new process and wait for it to terminate.

**Syntax** INTEGER FUNCTION START\_PROCESS@( COMMAND, PARAMS )  
CHARACTER\*(\*) COMMAND,PARAMS

**Description** COMMAND is the command line that is to be executed. PARAMS is a string of parameters that is appended to the command line.

**Return value** Returns 0 or a positive value if successful. A return value of -1 indicates an error condition.

**Notes** The new process is given the same ‘show’ state as that used when creating the calling process. This will usually be ‘show normal’ or ‘show maximised’.

The alternative function START\_PPROCESS@ takes the same form but returns immediately (i.e. it does not wait for the created process to terminate).

**Example**    INTEGER START\_PROCESS@, i  
          i=START\_PROCESS@('run.exe',' ')  
          END

---

## **STOP\_AUDIO\_CD@**

**Purpose** To stop audio playback from the CD.

**Syntax** SUBROUTINE STOP\_AUDIO\_CD@()

**See also** CLOSE\_CD\_TRAY@, PEN\_CD\_TRAY@, PLAY\_AUDIO\_CD@, SET\_CD\_POSITION@

---

## **TEMPORARY\_YIELD@**

**Purpose** Provides an alternative to YIELD\_PROGRAM\_CONTROL@(Y\_TEMPORARILY).

**Syntax** SUBROUTINE TEMPORARY\_YIELD@()

## UNDERLINE\_FONT@

**Purpose** To add an underline to text on the current *drawing surface*.

**Syntax** SUBROUTINE UNDERLINE\_FONT@( ACTIVE )  
INTEGER ACTIVE

**Description** When using the function DRAW\_CHARACTERS@ on a *drawing surface* with any font selected, a call to this routine with a value of 1 will make subsequent text underlined. A further call with a value of 0 will switch off the effect.

**See also** ROTATE\_FONT@, SCALE\_FONT@, SELECT\_FONT@, ITALIC\_FONT@, BOLD\_FONT

---

## UNMAP\_FILE@

**Purpose** To close a file map (Win32 only).

**Syntax** SUBROUTINE UNMAP\_FILE@( PTR )  
CHARACTER\*(\*) PTR

**Description** File maps are closed automatically when the program terminates but can be closed sooner by calling this routine. PTR is a value returned by MAP\_FILE\_FOR\_READING@ or MAP\_FILE\_FOR\_READ\_WRITE@.

**See also** MAP\_FILE\_FOR\_READING@, MAP\_FILE\_FOR\_READ\_WRITE@

---

## UPDATE\_WINDOW@

**Purpose** To redraw a *ClearWin* window.

**Syntax** SUBROUTINE UPDATE\_WINDOW@(W)  
INTEGER W

**Description** Causes the entire client area to be ‘invalidated’ and therefore re drawn. This should not be confused with the *ClearWin+* function WINDOW\_UPDATE@ (see page 26) that is used for *format* windows.

---

## USE\_APPROXIMATE\_COLOURS@

**Purpose** To switch off colour matching for all *drawing surfaces*.

**Syntax**

```
SUBROUTINE USE_APPROXIMATE_COLOURS@(BOOL)
 INTEGER BOOL
```

**Description** By default, routines that draw to a *drawing surface* use a colour matching algorithm that selects an optimum colour match corresponding to the supplied colour value. Colour matching can be switched off (for all drawing surfaces) by calling this routine with the argument set to zero. A non-zero value switches on again.

If automatic colour matching is switched off, a matched colour can still be obtained from GET\_MATCHED\_COLOURS@. The result is used as an RGB value in any of the *drawing surface* routines.

Using this routine may lead to a significant improvement in the performance of your program since colour matching is often unnecessary. When colour matching is desirable, it is only necessary to apply the matching algorithm once for each colour that is used.

---

## USE\_RGB\_COLOURS@

**Purpose** To switch between colour modes on a *drawing surface*.

**Syntax**

```
INTEGER FUNCTION USE_RGB_COLOURS@(HANDLE, BOOL)
 INTEGER HANDLE, BOOL
```

**Description** HANDLE is the programmer's handle for the *drawing surface* as used for example in SELECT\_GRAPHICS\_OBJECT@ and %`gr. If HANDLE is zero then the current *drawing surface* is assumed. BOOL can be either 1 or 0. If it is 1 then the RGB *colour mode* will be used otherwise VGA *colour mode* (see page 136).

**Return value** Returns 1 for success or zero if the handle is not valid.

**See also** SET\_RGB\_COLOURS\_DEFAULT@, %gr[rgb\_colours]

---

## USE\_WINDOWS95\_FONT@

**Purpose** To set the default WINIO@ font to Windows 95 font (Windows 95 only).

**Syntax**

```
SUBROUTINE USE_WINDOWS95_FONT@()
```

**Description** If this routine is called, by default all WINIO@ dialogs will use the Windows 95 font where available. This is equivalent to placing %`sf at the front of each WINIO@ string and adding a grave accent to every existing %sf format.

**See also** %sf

---

## WAV\_FILE\_READ@

**Purpose** To read a WAV file as a samples wave-form.

**Syntax**

```
INTEGER FUNCTION WAV_FILE_READ@(HANDLE, LEFT,
+ RIGHT, SAMPLES)
 INTEGER HANDLE, SAMPLES
 INTEGER*2 LEFT(SAMPLES), RIGHT(SAMPLES)
```

**Description** Samples sound data is read from the WAV file, returning at most SAMPLES values. HANDLE should have been returned by a call to OPEN\_WAV\_FILE\_READ@.

**Return value** Returns the number of samples read, which can be less than SAMPLES if the end of the file is reached.

**See also** OPEN\_WAV\_FILE\_READ@,

---

## WAV\_FILE\_WRITE@

**Purpose** To store any recorded data.

**Syntax**

```
SUBROUTINE WAV_FILE_WRITE@(HANDLE, LEFT, RIGHT, SAMPLES)
 INTEGER HANDLE, SAMPLES
 INTEGER*2 LEFT(SAMPLES), RIGHT(SAMPLES)
```

**Description** The data contained in LEFT and RIGHT is written to disk in WAV file format. HANDLE should have been returned by a call to OPEN\_WAV\_FILE\_WRITE@.

**See also** OPEN\_WAV\_FILE\_WRITE@

## WIN\_COUA@

**Purpose** To output *n* characters from a string to a *ClearWin* window.

**Syntax** SUBROUTINE WIN\_COUA@(W,STR,N)  
INTEGER W,N  
CHARACTER\*(\*) STR

**Description** Outputs N characters from the string STR to the *ClearWin* window with handle W. This function provides an alternative to the standard Fortran routines WRITE and PRINT that are normally used for text output to a *ClearWin* window.

---

## WIN\_GETCHAR@

**Purpose** To read a character from the keyboard.

**Syntax** INTEGER WIN\_GETCHAR@(W)  
INTEGER W

**Description** Changes the input focus to the *ClearWin* window with handle W and reads a character from the keyboard. This function provides an alternative to the standard READ routine.

---

## WIN\_GETLINE@

**Purpose** To read a line of text from the keyboard.

**Syntax** SUBROUTINE WIN\_GETLINE@(W,LINE)  
INTEGER W  
CHARACTER\*(\*) LINE

**Description** Changes the input focus to the *ClearWin* window with handle W and reads a line of text from the keyboard. This function provides an alternative to the standard READ routine.

