Introduction to Formatted windows

This chapter gives full details of the ClearWin+ function, `winio`. `winio` can be 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 can 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.

Much of the interaction between a Windows application and the user is performed via dialog boxes. Very simple dialog boxes can be created using the Windows API **MessageBox** function.

For example:

```
i=MessageBox(NULL,"Hello","Greeting",MB_OK);
```

The **MessageBox** function presents information to the user and invites the user to press one of a small set of pre-defined buttons (`OK`, `CANCEL`, `YES`, `NO`). Because this function is so easy for the programmer to code, many software products make use of the **MessageBox** dialog box even when the small range of canned responses available is not appropriate. The result is that dialog boxes often appear that are difficult for the user to interpret.

More sophisticated dialog boxes are traditionally created by writing a resource script that describes the appearance of the box and by interfacing the resource script to a complex call-back function that provides the desired control.

Although ClearWin+ fully supports this traditional method of dialog box programming, dialog boxes are easily created using `winio`, and this is a good place to begin an exploration of the possibilities offered by this function.

In most cases, when `winio` is employed, no resource script is required. Where a script is required, it can be attached to the C code by using a `#pragma resource` directive or linked separately.

Many windows programming tasks can be performed using `winio` without the need to supply any call-back functions. This is important because when the logic of a program has been scattered among many call-back functions the result can be very hard to read. Where call-back functions are required, they take no arguments and return an integer result. As a result, such functions may be coded in C, C++, or Fortran.

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 `winio`.

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 can be added to the window. As the example stands, clicking on these buttons has no effect since code for the corresponding processes has been omitted.

```
      options(INTL)
      winapp 500000,500000
      include <windows.ins>
      character*129 file,new_file,help_file
      integer*4 compile,link,run,i
      external compile,link,run
      help_file='myhelp.hlp'
      i=winio@('%ca[Example]&')
      i=winio@('%mn[&File[&Open,&Save,Save &As,E&xit]]&',
     1   'EDIT_FILE_OPEN',   '*.*',file,
     2   'EDIT_FILE_SAVE',   '*.*',new_file,
     3   'EDIT_FILE_SAVE_AS','*.*',new_file,
     4   'EXIT')
      i=winio@('%mn[&Edit[&Copy,Cu&t,&Paste]]&',
     1   'COPY','CUT','PASTE')
      i=winio@('%mn[&Help[&Contents,&Help on help]]&',
     1   'HELP_CONTENTS',help_file,
     2   '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 dispalyed ***
      i=winio@('%nl%nl      %^7bt[&Compile]&',compile)
      i=winio@('%nl%nl      %^7bt[&Link]&',   link)
      i=winio@('%nl%nl      %^7bt[&Run]',     run)
      end
C     *** Call-back functions that do nothing ***
      integer*4 function compile()
      compile=2
      end
      integer*4 function link()
      link=2
      end
      integer*4 function run()
      run=2
      end
#pragma windows 500000,500000
#include <windows.h>
int compile(){return 2;}
int link()   {return 2;}
int run()    {return 2;}
int main()
{
   char file[129],new_file[129];
   char help_file[]="myhelp.hlp";

   winio("%ca[Example]&");
   winio("%mn[&File[&Open,&Save,Save &As,E&xit]]&",
         "EDIT_FILE_OPEN",   "*.*",file,
         "EDIT_FILE_SAVE",   "*.*",new_file,
         "EDIT_FILE_SAVE_AS","*.*",new_file,
         "EXIT");

   winio("%mn[&Edit[&Copy,Cu&t,&Paste]]&",
                "COPY","CUT","PASTE");
```

```
   winio("%mn[&Help[&Contents,&Help on help]]&",
                "HELP_CONTENTS",help_file,
                "HELP_ON_HELP",help_file);

   winio("%ww%pv%30.10eb[vscrollbar,hscrollbar]&",NULL,0);
   winio("\n\n      %^7bt[&Compile]&",compile);
   winio("\n\n      %^7bt[&Link]&",link);
   winio("\n\n      %^7bt[&Run]",run);
}
```

The reminder of this chapter is taken up with a detailed explanation of how to set out the parameters of `winio`.

Call-backs

When a call-back function is complete it must return a value of either 0, 1 or 2. If a 0 is returned it will signal an exit and so terminate the window/program. A return value of 1 will cause a full update of the display so that any changes are made visible. This should be use sparingly because if a call-back is used repeatedly the whole display area will start to flicker due to the frequent screen updates. A more efficient approach is to use the remaining return value of 2. This will not cause any closure or update of the display. To make sure any part of the display that needs to be refreshed, a call to `window_update(&var)` will suffice. This will also improve the response of the entire system as only small section of the display will be redrawn. In summary return values can be:

0 Close associated window.

1 Return from call-back with display refresh.

2 Return from call-back with no refresh.

Window formats

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

```
int winio(const char* format, ... );
```

where the ellipses ... represents a list of arguments whose length depends on what appears in the character variable format. This list must only contain arguments of type `int`, `double`, or `char*`.

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

```
#pragma windows 500000,500000
#include <windows.h>
int main()
{
  double x=1.0,y=2.5;
  winio("X= \t%wf\nY = \t%wf",x,y);
}
```

The %wf *format code* presents the equivalent of the %f format in `printf` and gives a floating point value with six decimal places as the default. New lines are represented by \n, and tabs \t operate on a grid of 8-character intervals or by setting tab stops with the %tc format. Form-feed characters \f also have a special meaning. They will move down to a line 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.

By using the additional format codes described below, 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 <number> or <number>.<number>. <number> can be replaced by an asterisk "*" with the corresponding value(s) being supplied as one or two arguments of type `int*` 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.

Four special characters, known as *format modifiers* may be 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 formats to control the disabling (greying) of the control. The grave accent character (`) is also used to modify the action of certain codes. These characters should only be used with the codes which define them as described below.