---

## WINDOWS\_95\_FUNCTIONALITY@

**Purpose** To test if Windows 95 is in use.

**Syntax** INTEGER FUNCTION WINDOWS\_95\_FUNCTIONALITY@()

**Return value** Returns the value 1 if Windows 95, otherwise zero.

---

## WKEY\_WAITING@

**Purpose** To test if there are any keys in a *ClearWin* window keyboard buffer.

**Syntax** INTEGER FUNCTION WKEY\_WAITING@( )

**Description** This routine is only for use in a *ClearWin* window (e.g. with %cw).

**Return value** Returns the number of keys in the keyboard buffer.

**See also** FEED\_WKEYBOARD@, FLUSH\_WKEYBOARD@, GET\_WKEY@, GET\_WKEY1@

---

## WRITE\_GRAPHICS\_TO\_BMP@

**Purpose** To write the current *drawing surface* to a BMP file.

**Syntax** SUBROUTINE WRITE\_GRAPHICS\_TO\_BMP@( FILENAME, ERROR )

CHARACTER\*(\*) FILENAME

INTEGER ERROR

**Description** Writes the current *drawing surface* to the file with name FILENAME. ERROR returns one of the following:

0 success

1 no current *drawing surface* or unable to open file

3 unable to write to file

**See also** IMPORT\_BMP@, EXPORT\_BMP@

---

## WRITE\_GRAPHICS\_TO\_PCX@

**Purpose** To write the current *drawing surface* to a PCX file.

**Syntax** SUBROUTINE WRITE\_GRAPHICS\_TO\_PCX@( FILENAME, ERROR )

CHARACTER\*(\*) FILENAME

INTEGER ERROR

**Description** Writes the current *drawing surface* to the file with name FILENAME. ERROR returns one of the following:

|   |                                                          |
|---|----------------------------------------------------------|
| 0 | success                                                  |
| 1 | no current <i>drawing surface</i> or unable to open file |
| 3 | unable to write to file                                  |

**See also** IMPORT\_PCX@, EXPORT\_PCX@

## WRITE\_WAV\_FILE@

**Purpose** To store any recorded data.

**Syntax**

```
 SUBROUTINE WRITE_WAV_FILE@(FILENAME, CHAN, SAMPLES,
+ LEFT, RIGHT)
 CHARACTER*(*) FILENAME
 INTEGER CHAN, SAMPLES
 INTEGER*2 LEFT(SAMPLES), RIGHT(SAMPLES)
 PARAMETER SAMPLES
```

**Description** The data contained in LEFTand RIGHT is written to disk in wave file format. FILENAME should be set to the desired file name. CHAN can be either 1 (mono) or 2 (stereo). SAMPLES is the length of both the LEFTand RIGHT data arrays (they should be identical).

This routine writes a whole WAV file in one go, so it must be of a size that can fit into memory.

**See also** SOUND\_RECORDING@, RECORD\_SOUND@,  
SOUND\_PLAYING@, PLAY\_SOUND\_RESOURCE@, PLAY\_SOUND@,  
SET\_SOUND\_SAMPLE\_RATE@

## YIELD\_PROGRAM\_CONTROL@

**Purpose** To process Windows messages.

**Syntax**

```
 SUBROUTINE YIELD_PROGRAM_CONTROL@(OPTION)
 INTEGER OPTION
```

**Description** This routine contains a Windows message loop. It is called automatically by ClearWin+ so that the programmer does not need to include a message loop in a program. It can be used within a program to ensure that a process does not totally take over the CPU. OPTION may be either Y\_PERMANENTLY or Y\_TEMPORARILY.

You may need to call this routine using Y\_TEMPORARILY, if the program makes

protracted use of the CPU and you want to allow it to respond to events (such as mouse events) whilst the processing continues. Call this routine at suitable points during the process to enable the message queue for the current application to be processed. (Message queues for other applications that are running will automatically be processed because Windows 95 and Windows NT are pre-emptive multi-tasking operating systems.) `TEMPORARY_YIELD@()` is the same as `YIELD_PROGRAM_CONTROL@(Y_TEMPORARILY)`.

Calling this routine using `Y_PERMANENTLY` is equivalent to reaching the `END` statement of a **ClearWin+** program.



# 28.

# Glossary

## **Accelerator keys**

An accelerator key is a key sequence, such as Alt-C that can directly activate some process in a window (i.e. without popping up a menu). Accelerators can be defined using %ac. A special case of an accelerator key is defined using %es, which causes a window to respond to the ESC key by closing.

## **Bitmap**

A rectangular image stored as binary data (see %bm). Bitmaps are stored in files with the .BMP suffix.

## **Button**

A button is a control that is designed to respond to a mouse click. Several types of buttons are available. Simple 3-dimensional press and release buttons containing text (and sometimes an icon as well) can be created using %bt. Radio buttons are created with %rb. These are point like structures (with the name along side) that toggle when pressed. Buttons containing a picture can be created with %tb.

## **Call-back function**

A ClearWin+ call back function is a function passed to WIN10@ (and certain other functions) that is called when certain events occur. In ClearWin+, all such functions take no arguments and return an integer result.

## **Caption**

The distinctively coloured strip across the top of most windows that contains the title together with some controls for manipulating the window. A window can be dragged by its caption. See %ca.

## **Child window**

A child window is a window that is contained within a larger window (which could be a child of yet another window). The sub-windows of an MDI application (see %fr and %aw) are children, as are the separate sheets of a property sheet (see %ps and %sh).

However, most controls, such as buttons, are considered by Windows to be children of the window that contains them. This is why controls can be referred to by their window handle (see %lc).

### **ClearWin window**

A *ClearWin* window (no '+') refers to a window created to contain output that would have gone to the screen if the program were not a Windows application. The name refers to the fact that the original *ClearWin* only created such relatively primitive windows. A *ClearWin* window is still useful for debugging purposes, and may be properly embedded in a *format* window using %cw. Keyboard input can also be taken from a *ClearWin* window, but this is not recommended in finished programs.

### **Clipboard**

The clipboard is a structure maintained by Windows that can contain information which is to be copied within or between applications. The clipboard 'knows' the format of the data that it contains, so a text editor will not accidentally paste data of another format (e.g. an image) into its contents. By convention, the clipboard is accessed as a direct response to controls with names such as 'paste', 'cut', 'copy'. It is extremely anti-social to use the clipboard in any other way, as users learn to rely on the contents of the clipboard remaining unchanged unless they explicitly alter them.

### **Colour modes**

When drawing to *drawing surfaces*, ClearWin+ operates in one of two *colour modes*; *RGB mode* or *VGA mode*. In *VGA mode*, colour values are represented as indexes into a colour palette by which ClearWin+ emulates a DOS environment. This mode is useful for porting DOS programs to Windows. Currently the default *colour mode* is *VGA mode*. In *RGB mode* ClearWin+ uses RGB colours (the native Windows colour representation). *RGB mode* is recommended for new programs.

### **DLL**

This stands for Dynamic Link Library - a library of code that can be linked to a program at run time. DBOS provides such a mechanism, as does Win32. The Win32 file *salflibc.dll* is an example of a DLL.

### **Drawing surfaces**

ClearWin+ includes a library of drawing routines for drawing lines and text, for filling areas, for copying from one place to another, etc.. These drawing routines are applied to the current *drawing surface*. This surface could be one of a number of rectangular areas of the screen generated by %gr, it could be one of a number of printer bitmaps (created for example by OPEN\_PRINTER@) or it might be an internal bitmap created by CREATE\_GRAPHICS\_REGION@ (that is, a bitmap that is created for copying to another *drawing surface*). SELECT\_GRAPHICS\_OBJECT@ is used to switch from one *drawing surface* to another.

**Edit box**

This term is used to refer to boxes created with %eb to perform text editing. The term may also include other boxes containing editable text, such as numeric input boxes created with %rd, %rf, etc..

**Focus**

A control is said to have focus if it is the one that will respond to keyboard input. Even buttons can have focus - a button with focus can be activated by pressing the space bar. Usually the appearance of a control changes slightly when it acquires focus.

**Font**

Information describing the graphical layout of a set of characters. Windows fonts can be variable pitch (e.g. Times New Roman) or fixed pitch (e.g. Courier New). Variable pitch fonts fit characters into different amounts of horizontal space depending on their shape and the shapes of their neighbours (see %fn). Each font type comes in several sizes (see %ts) and variants - italic (%it), bold (%bf) and underline (%ul).

**Format**

|                 |                                                         |
|-----------------|---------------------------------------------------------|
| format window   | a window that has been created using WIN10@.            |
| format string   | the first argument of WIN10@.                           |
| format code     | a format substring beginning with %.                    |
| format modifier | one of the characters ^ ~ ? appearing in a format code. |

**GIF file**

An image file format popular in World Wide Web documents. GIF files can be partially transparent and/or animated. They can be incorporated into ClearWin+ programs as resources and accessed using the %gi format.

**Greyed out**

This expression refers to the process by which certain menu items, buttons, etc. can be given a washed out appearance which is an indication to the user that they are not available in current circumstances. Many controls and menus can be controllably greyed using the '~' format modifier together with an integer control variable that specifies the current state.

**GUI**

This stands for Graphical User Interface, and refers to a program with a Windows interface (or the equivalent on other platforms).

**Handle**

A handle is an integer supplied by Windows (in the context of Windows programming) used to refer to an object. Thus a window handle is an integer used to refer to a whole window or an individual control (most of which are treated as small child windows). A bitmap handle is an integer used to refer to a bitmap, etc..

### **High colour device**

A high colour graphics device is one that uses more than 8 bits (256 colours) per pixel. Colour palettes become redundant when using such a device. The function `HIGH_COLOUR_MODE@` can be used to test for a high colour device.

### **HTML**

Hypertext Mark-up Language is the language used to create Internet Web pages. A simplified form can be used to create hypertext pages in a format window (see `%ht`).

### **Icon**

An icon is a special sort of image, stored in a binary file with the .ICO suffix. It is partially transparent. Thus an icon can appear to have any shape (see `%ic` and `%mi`). ClearWin+ uses icons of size 32x32. However, larger transparent images can be manipulated as GIF files.

### **Maximise**

The process by which a window is expanded until it exactly fits the screen (or containing window in the case of MDI children). Menu options in the system menu (and buttons on the top right of the window) enable the user to select and de-select this state for a window. `%ww[maximise]` can be used to create a window in this state.

### **MDI**

This stands for Multiple Document Interface. This is a style of program in which one large window contains one or more smaller windows within it. These windows can be opened, closed, moved, etc. independently. Most Windows editors adopt this style with one window per opened file. The `%fr` and `%aw` formats can be used to create such programs under ClearWin+.

### **Menu**

Menus are text structures that respond to mouse clicks (and usually certain Alt-key sequences) to enable users to give commands to the program. There are three types of menus in Windows. 1) Ordinary window menus that start just below a window's caption and drop down to reveal sub options when they are clicked (see `%mn`). 2) The so called system menu, that can be activated by clicking in the top left corner of a window (see `%sm`), and 3) popup menus that appear when the right mouse button is clicked (see `%pm`). See also shortcut keys.

### **Minimise**

The process by which a window is shrunk to an icon (also known as iconification). Menu options in the system menu (and buttons on the top right of the window) enable the user to select this state. The user can access the system menu in order to return the window to normal by clicking on the icon. Use `%mi` to define the icon that will be used in the minimised state.

## Resources

Microsoft jargon for non-program information (e.g. images or sounds) built into a program. Resources are specified in resource scripts, and are compiled with the resource compiler (SRC). ClearWin+ programs use resources to incorporate images, sounds, and hypertext into the program.

## RGB colours

The Windows API represents a colour in terms of the intensity (in the range 0 to 255) of its red, green and blue components. Each component is thus an 8-bit value and the three components are packed into a 32-bit integer with the least significant 8 bits being red, the next 8 bits green and then blue. The most significant 8 bits are not used. The packing process is done for you by the function RGB@. For example RGB@(255,0,0) represents pure red.

## Screen resolution

The number of pixels horizontally and vertically on the entire screen. Typical values are 640 x 480, 800 x 600, and 1024 x 768. The screen will also have a colour resolution, typically 16, 256, 16k, or higher.

## Screen saver

A program designed to run while the computer is idle. Usually such programs are designed to stop as soon as mouse or keyboard input is received (see %ns and %sv).

## Scroll bar

A scroll bar is a structure to the right of a window or control that can be used to shift the contents vertically, or at the bottom of a window to shift the contents horizontally. A scroll bar has an arrow structure at either end that will scroll the contents by one unit (e.g. one line if the contents is text) and a moveable control (known as a *thumb*) that can be dragged to make larger movements. Movements of one page (the exact meaning of which depends on the window contents) can be made by clicking above or below the thumb (see %hx and %vx).

## Shortcut keys

A shortcut key represents a way of accessing a menu using the keyboard. Each item of a menu can have a letter underlined, indicating that it can be selected by holding down the Alt key and pressing the letter (or certain other keys). The components of subsequent sub-menus can then be selected by pressing further keys corresponding to underlined letters in the menus. The whole process of underlining the letters and responding to the keyboard is controlled by merely prefixing the letter in the menu item by an ‘&’ (see %mn and %sm).

## Standard character string

This is the name given to text in a *format string* that is either placed in square brackets or is replaced by an @ symbol and represented by an argument.

### **Status bar**

This is a strip at the bottom of a window with a distinct colour (usually grey) in which special information is placed. See the %ob format for details of how to create one.

### **Thumb**

The name given to a blank box that appears on a scroll bar. One of the mechanisms for scrolling involves dragging this box with the mouse.

### **Toolbar**

A toolbar is a rectangular array of bitmap buttons, usually but not always located under the window menu bar. Toolbars can be created as arrays of bitmap buttons using %tb.

### **Win16**

This refers to programs that interact with the 16-bit API of Windows 3.1. This is also available under Windows 95. Salford compilers generate 32-bit code for Win16 that operates using a 16/32 interface. Although the Win16 API is available under Windows NT, there are certain restrictions that prevent the above 16/32 interface from working - Win32 must be used for this operating system. The *ClearWin+ User's Supplement* provides additional information for writing Win16 applications.

### **Win32**

This refers to programs that exploit the 32-bit mode of operation available in Windows 95 and Windows NT. All Salford compilers are 32-bit, but the non-Win32 products execute in 32-bit mode but provide an interface to the 16-bit operating system via DBOS and/or WDBOS.

### **Windows application**

A Windows program - that is any program that opens at least one window. Non Windows applications use DBOS and read and write to the screen or DOS box. Under Win32 the difference is very slight, however under Win 3.1 non-Windows applications created with Salford compilers use DBOS and operate from within a DOS box.

### **Windows control**

A windows control is any part of a window that operates as a distinctive entity. For example, buttons, %rd integer input boxes, and toolbars are all controls.

### **Windows API**

This is the set of routines supplied by Microsoft to access the features of Windows. All Windows programs ultimately call these routines. For example, requests to ClearWin+ are executed internally by making calls to the Windows API.

**Windows message**

Windows communicates information about events (such as key presses, mouse movements, etc.) to the program by means of ‘messages’. These are small structures that are stored in a queue and accessed by means of a call to the Windows API. This process is handled automatically by ClearWin+. Specific Windows messages are referred to by names such as WM\_PAINT, WM\_CREATE, etc.

**WM\_PAINT message**

This is a message sent by Windows to an application to tell it to ‘re-paint’ one of its windows (for example because another window has obscured it). ClearWin+ programmers do not need to be aware of the fact that a window’s contents may have to be regenerated in this way. However some call-backs, such as the re-colour call back from %eb edit boxes, may be activated in response to this message.



# Appendix A.

## Functions ported from DBOS

This appendix contains information about functions that feature in the Salford DOS (DBOS) graphics library. It describes the extent to which this library has been ported to ClearWin+. The following classification of DBOS graphics functions can be made in relation to ClearWin+.

1. Functions that are useful in new programs (marked ✓). Some of these will have identical usage under ClearWin+ to that under DBOS. For some the functionality is slightly different. Details are given in Chapter 27.
2. Functions that can be used in new programs (marked ☑) but for which there is a suitable alternative. Details for these functions are given in the *ClearWin+ User's Supplement*. A table of the alternative routines is given at the end of this appendix. The alternatives are documented in this guide.
3. Functions that could be ported from existing DBOS programs but which should not be written into new programs because there is a better alternative under ClearWin+ (marked ⇔). A table of alternative routines is given at the end of this appendix.
4. Functions which have no operation or are not useful under ClearWin+ but can be left in old programs that have been ported from DBOS (marked ○).
5. Functions which have not been implemented under ClearWin+ (marked ✗).