The fourth character is the question mark (?) that can be used with any format which defines a control (as opposed to formats such as %ww which 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 in square brackets, or a "@" character is placed in the format string to indicate that the help string is supplied as an extra argument. A text string that can be supplied in either of these two forms is known as a *standard character string*, and is used in several format descriptors.

Where option-lists appear in format codes, these are always optional. However, if the option-list is empty and a help string is required, square brackets "[ ]" must be inserted to represent the empty list. These are then followed by the help string.

For example:

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

As an example of the help facility, in order to prompt the user for an integer (%rd) and augment the dialog box with an additional help string, one might write:

```
#pragma windows 500000,500000
#include <windows.h>
int main()
{
  int n_ch;
  winio("No of children %rd?rd"
        "[How many children have you got?]",&n_ch);
}
```

The following would produce the same result:

```
int n_ch;
char* help="How many children have you got?";
winio("No of children %?rd@",&n_ch,help);
```

Help strings can be spliced with line feed characters (\n) if necessary. The control to contain the help text will be sized to fit dimensions of the largest string. The help text can be positioned at another location (e.g. inside a box) using the %he format and/or used to supply bubble help using the %bh format.

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

Calls to `winio` can become very 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 and arguments are held over for another call to `winio`.

For example:

```
winio("Enter an integer      \t%rd\n&",&n);
winio("Enter a second integer\t%rd\n&",&n);
winio("%cn%`bt[Done]");
```

This creates a single window containing two integer edit boxes and 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.

The programming language that is used for the first call to `winio` (Fortran or C++) for a particular window must be the same as that used in any continuation calls for that window although different windows can be produced using different languages if desired.

Index of special format codes

The following table provides a list of the special format codes.

%ac Accelerator key format.
%ap Absolute positioning of next control.
%aw Attach window format.
%bc Button colour format - specifies the colour of the next %bt button.
%bf Switch to bold font.
%bg Background colour format.
%bh Bubble help format.
%bi Supply an icon for the next button.
%bm Bitmap format draws a bitmap.
%br Bar format - draws a horizontal or vertical bar which is partially filled with a user-selected colour.
%bt Button format - defines a button with text.
%bx Adds a raised grey bar to a tool bar.
%ca Caption format - defines the title of a dialog box.
%cb Box close format - closes a box opened by %ob.
%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.
%ch Child window format - inserts a child window.
%cl Displays a colour palette.
%co Control option format - used to modify subsequent %rd, %rf, and %rs boxes.
%cn Centring format - forces everything which follows it, up to the next new line or form-feed character, to be centred in the window.
%cu Establishes a cursor for the next control in a format.
%cv Control variable format.
%cw Embeds a *ClearWin* window.
%dc Establishes a default cursor for the window.
%dd Increase/decrease button for an integer.
%df Increase/decrease button for a floating point value.
%dl Allows a call-back function to be called at regular intervals via a timer.
%dr Drag and drop. Provides a call-back so that the `clearwin_string` can be examined.
%dw Owner draw box format - provides owner draw boxes.
%eb Edit box format - presents a edit box in which a text file can be displayed and modified.
%ff Form feed. Move down to below any existing controls.
%fl Floating point limit format - specifies the lower and upper limits for subsequent %rf formats.
%fn Font name format - selects a font for subsequent text.
%fr MDI frame format - defines a frame to contain child windows attached by %aw.
%fs File selection format - specifies the working directory and file filter for subsequent file open call-backs.
%ft File filter format - specifies filter information for subsequent file open call-backs.
%ga Gang format - enables radio buttons and/or bitmap buttons to be ganged together so that if one is switched *on* the others are switched *off*.
%gd Programmer's grid format - supplies a temporary grid to help with the positioning of controls.
%gf Get font handle format.
%gp Get window position format - used to get the current co-ordinates of the window position.
%gr Graphics format - provides a rectangular area for C/C++ graphics routines.
%he Help format - specifies the location of help information.

%hs Allow a horizontal scroll bar to be attached to a window. *See %vs*

%ht hypertext document attachment

%hw Obtain the window handle HWND

%ic Icon format - draws an icon.

%il Integer limit format - specifies the lower and upper limits for subsequent %rd formats.

%it Switch to italic font.

%lb List box format - Obsolete, use %ls

%ls List box format

%lw Leave window open format - allows `winio` to return without closing the window that it creates.

%mi Minimise icon format - supplies the name of an icon resource to be used if the window is minimised.

%mn Menu format - used to attach a menu to the window.

%ms Defines a multi-selection box that stores its settings in an array. *Similar to %ls*

%nd Never-down format - prevents controls from sliding down when sizing a window.

%nl New line

%ns Disable screen saver.

%nr Never-right format - prevents controls from sliding to the right when sizing a window.

%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.

%pd Can be used to insert a Sterling pound symbol '£' regardless of editor mode.

%pm Supplies a popup menu so that when the right mouse button is pressed in the main window a menu appears.

%pv Pivot format - used to create a pivotal point for any subsequent re-sizing of the window.

%ps Property sheet - layer windows to produce a card index style.

%rb Radio button format - defines a radio button with button text supplied directly.

%rd Integer input format - creates an edit box and displays an integer that can be updated.

%rf Floating point input format - creates an edit box and displays an floating point value that can be updated.

%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.

%rs Character string input format - creates an edit box and displays a character variable (i.e. a string) that can be updated.

%sc Start-up call-back. Causes a call-back to be called ONCE at program start-up, useful for drawing initial %gr data.

%sf Standard font format - resets to default text attributes after use of any combination of %bf, %it, %ul, %fn, and %ts.

%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.

%sl Produces a slider control using a floating point variable to determine its position.

%sm System menu format.

%sp Set window position format - used to initialise the position of the window on the screen.