### Graphics

|                            |                                                           |   |
|----------------------------|-----------------------------------------------------------|---|
| CLEAR_SCREEN@              | Clears the <i>drawing surface</i> .                       | ✓ |
| CLEAR_SCREEN_AREA@         | Clears a rectangular area of the <i>drawing surface</i> . | ☒ |
| COMBINE_POLYGONS@          | Gets the handle for a combination of polygons.            | ⇒ |
| CREATE_POLYGON@            | Gets a handle for a specified polygon.                    | ⇒ |
| DELETE_POLYGON_DEFINITION@ | Deletes a polygon definition.                             | ⇒ |

|                                |                                                                                   |   |
|--------------------------------|-----------------------------------------------------------------------------------|---|
| DRAW_HERSHEY@                  | Draws an Hershey character.                                                       | ⇒ |
| DRAW_LINE@                     | Draws a straight line on the <i>drawing surface</i> .                             | ⇒ |
| DRAW_TEXT@                     | Draws text on the <i>drawing surface</i> .                                        | ⇒ |
| EGA@                           | Switches to EGA graphics mode.                                                    | ○ |
| ELLIPSE@                       | Draws an ellipse.                                                                 | ⇒ |
| FILL_ELLIPSE@                  | Fills an ellipse.                                                                 | ⇒ |
| FILL_POLYGON@                  | Fills a polygon.                                                                  | ⇒ |
| FILL_RECTANGLE@                | Fills a rectangle.                                                                | ⇒ |
| GET_ALL_PALETTE_REGS@          | Gets all palette registers for colour graphics.                                   | ⇒ |
| GET_DEVICE_PIXEL@              | Gets a pixel colour from a graphics region.                                       | ✗ |
| GET_GRAPHICS_MODES@            | Gets details of all the graphics modes.                                           | ○ |
| GET_GRAPHICS_RESOLUTION@       | Gets details of the high resolution graphics mode.                                | ○ |
| GET_PIXEL@                     | Gets a pixel colour from the <i>drawing surface</i> .                             | ⇒ |
| GET_TEXT_MODES@                | Gets information about the available text modes.                                  | ✗ |
| GET_TEXT_SCREEN_SIZE@          | Gets the resolution of the current text mode.                                     | ✗ |
| GET_VIDEO_DAC_BLOCK@           | Gets a block of VGA DAC registers.                                                | ⇒ |
| GRAPHICS_MODE_SET@             | Sets the graphics mode to a given resolution.                                     | ⇒ |
| GRAPHICS_WRITE_MODE@           | Selects replace/XOR mode before writing to the screen, virtual screen or printer. | ✓ |
| HERSHEY_PRESENT@               | Tests if a character number has a Hershey representation.                         | ⇒ |
| HIGH_RESOLUTION_GRAPHICS_MODE@ | Switches to high resolution graphics mode.                                        | ○ |
| IS_TEXT_MODE@                  | Tests if the screen is in text or graphics mode.                                  | ✗ |
| LOAD_STANDARD_COLOURS@         | Loads the standard colours for 256 colour mode.                                   | ✓ |
| MOVE_POLYGON@                  | Moves the position of a polygon.                                                  | ⇒ |
| POLYLINE@                      | Draws a number of connected straight lines.                                       | ⇒ |
| RECTANGLE@                     | Draws a rectangle.                                                                | ⇒ |
| RESTORE_GRAPHICS_BANK@         | Restores the graphics bank after a BIOS call.                                     | ○ |
| RESTORE_TEXT_SCREEN@           | Restores a text screen saved with <b>SAVE_TEXT_SCREEN@</b> .                      | ✗ |
| SAVE_TEXT_SCREEN@              | Saves the whole of the text screen.                                               | ✗ |
| SCREEN_TYPE@                   | Gets the graphics screen type.                                                    | ○ |
| SET_ALL_PALETTE_REGS@          | Sets all palette registers for colour graphics.                                   | ⇒ |
| SET_DEVICE_PIXEL@              | Sets a pixel colour on the <i>drawing surface</i> .                               | ☒ |

|                      |                                                     |   |
|----------------------|-----------------------------------------------------|---|
| SET_PALETTE@         | Sets a palette register for colour graphics.        | ⇒ |
| SET_PIXEL@           | Sets a pixel colour on the <i>drawing surface</i> . | ⇒ |
| SET_TEXT_ATTRIBUTE@  | Sets the current graphics text attributes.          | ☒ |
| SET_VIDEO_DAC@       | Sets a VGA DAC register.                            | ⇒ |
| SET_VIDEO_DAC_BLOCK@ | Sets a block of VGA DAC registers.                  | ⇒ |
| TEXT_MODE@           | Returns to text mode.                               | ○ |
| TEXT_MODE_SET@       | Selects the current text mode.                      | ○ |
| USE_VESA_INTERFACE@  | Forces the VESA interface to be used.               | ○ |
| VGA@                 | Switches to VGA graphics mode.                      | ○ |

## Graphics plotter/screen

|                             |                                                     |   |
|-----------------------------|-----------------------------------------------------|---|
| CLOSE_PLOTTER@              | Closes the plotter device or file.                  | ⇒ |
| CLOSE_VSCREEN@              | Closes the virtual screen.                          | ☒ |
| CREATE_SCREEN_BLOCK@        | Creates a screen block in memory.                   | ☒ |
| GET_DACS_FROM_SCREEN_BLOCK@ | Uses palette information from a PCX file.           | ☒ |
| GET_SCREEN_BLOCK@           | Saves a rectangular area of the screen.             | ☒ |
| NEW_PAGE@                   | Provides a new page on the <i>drawing surface</i> . | ✓ |
| OPEN_PLOT_DEVICE@           | Opens the plotter.                                  | ⇒ |
| OPEN_PLOT_FILE@             | Directs plotter output to a file.                   | ⇒ |
| OPEN_VSCREEN@               | Opens a screen block as the virtual screen.         | ☒ |
| PCX_TO_SCREEN_BLOCK@        | Loads a file a screen block.                        | ☒ |
| PLOTTER_SET_PEN_TYPE@       | Selects a pen type for the plotter.                 | ○ |
| RESTORE_SCREEN_BLOCK@       | Displays a previously saved area of the screen.     | ☒ |
| SCREEN_BLOCK_TO_PCX@        | Saves a screen block a file.                        | ☒ |
| SCREEN_BLOCK_TO_VSCREEN@    | Loads a screen block to the virtual screen.         | ☒ |
| SCREEN_TO_VSCREEN@          | Loads the graphics screen to the virtual screen.    | ☒ |
| VSCREEN_TO_PCX@             | Saves the virtual screen to a file.                 | ☒ |
| VSCREEN_TO_SCREEN@          | Loads the virtual screen to the graphics screen.    | ☒ |
| WRITE_TO_PLOTTER@           | Writes a string to the plotter.                     | ○ |

## Graphics printer

|                             |                                                            |   |
|-----------------------------|------------------------------------------------------------|---|
| CLOSE_GRAPHICS_PRINTER@     | Closes the graphics printer device or file.                | ⇒ |
| GET_PCL_PALETTE@            | Gets the colour definitions for a given number of colours. | ✓ |
| LOAD_PCL_COLOURS@           | Loads the standard colour definitions.                     | ✓ |
| OPEN_GPRINT_DEVICE@         | Opens a graphics printer.                                  | ⇒ |
| OPEN_GPRINT_FILE@           | Directs graphics printer output to a file.                 | ⇒ |
| PRINT_GRAPHICS_PAGE@        | Prints a graphics page.                                    | ✓ |
| SELECT_DOT_MATRIX@          | Selects an Epson compatible dot matrix printer             | ✗ |
| SELECT_PCL_PRINTER@         | Specifies attributes of a PCL printer.                     | ⇒ |
| SET_PCL_BITPLANES@          | Sets the number of colours in the image.                   | ○ |
| SET_PCL_GAMMA_CORRECTION@   | Alters the “gamma correction” for colours.                 | ○ |
| SET_PCL_GRAPHICS_DEPLETION@ | Improves the image quality.                                | ○ |
| SET_PCL_GRAPHICS_SHINGLING@ | Makes a number of print passes.                            | ○ |
| SET_PCL_LANDSCAPE@          | Sets LANDSCAPE or PORTRAIT orientation.                    | ☒ |
| SET_PCL_PALETTE@            | Loads the colour definitions.                              | ○ |
| SET_PCL_RENDER@             | Sets the “rendering algorithm”.                            | ○ |

## Mouse

|                               |                                                                                 |   |
|-------------------------------|---------------------------------------------------------------------------------|---|
| DISPLAY_MOUSE_CURSOR@         | Shows the mouse cursor on the screen.                                           | ○ |
| GET_MOUSE_BUTTON_PRESS_COUNT@ | Gets the number of times a button has been pressed.                             | ○ |
| GET_MOUSE_EVENT_MASK@         | Gets the mask for the most recent mouse interrupt.                              | ○ |
| GET_MOUSE_PHYSICAL_MOVEMENT@  | Gets the mouse pad distance from the last call.                                 | ○ |
| GET_MOUSE_POSITION@           | Gets the present state of the mouse cursor.                                     | ⇒ |
| GET_MOUSE_SENSITIVITY@        | Gets the values of the physical movement ratios and the double speed threshold. | ○ |
| HIDE_MOUSE_CURSOR@            | Hides the mouse cursor on the screen.                                           | ○ |
| INITIALISE_MOUSE@             | Initialises the mouse driver.                                                   | ○ |
| MOUSE@                        | Performs a mouse interrupt.                                                     | ○ |

|                             |                                                                           |   |
|-----------------------------|---------------------------------------------------------------------------|---|
| MOUSE_CONDITIONAL_OFF@      | Switches off the cursor when it enters a specified rectangle.             | ○ |
| MOUSE_LIGHT_PEN_EMULATION@  | Uses the mouse as a light-pen.                                            | ○ |
| MOUSE_SOFT_RESET@           | Initialises the mouse.                                                    | ⇒ |
| QUERY_MOUSE_SAVE_SIZE@      | Gets the buffer size for the mouse state.                                 | ○ |
| RESTORE_MOUSE_DRIVER_STATE@ | Restores a former state of the mouse driver.                              | ○ |
| SAVE_MOUSE_DRIVER_STATE@    | Saves the current state of the mouse driver.                              | ○ |
| SET_MOUSE_BOUNDS@           | Restricts mouse movements to a specified rectangle.                       | ○ |
| SET_MOUSE_GRAPHICS_CURSOR@  | Specifies the shape of the mouse cursor for graphics mode.                | ○ |
| SET_MOUSE_INTERRUPT_MASK@   | Enables mouse actions to produce interrupts.                              | ○ |
| SET_MOUSE_MOVEMENT_RATIO@   | Sets the mouse cursor sensitivity.                                        | ○ |
| SET_MOUSE_POSITION@         | Moves the mouse cursor to a particular position.                          | ⇒ |
| SET_MOUSE_SENSITIVITY@      | Sets the mouse cursor sensitivity and the threshold for the double speed. | ○ |
| SET_MOUSE_SPEED_THRESHOLD@  | Sets the threshold for double speed.                                      | ○ |
| SET_MOUSE_TEXT_CURSOR@      | Specifies details of the mouse cursor for text mode.                      | ○ |

## Table of alternative routines

| DBOS routine               | ClearWin+ alternative        |
|----------------------------|------------------------------|
| CLEAR_SCREEN_AREA@         | DRAW_FILLED_RECTANGLE@       |
| CLOSE_GRAPHICS_PRINTER@    | CLOSE_PRINTER@               |
| CLOSE_PLOTTER@             | CLOSE_PRINTER@               |
| CLOSE_VSCREEN@             | CREATE_GRAPHICS_REGION@ etc. |
| COMBINE_POLYGONS@          | DRAW_FILLED_POLYGON@         |
| CREATE_POLYGON@            | DRAW_FILLED_POLYGON@         |
| CREATE_SCREEN_BLOCK@       | CREATE_GRAPHICS_REGION@ etc. |
| DELETE_POLYGON_DEFINITION@ | DRAW_FILLED_POLYGON@         |
| DRAW_HERSHEY@              | DRAW_CHARACTERS@             |
| DRAW_LINE@                 | DRAW_LINE_BETWEEN@           |
| DRAW_TEXT@                 | DRAW_CHARACTERS@             |

|                          |                              |
|--------------------------|------------------------------|
| ELLIPSE@                 | DRAW_ELLIPSE@                |
| FILL_ELLIPSE@            | DRAW_FILLED_ELLIPSE@         |
| FILL_POLYGON@            | DRAW_FILLED_POLYGON@         |
| FILL_RECTANGLE@          | DRAW_FILLED_RECTANGLE@       |
| GET_ALL_PALETTE_REGS@    | GET_COLOURS@ *               |
| GET_MOUSE_POSITION@      | CLEARWIN_INFO@               |
| GET_PIXEL@               | GET_POINT@                   |
| GET_SCREEN_BLOCK@        | CREATE_GRAPHICS_REGION@ etc. |
| GET_VIDEO_DAC_BLOCK@     | GET_COLOURS@ *               |
| GRAPHICS_MODE_SET@       | LOAD_STANDARD_COLOURS@ *     |
| HERSHEY_PRESENT@         | DRAW_CHARACTERS@             |
| MOUSE_SOFT_RESET@        | SET_MOUSE_CURSOR_POSITION@   |
| MOVE_POLYGON@            | DRAW_FILLED_POLYGON@         |
| OPEN_GPRINT_DEVICE@      | OPEN_PRINTER@                |
| OPEN_GPRINT_FILE@        | OPEN_PRINTER_TO_FILE@        |
| OPEN_PLOT_DEVICE@        | OPEN_PRINTER@                |
| OPEN_PLOT_FILE@          | OPEN_PRINTER_TO_FILE@        |
| OPEN_VSCREEN@            | CREATE_GRAPHICS_REGION@ etc. |
| PCX_TO_SCREEN_BLOCK@     | CREATE_GRAPHICS_REGION@ etc. |
| POLYLINE@                | DRAW_POLYLINE@               |
| RECTANGLE@               | DRAW_RECTANGLE@              |
| RESTORE_SCREEN_BLOCK@    | CREATE_GRAPHICS_REGION@ etc. |
| SCREEN_BLOCK_TO_PCX@     | CREATE_GRAPHICS_REGION@ etc. |
| SCREEN_BLOCK_TO_VSCREEN@ | CREATE_GRAPHICS_REGION@ etc. |
| SCREEN_TO_VSCREEN@       | CREATE_GRAPHICS_REGION@ etc. |
| SELECT_PCL_PRINTER@      | OPEN_PRINTER@                |
| SET_ALL_PALETTE_REGS@    | SET_COLOURS@ *               |
| SET_DEVICE_PIXEL@        | DRAW_POINT@                  |
| SET_MOUSE_POSITION@      | SET_MOUSE_CURSOR_POSITION@   |
| SET_PALETTE@             | SET_COLOURS@ *               |
| SET_PCL_LANDSCAPE@       | SET_PRINTER_ORIENTATION@     |
| SET_PIXEL@               | DRAW_POINT@                  |
| SET_TEXT_ATTRIBUTE@      | BOLD_FONT@ etc.              |

\* These routines are documented in the *ClearWin+ User's Supplement*.

|                      |                              |
|----------------------|------------------------------|
| SET_VIDEO_DAC@       | SET_COLOURS@*                |
| SET_VIDEO_DAC_BLOCK@ | SET_COLOURS@*                |
| VSCREEN_TO_PCX@      | CREATE_GRAPHICS_REGION@ etc. |
| VSCREEN_TO_SCREEN@   | CREATE_GRAPHICS_REGION@ etc. |

---



# Index

## %

%ac, 88, 215  
%ap, 95, 216  
%aw, 123, 216  
%bc, 54, 217  
%bd, 217  
%bf, 97, 218  
%bg, 127, 218  
%bh, 119, 219  
%bi, 54, 219  
%bk, 220  
%bm, 75, 220  
%br, 61, 221  
%bt, 51, 221  
%bv, 71, 222  
%bx, 59, 225  
%ca, 121, 225  
%cb, 226  
%cc, 130, 226  
%ch, 123, 226  
%cl, 73, 227  
%cn, 89, 227  
%co, 40, 228  
%cu, 76, 228  
%cv, 123, 230  
%cw, 128, 230  
%dc, 76, 231  
%dd, 41, 232  
%de, 232  
%df, 41, 233  
%dl, 130, 233  
%dp, 42, 263  
%dr, 125, 234  
%dw, 157, 234  
%dy, 235  
%eb, 107, 235  
%el, 236  
%ep, 42, 263  
%eq, 100, 237  
%es, 237  
%ew, 238  
%fb, 238  
%fd, 238  
%ff, 239  
%fh, 239  
%fl, 41, 239  
%fn, 97, 240  
%fp, 42, 263  
%fr, 123, 240

%fs, 105, 241  
%ft, 106, 241  
%ga, 59, 242  
%gd, 95, 242  
%gf, 97, 242  
%gi, 243  
%gp, 94, 244  
%gr, 136, 245  
%he, 120, 245  
%ht, 199, 246  
%hw, 121, 246  
%hx, 62, 247  
%ib, 58, 247  
%ic, 77, 249  
%if, 249  
%il, 42, 250  
%it, 97, 250  
%lc, 121, 250  
%ld, 251  
%ls, 62, 251  
%lv, 252  
%lw, 122, 256  
%mg, 257  
%mi, 78, 257  
%mn, 81, 258  
%ms, 64, 258  
%mv, 259  
%nc, 260  
%nd, 260  
%nl, 260  
%nr, 261  
%ns, 133, 261  
%ob, 91, 261  
%og, 263  
%pb, 42, 263  
%pd, 100, 264  
%pl, 264  
%pm, 84, 264  
%ps, 129, 265  
%pv, 266  
%rb, 55, 266  
%rd, 37, 267  
%re, 268  
%rf, 38, 269  
%rj, 90, 270  
%rm, 270  
%rp, 270  
%rs, 40, 271  
%sc, 123, 272