%ss Allows the settings of selected controls to be auto saved and loaded from an INI file.

%st Variable string format - lays a string out in a field of *n* characters. The string is re-drawn each time the window is renewed.

%sv Allows the program to become a screen saver executable.

%sz Size window format.

%ta Insert tab 8-character intervals or to predefined tab stops. *See %tl*

%tb Bitmap button and toolbar format - defines a bit mapped button or a whole tool-bar.

%tc Sets the colour of subsequent text.

%tl Sets up tab locations for use with %ta and \t.

%ts Text size format - used to scale the text font size either up or down.
%tt Textual toolbar format.
%tv Hierarchical tree - type view.
%tx Displays an array of text in a rectangular region, defined by its parameters.
%ul Underline text.
%uw User window allows windows code to be interfaced to `ClearWin+`.
%vs Allows a vertical scroll bar to be attached to a window. *See* %hs
%wc Character output.
%wd Integer output.
%we Floating point output in exponent form.
%wf Floating point output in decimal form.
%wg Floating point output in decimal or exponent form.
%wp Supplies the name of a wallpaper bitmap which is used as a back drop to its contents.
%ws Character string output.
%ww Window control format - causes the resultant window to look like a normal application window, rather than a dialog window.
%wx Hexadecimal integer output.

Format codes

1)	Interactive I/O:	%co, %dd, %dl, %df, %dr, %fl, %il, %rd, %rf, %rs
2)	Controls:	%bc, %br, %bt, %co, %ga, %hs, %ms, %lb, %rb, %sl, %tb, %tt, %vs
3)	Bitmaps, Cursors, Icons:	%bi, %bm, %cu, %dc, %ic, %mi, %si
4)	Menus and accelerator keys:	%ac, %mn, %pm, %sm, %tv
5)	Layout and positioning:	%ap, %cn, \f, %gd, %gp, %nd, \n, %nr, %pv, %rj, \t
6)	Boxes:	%cb, %ob
7)	Displaying text:	%bf, %fn, %gf, %it, %pd, %sd, %sf, %st, %su, %tc, %ts, %tl, %ul, %ht
8)	Help:	%bh, %he
9)	Graphics:	%dw, %gr
10)	Main window attributes:	%ca, %cv, %lw, %ns, %sp, %sv, %sz, %wp, %ww
11)	Background colour	%bg
12)	Child windows:	%aw, %ch, %cl, %cw, %fr, %ps, %sh, %uw
13)	Formats for call-back functions:	%cc, %fs, %ft, %sc
14)	Edit boxes:	%eb, %ls, %pb, %tx
15)	Third party products:	%vb
16)	Data output:	%ss, %wc, %wd, %we, %wf, %wg, %ws, %wx

The heading for each format code includes the general form of the format. In this general form, square brackets are used to represent optional components. When the code is written, some components are themselves enclosed in square brackets. For example, *standard character strings* and option lists are enclosed in square brackets. Thus in the %br format the general form is:

%<n>br[<option list>]

where <n> represents an essential component and [<option list>] represents an optional component called <option list>. If multiple options are required then they must be separated by commas.

Interactive I/O

Reading numbers or character strings from a window under ClearWin+ is very similar to reading data in a traditional program using `scanf`. 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.

Those controls which allow your 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`'.

Control option format

%co<option>

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) on rd/rf/rs on loss of focus only. This is particularly valuable if numeric limits have been imposed by %il/%fl.
`data_border`	Put a border round rd/rf/rs boxes (default)
`full_check`	Perform checks and call-back on rd/rf/rs every change (default)
`no_data_border`	Omit border round rd/rf/rs boxes

For example:

```
winio("%co[no_data_border]%il%rd&",1,10,&j);
winio("\n%co[data_border]%il%rd",1,10,&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.

For example:

```
#pragma 500000, 500000
#include <windows.h>
int val=0, step=5;
int main()
{
  winio("%ca[Spin]%dd%`rd",step,val);
  return 0;
}
```

Floating point increase/decrease format

%df

Supplies an spin control button to go with a %rf edit box. %df takes one `double` 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.

Drag and drop

%dr

%DR Example

This format takes a call-back function which is called when a file is dropped onto a window. This can be achieved for example, by the following actions:

- Open the file manager ('Explorer' in Windows 95) and select a file from any disk/hard/CD drive.
- With the left mouse button still held down move the mouse pointer onto the ClearWin+ window and release the left mouse button. An image of the
- selected icon (or file) will follow the mouse cursor.

The `clearwin_string DROPPED_FILE` can then be interrogated to obtain the full path name of the file.

An text editor might use this by opening the file recently dropped onto it and displaying its contents for editing.

For example:

```
#pragma windows 500000,500000
#include <windows.h>
#include <string.h>
char drop_file[129]="no file";
int drop()
{
  strcpy(drop_file, clearwin_string("DROPPED_FILE"));
  window_update(drop_file);
  return 2;
}

int main()
{
    winio("%ca[Drag and drop]\nThe string should
           contain%dr&",drop);
    winio(" the path of the dropped
           file:\n\n[%`rs]\n",drop_file,65);
}
```

Floating point limit format

%fl

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

The lower limit should be specified with care unless `%co[CHECK_ON_FOCUS_LOSS]` is used. This is necessary because if, for example, the lower limit is 500 and the user desired to enter the value 525, it would be rejected as soon as the first digit 5 is entered. If `CHECK_ON_FOCUS_LOSS` is not used then values of 1 or less will remove this problem.

For example:

```
double x;
winio("%fl%rf",0.0,10.0,&x);
```

Integer limit format

%il

Specifies the lower and upper limits (as arguments of type `int`) for subsequent %rd formats. The lower limit should be specified with care unless `%co[CHECK_ON_FOCUS_LOSS]` is used. This is necessary because if, for example, the lower limit is 500.0 and the user desired to enter the value 525, it would be rejected as soon as the first digit 5 is entered. If `CHECK_ON_FOCUS_LOSS` is not used then values of 1 or less will remove this problem.

For example:

```
winio("%si*Enter a small positive integer: %il%rd",0,99,&p);
```

Integer input format

%[<n>]rd

Creates an edit box and displays the value of the corresponding (`int*`) 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). By default the edit box will be made large enough to hold integers of the current range, although the parameter *n* may be used to override this.

For example:

```
int p;
winio("%si*Enter an integer:%3rd",&p);
```

A call-back function can be supplied using the caret (^) modifier. This function is called whenever a change is made, and must return either a 0, 1 or 2.

A grave accent format modifier (`) may be used to make the control read-only. In this form no edit box is supplied, and the value displayed may only be changed by the program, using `window_update` to reflect the changes. The caret can be added to provide a call-back function that will be called on every change to the edit window.

Floating point input format

%[<n>]rf

Creates an edit box and displays the value of the corresponding (`double*`) 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). Numbers can be 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 parameter *n* may be used to override this.

A call-back function can be supplied using the caret (^) modifier. This function is called after the change has been made and all the dialog controls are refreshed when it returns.

For example, suppose you wished 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. All dialog controls are recalculated after a call-back function has been called, so it is easy to achieve the effect.

```
#include <math.h>
double X,Y,R,THETA;
int convert_to_polar()
{
  R=sqrt(X*X+Y*Y);
  THETA=(X==0.0 && Y==0.0)? 0.0:atan2(Y,X);
  return 1;
}
int convert_to_XY()
{
  X=R*cos(THETA);
  Y=R*sin(THETA);
  return 1;
}
main()
{
  winio("X = %^rf   Y= %^rf    R = %^rf   THETA = %^rf",
    &X,convert_to_polar,&Y,convert_to_polar,
    &R,convert_to_XY,&THETA,convert_to_XY);
}
```

A grave accent format modifier (`) may be used to make the control read-only. In this form no edit box is supplied, and the value displayed may only be changed by the program, using `window_update` to reflect the changes. The caret can be added to provide a call-back function that will be called on every change to the edit window.

The %co format code can be used to modify the way that %rf and %rd call-back functions behave.

Character string input format

%rs

%RS Example

This format takes one character variable and displays it in an edit box so that the user may update it. It also requires an argument to determine the length of the edit box in characters.

For example:

```
char str[80]="Initial string";
winio("%rs",str,10);
```

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

A grave accent format modifier (`) may be used to make the control read-only. In this form no box is displayed, and the string may 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. The caret can be added to provide a call-back function that will be called on every change to the edit window.

Button format

%[<n>]bt<button>

This format defines a button where <button> represents a *standard character string* consisting of the button text. The text must either be enclosed in square brackets, or an @ symbol can be 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 the button used to close the opened 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:

```
winio("Idiot!\t%bt@",”&Sorry");
```

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

The caret character (^) may be used to define a call-back function. The call-back function takes no arguments and returns an integer. If the return value is zero the window closes. Otherwise it assumes that the function has performed the action specified by the button and continues to wait for user interaction.

For example:

```
int func()
{
  do_something();
  return 1;
}

main()
{
  winio("Press this to see what happens %^bt[PRESS]", func);
}
```

Normally a button created with %bt is permanently enabled. However, if the tilde (~) format modifier is used then an extra `int*` argument must be supplied (before the call-back function if any). 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 menus.

For example:

```
#pragma windows 500000,500000
#include <windows.h>

int grey_control;
int open_func()    { grey_control=1;return 2;}
int save_func()    { grey_control=0;return 2;}
int save_as_func() { grey_control=0;return 2;}

int main()
{
     //Only the OPEN button is initially available
     grey_control=0;
     winio("%^bt[Open]  %~^bt[Save]  %~^bt[Save as]",
     open_func, &grey_control, save_func,
     &grey_control, save_as_func);
```

```
}
```

If the parameter *n* is supplied, this specifies the width of the button, rather than this being determined by its text. A button is created with enough room for at least *n* characters (possibly more, because of the proportional font). This enables a number of buttons to be created of the same size.

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

For example:

```
char button_name[]="Press\nhere";
winio("%bt@",button_name);
```

Button colour format

%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 with a call to `RGB()`.

For example:

```
winio("%ca[Button Colour]%bc[RED]%bt[Button A] %bc
%bt[Button B]",RGB(128,255,0) );
```

Button icon format

%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 "`#pragma resource`" (see %ic. 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!]
```

Bar format

%<n>br[<option list>]

This format draws a horizontal or vertical bar which is partially filled with a user-selected colour. The first argument is a `double*` control variable which indicates the extent of the fill. This variable should have a value between 0 and 1 inclusive. The second argument is of the standard Windows API COLORREF type, and is most conveniently created using the RGB(*r,g,b*) macro. *n* is used to control the length, in standard characters, of the bar 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.

By using the `window_update` function the bar will be updated. This should be called at sufficient intervals to provide a pleasant effect. This is illustrated in the following example:

```
#pragma windows 500000,500000
#include <windows.h>
int main()
{
     double bar_value;
     int window_ctrl=1;

     winio("%ca[Bar format]Processing .....\n\n%20br%lw",
         &bar_value,RGB(255,255,0),&window_ctrl);

     for(bar_value=0; bar_value<=1.0; bar_value+=0.003)
     {
       for(int i=0;i<100000;++i);  //do something here
       window_update(&bar_value);
     }
}
```

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

Tool bar border

%bx

This format will add a raised grey bar effect to any tool bar set at the top of the window. It takes one floating point argument that specifies the additional depth (in average characters) to be inserted around the bar.

Gang format

%<n>ga

Using this format, radio buttons and/or bitmap buttons (e.g. components of a toolbar) can be ganged together. This means that at any time only one control may 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. It specifies the number of controls to be ganged. The argument list must contain *n* `int*` arguments, that contain the *n* controlling integers. For example, in the toolbar example (see %tb), the three controls could be ganged together by modifying the code thus:

```
int b1=0,b2=1,b3=0;
winio("%3ga%3.1?^tb",&b1,&b2,&b3,"[Apples][Oranges][Pears]",
      "OFF1","ON1","DWN1",&b1,func1,
      "OFF2","ON2","DWN2",&b2,func2,
      "OFF3","ON3","DWN3",&b3,func3);
```

The ganging format can be placed before or after the formats in which the control variables are actually used. No variable should appear in more than one gang format. However, a particular variable can be used in more than one control. This format is most useful when used with %tb.

Horizontal scroll bars

%hs

Both %hs and %vs have identical meaning and usage except that one (%vs) generates a vertical scroll bar on the right-hand side of the window and the other (%hs) a horizontal scroll bar along the bottom of the window. Three arguments must be supplied, the first is an `integer` and determines the steps the slider control moves each time the left mouse button is pressed inside the scroll bar region i.e. a whole page. This is usually linked to the effect the page keys have on a text window. The second is the maximum value that the scroll bar can generate. The third argument holds the current value of the scroll bar which is of type `integer` and will be in the range 0 .. maximum - 1. A call-back function may also be provided if the caret modifier is used.

For example:

```
#pragma windows 500000,500000
#include <windows.h>
 int x,y;
 int update()
 {
  window_update(&y);
  window_update(&x);
  return 1;
 }

int  main()
 {
  winio("%ca[Scroll Bars]\n \tx info\t%`rd\t\n%^hs&",
           &x,300,1000,&x,update);
  winio("\n\ty info \t%`rd\t\n%^vs\n",&y,1,100,&y,update);
  return 0;
 }
```

List box format

%[<m.n>]ls (or %<n.m>lb now obsolete)

This format supplies a simple list-box facility. The corresponding arguments are a `char**` pointer to an array of pointers to strings, and an `int*` argument to both set the initial selection and to return the result. The array should be terminated with a `NULL` pointer.

For example:

```
char* greek[]={"alpha","beta","gamma",”delta”,NULL};
int selection=1;
winio("Select a Greek letter:  %7.3ls",greek,&selection);
```

The characters “7.3” before “ls” specify the width of the list box as a number of characters and the depth of the list box. Pre-setting *selection* to 1 would open the dialog box with “alpha” already highlighted. Selecting “beta” would return with variable *selection* set to 2. If *selection* is pre-set to zero, none of the items will be initially selected. If the user exited without making a selection, *selection* would keep its pre-set value.

The grave accent (`) may be used to produce a drop-down combo box rather than a list box. The caret (^) character may be used in association with a call-back function (supplied by the programmer) that will be called whenever an item is selected.

For example:

```
int ls_function(void);
winio("%^ls",array,&n,ls_function);
```

(See also the `LIST_BOX_ITEM_SELECTED` parameter in clearwin_info.

The %lb format, which has reverse parameters to %ls, is now obsolete and is only included for backwards compatibility.

Multiple selection box

%[<m.n>]ms

Sometimes it is required to select more than one item at once. This can be done using %ms. This is analogous to %ls except that the resulting variable is replaced by an integer array. The elements of this array should be initialised to zero for initially de-selected items and one for selected items. The array is updated as the user changes the selection. %ms requires two arguments the first being a pointer to an array of strings terminated by a `NULL` string and the second an array of integers. Both arrays, excluding the `NULL` terminator, must have the same number of elements.

The following program illustrates some typical cases:

```
#pragma windows 1000000 1000000
#include <windows.h>
int k;
char* things[]={"Apples","Bananas","Cherries",
               "Grapes”,”Oranges”,”Pears”, “Raspberries",NULL};
int ivec[7]={0,0,0,0,0,0,0};
main()
{
    ivec[1]=1; // select as on
    ivec[4]=1;
    ivec[5]=1;
    winio("%ca[Selecting things]%3tl&",15,30,45);
    winio("%tc[red]Simple\tDrop-down\tScrolling\tMultiple
selection\n%tc[black]&");
    winio("%ls\t&",things,&k);
    winio("%`ls\t&",things,&k);
    winio("%10.3ls\t&",things,&k);
    winio("%ms&",things,ivec);
    winio("\f\nNote that the first three boxes are coupled”
                “together\n&");
    winio("because they share the result variable k");
}
```

Radio button format

%rb<button name>

This format defines a radio button. The button name is a *standard character string*. It takes a single `int*` argument that is set to 0 or 1 according to whether the corresponding control is *off* or *on*. The grave accent (`) may be used to modify this control in order to use a check box.

For example:

```
int r1=1,r2=0,r3=0,r4=0,r5=1;
main()
{
  winio("%ca[Radio Buttons]%2`ga%rb[PCX]\t%rb[BMP]\n\n&",
           &r1,&r2,&r1,&r2);
  winio("Compress\n%3`ga%`rb[ HI ]\n%`rb[MED ]\n%`rb[LOW]",
           &r3,&r4,&r5,&r3,&r4,&r5);
}
```

The caret character (^) may be used in association with a call-back function (supplied by the programmer) that will be called whenever an item is selected. Radio buttons may also be ganged together (see %ga.