%sd, 98, 272  
 %sf, 98, 273  
 %sh, 130, 273  
 %si, 78, 273  
 %sl, 44, 274  
 %sm, 86, 275  
 %sp, 94, 126, 275  
 %ss, 48, 275  
 %st, 100, 276  
 %su, 98, 277  
 %sv, 132, 277  
 %sy, 277  
 %sz, 126, 278  
 %ta, 279  
 %tb, 56, 279  
 %tc, 99, 280  
 %th, 119, 281  
 %ti, 281  
 %tl, 90, 282  
 %tp, 42, 263  
 %ts, 99, 282  
 %tt, 54, 283  
 %tv, 68, 283  
 %tx, 103, 285  
 %ul, 97, 286  
 %up, 42, 263  
 %uw, 131, 286  
 %vx, 62, 286  
 %wc, 45  
 %wd, 45  
 %we, 46  
 %wf, 47  
 %wg, 47  
 %wp, 131, 288  
 %ws, 48  
 %ww, 126, 289  
 %wx, 48

**&**

& menu accelerator, 81  
 +, standard call-back, 212  
 ? format modifier, 24, 119  
 ^ format modifier, 24  
 ` format modifier, 24  
 ~ format modifier, 24

**A**

ABORT@ routine, 329  
 ABOUT, standard call-back, 206  
 Absolute position, 95  
 Accelerator keys, 88, 421  
 ACTIVATE\_BITMAP\_PALETTE@ routine, 329  
 ADD\_ACCELERATOR@ routine, 330  
 ADD\_CURSOR\_MONITOR@ routine, 330  
 ADD\_FOCUS\_MONITOR@ routine, 331  
 ADD\_GRAPHICS\_ICON@ routine, 331  
 ADD\_HYPertext@ routine, 332

ADD\_HYPertext\_RESOURCE@ routine, 332  
 ADD\_KEYBOARD\_MONITOR@ routine, 333  
 ADD\_MENU\_ITEM@ routine, 333  
 ADD\_WINIO\_CHARACTER@ routine, 29  
 ADD\_WINIO\_FLOAT@ routine, 29  
 ADD\_WINIO\_FUNCTION@ routine, 29  
 ADD\_WINIO\_INTEGER@ routine, 29  
 ADVANCE\_EDIT\_BUFFER@ routine, 115  
**API**  
 CreatePen, 158  
 FillRect, 158  
 MessageBeep, 79  
 RGB, 99, 128  
 SelectObject, 335  
 Attach window (child window), 123  
 ATTACH\_BITMAP\_PALETTE@ routine, 334

**B**

Background colour  
 of window, 127  
 BEEP, standard call-back, 206  
 Bitmap, 75, 421  
 Bold fonts, 97  
 BOLD\_FONT@ routine, 334  
 Box selection, 144  
 Boxes, 91, 311  
 behaviour with %cn, 90  
 shaded, 262  
 BREAK\_MODEM\_CONNECTION@ routine, 204  
 Button, 421  
 bitmap button, 56  
 call-back, 51  
 colour, 54  
 greying, 52  
 icon format, 54  
 radio, 55  
 textual toolbar, 54

**C**

Call-back  
 HTML\_PRINTER\_OPEN, 191  
 Call-back function, 421  
 Call-backs, 21  
 Captions, 121, 421  
 CASCADE, standard call-back, 206  
 Centring, 89  
 CHANGE\_HYPertext@ routine, 335  
 CHANGE\_PEN@ routine, 335  
 Child window format, 123  
 Child windows, 421  
 MDI frames, 123  
 other, 123  
 CHOOSE\_COLOUR@ routine, 335  
 CHOOSE\_FONT@ routine, 336  
 CLEAR\_BITMAP@ routine, 336

CLEAR\_SCREEN@ routine, 336  
 CLEAR\_WINDOW@ routine, 337  
 ClearWin window, 422  
     text output, 301  
 CLEARWIN\_FLOAT@ routine, 337  
 CLEARWIN\_INFO@  
     ACTION\_X, 337  
     ACTION\_Y, 337, 338  
     ACTIVE\_EDIT\_BOX, 337, 338  
     CALL\_BACK\_WINDOW, 337, 338  
     CURSOR\_WINDOW, 337, 338  
     DRAGGED\_ICON, 337, 338  
     DROPPED\_COUNT, 337, 338  
     DROPPED\_CURRENT, 337, 338  
     DROPPED\_ICON, 337, 338  
     FOCUS\_WINDOW, 337, 338  
     FOCUSSED\_WINDOW, 338  
     GAINING\_FOCUS, 338  
     GIF\_MOUSE\_X, 338  
     GIF\_MOUSE\_Y, 338, 339  
     GRAPHICS\_DC, 338, 339  
     GRAPHICS\_DEPTH, 338, 339  
     GRAPHICS\_HDC, 338, 339  
     GRAPHICS\_MOUSE\_FLAGS, 339  
     GRAPHICS\_MOUSE\_X, 339  
     GRAPHICS\_MOUSE\_Y, 339, 340  
     GRAPHICS\_RESIZING, 338, 339  
     GRAPHICS\_WIDTH, 338, 339  
     LATEST\_FORMATTED\_WINDOW, 339, 340  
     LATEST\_VARIABLE, 339, 340  
     LATEST\_WINDOW, 339, 340  
     LISTBOX\_ITEM\_SELECTED, 339, 340  
     LOSING\_FOCUS, 339, 340  
     MESSAGE\_HWND, 339, 340  
     MESSAGE\_LPARAM, 339, 340  
     MESSAGE\_WPARAM, 339, 340  
     OPENGL\_DEPTH, 339, 340  
     OPENGL\_DEVICE\_CONTEXT, 339, 340  
     OPENGL\_WIDTH, 339, 340  
     PIXELS\_PER\_H\_UNIT, 339, 340  
     PIXELS\_PER\_V\_UNIT, 339, 341  
     PRINTER\_COLLATE, 339, 341  
     SCREEN\_DEPTH, 339, 341  
     SCREEN\_WIDTH, 339, 341  
     SHEET\_NO, 339, 341  
     SPIN\_CONTROL\_USED, 339, 341  
     TEXT\_ARRAY\_CHAR, 103, 339, 341  
     TEXT\_ARRAY\_MOUSE\_FLAGS, 104  
     TEXT\_ARRAY\_RESIZING, 104  
     TEXT\_ARRAY\_WIDTH, 104  
     TEXT\_ARRAY\_X, 104  
     TEXT\_ARRAY\_Y, 104  
     TREEVIEW\_ITEM\_SELECTED, 339, 341  
 CLEARWIN\_INFO@ routine, 337  
 CLEARWIN\_STRING@ routine, 337

CURRENT\_MENU\_ITEM, 342  
 CURRENT\_TEXT\_ITEM, 341, 342  
 DROPPED\_FILE, 342  
 PRINTER\_DOCUMENT, 401  
 CLEARWIN\_STRING@ routine, 341  
 CLEARWIN\_VERSION@ routine, 343  
 Clipboard, 422  
 CLIPBOARD\_TO\_SCREEN\_BLOCK@ routine, 344  
 CLOSE\_CD\_TRAY@ routine, 344  
 CLOSE\_METAFILE@ routine, 345  
 CLOSE\_PRINTER@ routine, 345  
 CLOSE\_PRINTER\_ONLY@ routine, 345  
 CLOSE\_WAV\_FILE@ routine, 346  
 Closing a window, 25  
 Closure control format, 130  
 Colour modes, 136, 422  
 Colour palette, 73  
 Compiler option /WINDOWS, 293  
 Compiler options, 293  
 Compiling a Windows application, 293  
 Conditional hypertext, 203  
 CONFIRM\_EXIT, standard call-back, 206  
 CONTINUE, standard call-back, 207  
 CONTROL\_NAME\_FROM\_HANDLE@ routine, 313  
 COPY, standard call-back, 206  
 COPY\_FROM\_CLIPBOARD@ routine, 346  
 COPY\_GRAPHICS\_REGION@ routine, 347  
 COPY\_TO\_CLIPBOARD@ routine, 348  
 CREATE\_BITMAP@ routine, 349  
 CREATE\_CURSOR@ routine, 349  
 CREATE\_GRAPHICS\_REGION@ routine, 350  
 CREATE\_WINDOW@ routine, 350  
 Cursor format, 76  
 CUT, standard call-back, 206