Slider format

%<n>sl[<option>]

%SL Example

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:

```
#pragma windows 500000,500000
#include <windows.h>
double value=0;
int  main()
{
  winio("%CA[slider test]\n\n%30sl[horizontal]\n\n&",
         &value,-1.5,10.5);
  winio("\n\nValue is %df%fl%rf\n\n",1.0,&value);
  return 0;
}
```

Bitmap button and toolbar format

%[<n[.m]>]tb

This format defines a bit mapped button or a whole tool-bar. The names of three bitmap resources representing the *off* state, *on* state, and *depressed* state should be supplied as `char*` arguments, followed by a pointer to an integer 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 terminate the dialogue. The correct argument format is as follows:

'BMPoff' , 'BMPon' , 'BMPdown' , ControlVariable , '<Call_back>'

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

For example:

'BMPoff' , 'BMPon' , 'BMPdown' , 'BMPdisabled' , ControlVariable, GreyControlVariable , '<Call_back>'

A call-back function may also be supplied (using the format modifier ^). If the call-back function returns zero, the dialogue terminates and returns zero to the caller.

In order to obtain the effect of a non-toggling button, simply specify the same button 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. Here is a simple example of the use of bitmap buttons:

```
#include <windows.h>
main()
{
  int answer;
  winio("Hello\t%tb","BT_OFF","BT_ON","BT_DOWN",&answer);
}
#pragma resource
BT_OFF BITMAP  "BT1.BMP"
BT_ON BITMAP   "BT2.BMP"
BT_DOWN BITMAP "BT3.BMP"
```

The corresponding bitmap files can be created by any paint program that can produce the .BMP format.

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 pair 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 defaulted to the *on* state.

```
{
  int b1=0,b2=1,b3=0;
  winio("%3.1?^tb[Apples][Oranges][Pears]",
       "OFF1","ON1","DWN1",&b1,func1,
       "OFF2","ON2","DWN2",&b2,func2,
       "OFF3","ON3","DWN3",&b3,func3);
}
#pragma resource
OFF1 BITMAP "BT1.BMP"
. . . .
```

Bitmap buttons may be ganged (see %ga). A greyed line may also be drawn around the tool bar.

Textual toolbar format

%tt<button>

%TT Example

This format operates in the same way as the button format, %bt, except that the button is thinner and 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:

```
#pragma windows 500000, 500000
#include <windows.h>

int main()
{
 winio("%ca[Textual Tools]&");
winio("This example shows how several %%tt\n&");
winio("controls can be placed into a status box.\n \n &");
winio("%ob[status,thin_panelled]&");
winio("%tt[Compile]%tt[link]%tt[Run]%cb");
  return 0;
 }
```

Bitmap format

%bm[<bitmap name>]

This is similar to the icon format %ic but the picture is supplied as a simple bitmap. For example we could change the icon example above with the line:

```
winio("This is a bitmap: %bm[tea_bitmap]");
```

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

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

As with the icon format, the grave accent can be used to specify that the bitmap is supplied by handle rather than name.

For example:

```
HBITMAP h_bitmap=LoadBitmap(NULL,OBM_DNARROW);
winio("%`bm",h_bitmap);
```

A call-back function can be added to a bitmap with the inclusion of the (^) format modifier.

Cursor format

%<n>cu[<cursor name>]

%CU Example

By default the mouse cursor is represented by an arrow except in graphics regions, where 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 a suitable string is supplied in the arguments). The string should be the name of a `CURSOR` resource. By using a grave accent (`), you can use the constants representing the standard Windows cursors.

For example:

```
winio("%`dc%ob%`cu%gr%cb\f\n%cn%`bt[OK]",
                  IDC_IBEAM,IDC_WAIT,100,100);
```

These two formats also have a more complex form in which several alternative cursors are supplied together with an integer used to select which one to use. The integer should be in the range 1 to the number of cursors.

For example:

```
     int k=1;
   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 can be used in a similar way. For example, using this mechanism, a graphics region can arrange to change its cursor when it changes its selection mode.

Default cursor format

%<n>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 format 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:

```
winio("%`dc",CURSOR_IBEAM);
```

Whilst for a user defined cursor:

```
winio("%dc[my_cursor]");
```

combined with a resource script or `#pragma resource` that uses the line:

```
my_cursor CURSOR cursor.cur
```

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

See the description of %cu for a further example of how %dc is used and how several different cursors can be used for different regions of the screen/window.

Icon format

%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 window.

For example:

```
#include <windows.h>
int main()
{
  winio("This is an icon: %ic[clear_win]");
}

#pragma resource
clear_win ICON clearwin.ico
```

The (^) character may be used to attach a call-back function to an icon. The function is called each time the user clicks on the icon. If the function returns zero, the window is closed.

If a grave accent is used, the icon is supplied as an icon handle argument as follows.

```
HICON h_icon=LoadIcon(NULL,IDI_QUESTION);
winio("%`ic",h_icon);
```

Icons have one special difference from bitmaps; they can have a transparent region which makes the image appear as if it is sitting on the background in the same way that text does. Its is also possible to include icon data in the program code with the make_icon routine.

Minimise icon format

%mi<icon name>

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

For example:

```
main()
{
  winio("Don't be late for tea!%mi[clear_win]%ww");
}
#pragma resource
clear_win ICON clearwin.ico
```

Standard icon format

%[<n>]si<symbol>

Defines a standard icon that is to be placed to the left of the block of text that follows the descriptor. The symbol may be "**!**", "**?**", "**#**", or "*****" (**#** specifies a stop sign and * specifies an information sign).

For example:

```
winio("%2si?%2si*%^2si!%2si#\n\nThese are all the standard       icons.
\n\n%cn%7bt[OK]",cb_activeicon);
```

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 a tasteful gap between the icon and subsequent text. This reproduces the appearance of a **MessageBox** window. The correct sound can be added by attaching the "`BEEP`" call-back.

To add the correct warning sound to the standard icons call the Windows API function **MessageBeep** before drawing the box.

The '^' character may be used to attach a call-back function to an icon. The function is called each time the user clicks on the icon. In the above example the '!' icon has the call-back function `cb_activeicon`. If the function returns zero, the window is closed.

Accelerator key format

%ac<key>

Format %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:

```
int call_back_func(void);
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.

Menu format

%mn<menu specification>

%MN Example

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

```
%mn[&Alpha,&Beta[Beta1,Beta2,|,Beta3],&Gamma]
```

This format specifies five selectable items, three of which, *beta1*, *beta2*, and *beta3* are contained in a pop-up sub-menu. A pipe symbol (|) separates *beta2* and *beta3*. This places a horizontal separator bar in the menu at this point. The three top-level menu items are selectable using accelerator keys Alt-A, Alt-B and Alt-G which are specified by the inclusion of the ampersand character (&) placed to the left of the character chosen to be the (unique) accelerator. Pointers to five call-back functions are required as arguments.

The corresponding call-back function is called when a menu item is selected. The call-back function takes no arguments and returns an integer result. If it returns zero, the window is closed. If it returns 1, the window is updated (to react to any changes to the contents of edit boxes etc.).

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 `int*` argument to point to a control integer. When 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:

```
#pragma windows 500000,500000
#include <windows.h>
int squash_func() {return 1;}
int beer_func()   {return 1;}
int tea_func()    {return 1;}
int main()
{
  int drinker=0;
  winio("%mn[Squash,~Beer,Tea]",
        squash_func, &drinker, beer_func, tea_func);
}
```

In the same way as the tilde controls the greying of a menu item, the hash character (#) is used to control the placement of a check mark (tick) at the front of a menu item. It also uses a control variable in the same way as the tilde (~). Top level menu items cannot be checked.

An accelerator key may be associated with a menu item and its corresponding call-back function as illustrated in the following example.

```
winio("%mn[File[&Open\tCtrl+F12]]",open_func);
```

The tab character (\t) is followed by the key name. Valid key names are illustrated with the accelerator key format %ac above. This is only permitted in sub-menus and should not be used in top level menus.

To continue the menu `%mn[Test,File[Open]]` with the menu `%mn[[Save],Help]` will produce a menu `%mn[Test,File[Open,Save],Help].` The continuing menu can start with as many open square brackets as required, but an error will be generated if you try to append to a menu item which is not a submenu!

Dynamic Menus

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

```
winio("%mn[&Window[*]] &",hndl)
```

The (*) character makes the %mn provide a handle which can then be passed to either add_menu_item or remove_menu_item. 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").`

Popup menu format

%pm<menu specification>

%PM Example

This format is almost identical to %sm and %mn except that it is activated when the right mouse button is pressed in the window. For a description of the correct format see %mn. In the example below a popup menu controls the mathematical operation carried out on the two values that are provided by the slider controls. The divide option is greyed out whenever either of the two sliders are at zero. An attempt to divide by zero would cause a processor exception. This has been included to show you how to use the tilde (~) character. The popup menu is visible in the top right hand corner of the image included below. It has been placed there so as not to obscure any other controls in the image.

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdlib.h>
double v1=0,v2=0,sum=0,mode=0;
int z=1;

int cbadd()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=2; return 2;
}

int cbsub()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=3; return 2;
}

int cbmul()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=0;  return 2;
}

int cbdiv()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=1;  return 2;
}

int math()
{
 switch( mode )
 { case 0: sum=v1*v2; break;
   case 1: if ( (v1==0.0) || (v2==0.0) ) { z=0; break; }
           sum=v1/v2; break;
   case 2: sum=v1+v2; break;
   case 3: sum=v1-v2; break;
 }
```

```
 window_update(&sum); return 2;
}

int  main()
{
winio("%ca[Popup]\n %10sl[vertical]      %10sl[vertical]
\n&",&v1,0.0,100.0,&v2,0.0,100.0);
winio("\n  Value 1 is %df%rf \n\n  Value 2 is %df%rf&",
         0.1,&v1,0.1,&v2);
winio("\n\n The sum is %^tt[Do Math]  %`rf&",math,&sum);
winio("%pm[Multiply,~Divide,|,Add,Subtract]",cbmul,&z,cbdiv,cbadd,
       cbsub);
return 0;
```

System menu format

%sm<menu specification>

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

For example:

```
winio("%sm[Alpha,~Beta]",Alpha_func,&Beta_cntrl,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_cntrl* is zero. *Alpha_func* and *Beta_func* are the associated call-back functions. If a grave is supplied in the form %`sm, then the two menu items will replace the system menu.

A tab (\t) character may be used to attach an accelerator key to a particular menu item and its associated call-back function. The syntax is the same as that for %mn above.

Tree view

%tv

The %tv format can be used to construct a hierarchical list box, which is analogous to the type of control sometimes used to display a directory structure. This takes a `char*` array of strings terminated by a NULL pointer. The final argument is an `int*` which is used to receive the index of the chosen item. The control operates similarly to the 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:

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

As you can see, the children of a node (together with any of their children, etc) are placed immediately beneath the object. In the example shown the types of nuts would not immediately be shown because of the ‘C’ in ‘BCNuts’. Nodes can be expanded and contracted by the user simply by clicking on them. 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 seem to start from where it left off. You could even store the strings in a file so that the display would persist in appearance from run to run.

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. Empty strings have the same effect. If a node has no children you can code '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 fall by any amount.

%tv can take a call-back function. This can look at the result variable and also interrogate the `clearwin_info` variable `TREEVIEW_ITEM_SELECTED`. This will be 1 if and only if the call-back is responding to a double click event.

By default the treeview display uses bullet marks to the left of the labels. These can be replaced by icons of your own choosing. By using a grave accent on the format these can be replaced by user-selectable icons. If a grave accent is used, then a third character is removed from the string. It is interpreted as an index ('A' - 'Z') into a list of icons. This list is supplied as an extra argument, and should consist of a list of icon resources separated by commas (e.g. 'ICON1,ICON2,ICON3'). Only small images are required, you should construct icons so that the image is confined to the top left 16 x 16 corner. The rest of the icon must be filled with the transparent colour. Note that these icons mimic the small icons that are available under Windows 95. However, this code will work equally well under Windows 3.1 or Windows NT.