**D**

Debugging with an auxilliary terminal, 312  
 DEC, standard call-back, 208  
 Default cursor, 76  
 DEFINE\_FILE\_EXTENSION@ routine, 351  
 Delayed auto recall, 130  
 DELETE\_GRAPHICS\_REGION@ routine, 352  
 DESTROY\_WINDOW@ routine, 352  
 DIB\_PAINT@ routine, 352  
 Disable screen saver, 133  
 DISPLAY\_DIB\_BLOCK@ routine, 353  
 DISPLAY\_POPUP\_MENU@ routine, 353  
 DLL, 422  
 DO\_COPIES@ routine, 354  
 DRAW\_BEZIER@ routine, 354  
 DRAW\_CHARACTERS@ routine, 354  
 DRAW\_ELLIPSE@ routine, 355  
 DRAW\_FILLED\_ELLIPSE@ routine, 355  
 DRAW\_FILLED\_POLYGON@ routine, 356  
 DRAW\_FILLED\_RECTANGLE@ routine, 356  
 DRAW\_LINE\_BETWEEN@ routine, 356  
 DRAW\_POINT@ routine, 357

DRAW\_POLYLINE@ routine, 357  
 DRAW\_RECTANGLE@ routine, 357  
 Drawing surfaces, 422

**E**

Edit box, 107, 423  
 EDIT\_CLOSE\_MARK@ routine, 117  
 EDIT\_DELETE\_LINES@ routine, 116  
 EDIT\_FILE, standard call-back, 207  
 EDIT\_FILE\_OPEN, standard call-back, 207  
 EDIT\_FILE\_SAVE, standard call-back, 207  
 EDIT\_FILE\_SAVE\_AS, standard call-back, 207  
 EDIT\_MOVE\_BOF@ routine, 116  
 EDIT\_MOVE\_END@ routine, 116  
 EDIT\_MOVE\_HOME@ routine, 116  
 EDIT\_MOVE\_TOF@ routine, 116  
 EDIT\_OPEN\_BLOCK\_MARK@ routine, 117  
 EDIT\_OPEN\_LINE\_MARK@ routine, 117, 118  
 Enhanced menu, 86  
 EQUIVALENCE, 310  
 EXIT, standard call-back, 208  
 EXPORT\_BMP@ routine, 358  
 EXPORT\_PCX@ routine, 358

**F**

FATAL, standard call-back, 208  
 FEED\_WKEYBOARD@ routine, 359  
 File filter, 106  
 File selection, 105  
 FILE\_OPENR, standard call-back, 208  
 FILE\_OPENW, standard call-back, 209  
 FILL\_SURFACE@ routine, 359  
 FILL\_TO\_BORDER@ routine, 359  
 FIND\_EDIT\_STRING@ routine, 114  
 FIND\_REPLACE\_EDIT\_STRING@ routine, 114  
 FLUSH\_WKEYBOARD@ routine, 360  
 Focus, 331, 337, 338, 423  
 Font, 423  
     format, 97  
     get, 97  
 FONT\_METRICS@ routine, 360  
 Format codes, 23  
 Format codes, index of, 31  
 Format modifiers, 24  
 Format windows, 19  
 Fortran  
     calling Windows API routines from, 305  
     using the Windows API, 310

**G**

Gang controls together, 59  
 Get and set window position, 94  
 Get font name, 97  
 GET\_BITMAP\_DC@ routine, 360  
 GET\_CLEARWIN\_TEXT@ routine, 303, 361  
 GET\_CURRENT\_DC@ routine, 361

GET\_DEFAULT\_WINDOW@ routine, 361  
 GET\_DIB\_BLOCK@ routine, 361  
 GET\_DIB\_SIZE@ routine, 362  
 GET\_FILTERED\_FILE@ routine, 363  
 GET\_FONT\_ID@ routine, 364  
 GET\_FONT\_NAME@ routine, 365  
 GET\_GRAPHICAL\_RESOLUTION@ routine, 365  
 GET\_GRAPHICAL\_SELECTED\_AREA@ routine, 365  
 GET\_IM\_INFO@ routine, 366  
 GET\_MATCHED\_COLOUR@ routine, 366  
 GET\_MOUSE\_INFO@ routine, 367  
 GET\_NEAREST\_SCREEN\_COLOUR@ routine, 368  
 GET\_POINT@ routine, 368  
 GET\_PRINTER\_ORIENTATION@ routine, 368  
 GET\_RGB\_VALUE@ routine, 369  
 GET\_SCREEN\_DIB@ routine, 369  
 GET\_SYSTEM\_FONT@ routine, 369  
 GET\_TEXT\_SIZE@ routine, 370  
 GET\_TRACK\_LENGTH@ routine, 370  
 GET\_WINDOW\_LOCATION@ routine, 371  
 GET\_WKEY@ routine, 371  
 GET\_WKEY1@ routine, 372  
 GetTextMetrics, 310  
 GIF file, 423  
 GPRINTER\_OPEN, standard call-back, 209  
 Graphics, 136

    background colour, 137  
     basic primitives, 138  
     call-backs, 142  
     drawing text, 138  
     icons, 152  
     off-screen graphics, 150  
     printers, 195  
     re-sizing, 141  
     selection modes, 143  
     USER\_SURFACE, 148  
 Graphics selection  
     Box selection, 144  
     Line selection, 144  
     Rubber-banding, 143

GRAPHICS\_TO\_CLIPBOARD@ routine, 372  
 GRAPHICS\_WRITE\_MODE@ routine, 373  
 Grey buttons, 52  
 Greyed menus, 81  
 Greyed out, 423  
 Greying controls, 24  
 Grid display, 95  
 GUI, 423

**H**

Handle, 423  
 HANDLE\_FROM\_CONTROL\_NAME@ routine, 313  
 Help format, 120  
 HELP\_CONTENTS, standard call-back, 209  
 HELP\_ON\_HELP, standard call-back, 209  
 HIBYTE@ routine, 373  
 High colour device, 424

HIGH\_COLOUR\_MODE@ routine, 373

HIWORD@ routine, 374

Horizontal scroll bar, 62

HTML, 424

Hypertext, 199

&, 200

<, 200

>, 200

bold, 201

colours, 201

conditional, 203

DOC, 200

horizontal rule, 201

HREF, 200

HTML, 200

images, 201

italic, 201

lists, 201

Paragraph, 201

TITLE, 200

underline, 201

## I

Icons, 77, 424

IMPLICIT NONE, 295

IMPORT\_BMP@ routine, 374

IMPORT\_PCX@ routine, 375

INC, standard call-back, 209

INSERT\_EDIT\_STRING@ routine, 116

INTERNET\_HYPERLINK, standard call-back, 210

Italic fonts, 97

ITALIC\_FONT@ routine, 375

## L

Line selection, 144

List box, 62

LOBYTE@ routine, 375

LOWORD@ routine, 376

## M

MAKE\_BITMAP@ routine, 376

MAKE\_ICON@ routine, 377

MAP\_FILE\_FOR\_READ\_WRITE@, 378

MAP\_FILE\_FOR\_READING@, 378

Maximise, 424

MDI frames, 123, 424

Menus, 81, 424

METAFILE\_TO\_CLIPBOARD@ routine, 379

Minimise, 424

Minimise icon, 78

MOVE\_WINDOW@ routine, 379

MOVIE\_PLAYING@ routine, 380

Multiple selection boxes, 64

## N

NEW\_PAGE@ routine, 380

NEXT\_EDIT\_OPERATION@ routine, 115

Null terminated strings, 108

## O

Open box, 91

OPEN\_CD\_TRAY@ routine, 380

OPEN\_EDIT\_FILE@ routine, 116

OPEN\_GL\_PRINTER@ routine, 381

OPEN\_GL\_PRINTER1@ routine, 381

OPEN\_METAFILE@ routine, 382

OPEN\_PRINTER@ routine, 382

OPEN\_PRINTER\_TO\_FILE@ routine, 383

OPEN\_PRINTER1@ routine, 383

OPEN\_PRINTER1\_TO\_FILE@ routine, 384

OPEN\_TO\_WINDOW@ routine, 384

OPEN\_WAV\_FILE\_READ@ routine, 385

OPEN\_WAV\_FILE\_WRITE@ routine, 385

Owner draw box, 157

## P

Parameter box, 42

PASTE, standard call-back, 206

PAUSE Fortran statement, 309

PERFORM\_GRAPHICS\_UPDATE@ routine, 385

PERMIT\_ANOTHER\_CALLBACK@ routine, 386

PLAY\_AUDIO\_CD@ routine, 386

PLAY\_CLIPBOARD\_METAFILE@ routine, 387

PLAY\_SOUND@ routine, 387

PLAY\_SOUND\_RESOURCE@ routine, 388

Popup menus, 84

PRINT\_ABORT, standard call-back, 210

PRINT\_DIB@ routine, 388

PRINT\_GRAPHICS\_PAGE@ routine, 389

PRINT\_OPENGL\_IMAGE@ routine, 389

PRINTER\_OPEN, standard call-back, 210

PRINTER\_OPEN1, standard call-back, 210

PRINTER\_SELECT, standard call-back, 210

Property sheet, 129

Property sheet close, 130

PUT\_DIB\_BLOCK@ routine, 391

## R

READ\_URL@ routine, 204

RECORD\_SOUND@ routine, 391

Recording events, 313

RELEASE\_BITMAP\_DC@ routine, 392

RELEASE\_SCREEN\_DIB@ routine, 392

REMOVE\_ACCELERATOR@ routine, 392

REMOVE\_CURSOR\_MONITOR@ routine, 393

REMOVE\_FOCUS\_MONITOR@ routine, 393

REMOVE\_GRAPHICS\_ICON@ routine, 393

REMOVE\_KEYBOARD\_MONITOR@ routine, 394

REMOVE\_MENU\_ITEM@ routine, 394

REPLY\_TO\_TEXT\_MESSAGE @ routine, 394

RESIZE\_WINDOW@ routine, 394

Resource Compiler, using SRC, 296

Resources, 425

RGB colours, 425

RGB@ routine, 395

Right justify, 90

ROTATE\_FONT@ routine, 395

Routines

- Graphics, 429

- Graphics plotter/screen, 431

- Graphics printer, 432

- Mouse, 432

Rubber-banding, 143

## S

Save settings, 48

SCALE\_FONT@ routine, 395

Screen resolution, 425

Screen saver, 132, 425

Scroll bar, 425

SCROLL\_EDIT\_BUFFER@ routine, 115

SCROLL\_GRAPHICS@ routine, 396

SEE\_PROPERTYSHET\_PAGE @ routine, 396

SEE\_TREEVIEW\_SELECTION@ routine, 397

SELECT\_ALL, standard call-back, 211

SELECT\_FONT@ routine, 397

SELECT\_FONT\_ID@ routine, 398

SELECT\_GRAPHICS\_OBJECT@ routine, 398

SELECT\_PRINTER@ routine, 398

SEND\_TEXT\_MESSAGE @ routine, 399

SET, standard call-back, 211

SET\_CURSOR\_WAITING@ routine, 403

SET\_ALL\_MAX\_LINES@ routine, 400

SET\_CD\_POSITION@ routine, 400

SET\_CLEARWIN\_FLOAT@ routine, 400

SET\_CLEARWIN\_INFO@ routine, 401

SET\_CLEARWIN\_STRING@ routine, 401

SET\_CLEARWIN\_STYLE@ routine, 402

SET\_DEFAULT\_FONT@ routine, 403

SET\_DEFAULT\_TO\_FIXED\_FONT@ routine, 404

SET\_DEFAULT\_TO\_PROPORIONAL\_FONT@ routine, 404

SET\_DEFAULT\_WINDOW@ routine, 404

SET\_GRAPHICS\_SELECTION@ routine, 404

SET\_LINE\_STYLE@ routine, 406

SET\_LINE\_WIDTH@ routine, 406

SET\_MAX\_LINES@ routine, 407

SET\_MOUSE\_CURSOR\_POSITION@ routine, 407

SET\_OLD\_RESIZE\_MECHANISM@ routine, 407

SET\_OPEN\_DIALOG\_PATH@ routine, 408

SET\_PRINTER\_ORIENTATION@ routine, 408

SET\_RGB\_COLOURS\_DEFAULT@ routine, 408

SET\_SOUND\_SAMPLE\_RATE@ routine, 409

Shortcut keys, 425

SHOW\_MOVIE@ routine, 409

SIZE\_IN\_PIXELS@ routine, 410

SIZE\_IN\_POINTS@ routine, 410

SIZEOF\_CLIPBOARD\_TEXT@ routine, 410

Slider controls, 44

SLINK commands, 295

SOUND, standard call-back, 211

SOUND\_PLAYING@ routine, 411

SOUND\_RECORDING@ routine, 411

SOUND\_SAMPLE\_RATE@ routine, 411

SRC Resource Compiler, 296

Standard call-backs

- + , 212

- EDIT\_FILE\_SAVE\_AS, 207

- ABOUT, 206

- BEEP, 206

- CASCADE, 206

- CONFIRM\_EXIT, 206

- CONTINUE, 207

- COPY, CUT, PASTE, 206

- DEC, 208

- EDIT\_FILE, 207

- EDIT\_FILE\_OPEN, 207

- EDIT\_FILE\_SAVE, 207

- EXIT, 208

- FATAL, 208

- FILE\_OPENR, 208

- FILE\_OPENW, 209

- GPRINTER\_OPEN, 209

- HELP\_CONTENTS, 209

- HELP\_ON\_HELP, 209

- INC, 209

- INTERNET\_HYPERLINK, 210

- PRINT\_ABORT, 210

- PRINTER\_OPEN, 210

- PRINTER\_OPEN1, 210

- PRINTER\_SELECT, 210

- SELECT\_ALL, 211

- SET, 211

- SOUND, 211

- STOP, 211

- SUPER\_MAXIMISE, 211

- TEXT, 212

- TEXT\_HISTORY, 212

- TOGGLE, 212

Standard character string, 425

Standard icons, 78

START\_CLEARWIN\_PLAYBACK@ routine, 314

START\_CLEARWIN\_RECORDER@ routine, 313

START\_PROCESS@ routine, 412

Startup call-back, 123

Status bar, 426

STDCALL statement, 306

Sterling pound symbol, 100

STOP Fortran statement, 309

STOP, standard call-back, 211

STOP\_AUDIO\_CD@ routine, 412

STOP\_CLEARWIN\_RECORDING@ routine, 313

Subscript font, 98  
SUPER\_MAXIMISE, standard call-back, 211  
Superscript font, 98  
System font, 98  
System menu, 86  
SYSTEM\_FIXED\_FONT, 301

## T

Tabs, 90  
TEMPORARY\_YIELD@ routine, 412  
Text array, 103  
Text colour, 99  
Text size, 99  
TEXT, standard call-back, 212  
TEXT\_HISTORY, standard call-back, 212  
Thumb, 426  
TOGGLE, standard call-back, 212  
Toolbar, 426  
Tree view selection box, 68

## U

Underline font, 97  
UNDERLINE\_FONT@ routine, 413  
UNDO\_EDIT\_CHANGE@ routine, 115  
UNMAP\_FILE@ routine, 413  
UPDATE\_WINDOW@ routine, 413  
USE\_APPROXIMATE\_COLOURS@ routine, 414  
USE\_RESOURCE\_LIBRARY@ routine, 294  
USE\_RGB\_COLOURS@ routine, 414  
USE\_URL@ routine, 204  
USE\_WINDOWS95\_FONT@ routine, 414  
User defined child window, 131

## V

Variable string, 100  
Vertical scroll bar, 62

## W

Wallpaper, 131  
WAV\_FILE\_READ@ routine, 415  
WAV\_FILE\_WRITE@ routine, 415  
Web links, 204  
WIN\_COUA@ routine, 416  
WIN\_GETCHAR@ routine, 416  
WIN\_GETLINE@ routine, 416  
Win16, 426  
Win32, 426  
Window control types, 126  
Window position, 126  
Window size, 126  
WINDOW\_UPDATE@, 27  
Windows 95  
    Graphics region updates, 150  
    Scalable font, 98  
    WINDOWS\_95\_FUNCTIONALITY, 98  
Windows API, 426  
Windows application, 426  
WINDOWS compiler option, 293  
Windows control, 426  
Windows message, 427  
WINDOWS.INS, 294  
WINDOWS\_95\_FUNCTIONALITY @ routine, 416  
WINIO@ routine, 22  
WKEY\_WAITING@ routine, 417  
WM\_PAINT message, 427  
WRITE\_GRAPHICS\_TO\_BMP@ routine, 417  
WRITE\_GRAPHICS\_TO\_PCX@ routine, 417  
WRITE\_WAV\_FILE@ routine, 418

## Y

YIELD\_PROGRAM\_CONTROL@ routine, 418