By providing a call-back which examines the expansion indicator (character two of each string) it is possible to change the icon index to reflect whether the icon is expanded or not. 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 textview controls:

```
#pragma windows 500000 500000
#include <windows.h>
```

```
#include <string.h>

char* contents[]={     strdup("ACABook"),
strdup("BCAChapter 1"),            strdup("CCASection 1.1"),
strdup("CCASection 1.2"),          strdup("CCASection 1.3"),
strdup("CCASection 1.4"),          strdup("BCAChapter 2"),
strdup("CCASection 2.1"),          strdup("CCASection 2.2"),
strdup("CCASection 2.3"),          strdup("CCASection 2.4"),
strdup("BCAChapter 3"),            strdup("CCASection 3.1"),
strdup("CCASection 3.2"),          strdup("CCASection 3.3"),
strdup("CCASection 3.4"),          strdup("BCAChapter 4"),
strdup("CCASection 4.1"),          strdup("CCASection 4.2"),
strdup("CCASection 4.3"),          strdup("CCASection 4.4"),
strdup("BCAChapter 5"),            strdup("CCASection 5.1"),
strdup("CCASection 5.2"),          strdup("CCASection 5.3"),
strdup("CCASection 5.4"),          strdup("BCAChapter 6"),
strdup("CCASection 6.1"),          strdup("CCASection 6.2"),
strdup("CCASection 6.3"),          strdup("CCASection 6.4"),
strdup("BCAChapter 7"),            strdup("CCASection 7.1"),
strdup("CCASection 7.2"),          strdup("CCASection 7.3"),
strdup("CCASection 7.4"),          strdup("BCAChapter 8"),
strdup("CCASection 8.1"),          strdup("CCASection 8.2"),
strdup("CCASection 8.3"),          strdup("CCASection 8.4"),
strdup("BCAChapter 9"),            strdup("CCASection 9.1"),
strdup("CCASection 9.2"),          strdup("CCASection 9.3"),
strdup("CCASection 9.4"),          strdup("BCAChapter 10"),
strdup("CCASection 10.1"),         strdup("CCASection 10.2"),
strdup("CCASection 10.3"),         strdup("CCASection 10.4"),
strdup("BCAChapter 11"),                 strdup("CCASection 11.1"),
strdup("CCASection 11.2"),         strdup("CCASection 11.3"),
strdup("CCASection 11.4"),         NULL};
int item=6;

int test()
{
//   Call-back function sets the icon for each
//   object according to whether it is
//   expanded or not
     char* str=contents[item-1];
       if(*(str+1)=='E')  *(str+2)='B';
                  else
                        *(str+2)='A';
       window_update(contents);
    return 2;
    }

main()
{
```

```
  winio("%ww%ob%pv%^`20.15tv%cb\f\n%cn%`bt[OK]",
     contents,&item,"closed_book,open_book",test);
}

#pragma resource
closed_book icon book1.ico
open_book   icon book2.ico
```

Layout and positioning

By default the text and controls in a window are laid out in the order and position in which they are presented in the format string (i.e. the first argument of `winio`). \t is used as a tab control and \n produces a new line in the window. Also \f moves down to a line beneath any existing controls. The window is automatically constructed with a size to suit its contents. Note that %ww and %eb have a `no_border` option which removes the blank border that appears by default in these windows. %ww modifies the main window only.

Additional layout control is available by using the format codes in this sub-section.

Absolute position format

%ap

This format 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. This format should be used sparingly. For many purposes the centring, right justifying, etc. formats are more convenient. 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 undesirable.
- If a control is placed to the right of or beneath a pivot point, the supplied position will be the position corresponding to the default window size. The control will move if the window is re-sized in the usual way.

Centring format

%cn

This format forces everything that follows it (text, buttons, formats etc.), up to the next new line (\n) or form-feed character (\f), to be centred in the window. It is often used in conjunction with buttons.

For example:

```
winio("What shall I do ?\n\n%cn%bt[Continue] %bt[Give up]");
```

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

For example:

```
winio("%`cnProgram to test prime numbers \n\nWritten by Joe"
        "Bloggs\n\n%`bt[OK]")
```

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

Programmer grid format

%gd

This format is for **program development only**. It overlays a grid over the window and any controls within it. The result is ugly but useful to enable 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.

```
winio("%`cnProgram to test prime numbers&");
winio("\n\nWritten by Joe Bloggs\n\n%`bt[OK]");
```

Never down and never right formats

%nd, %nr

These formats are used with %ww and %pv and have the effect of fixing controls that otherwise move when a window is sized. %nd prevents subsequent controls from sliding down whilst %nr prevents controls from sliding to the right. The grave accent format modifier can be used with these two formats in order to cancel the effect for controls that follow.

Pivot format

%pv

%PV Example

If the %ww format is specified, the resulting window can be re-sized by the user, either by dragging the edges, or by using the maximise button. The %pv format marks a pivot point so that items to the right of this point move to the right as the window is widened, and items below the pivot point move downwards as the window is lengthened. If the formatted item to the immediate right of the pivot format is a multi-line text-edit box (%eb), then this item will expand to fill the extra space made available.

In the original ClearWin+ specification you could create windows which would re-size without any control changing its dimensions to use the new size. This happened if you used %ww but either did not use a pivot, or applied a pivot to a control which could not use it. This could result in many bizarre effects, and does not produce anything useful, so we changed the specification slightly. To obtain the old specification you should call `set_old_resize_mechanism()`.

The new scheme is as follows: In the absence of a pivot a window will not re-size (except to minimise). A pivot may only be placed on a control which can actually 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):

%eb	Edit boxes
%cw	Embedded ClearWin+ windows
%dw	Owner draw box
%fr	Frame for MDI child windows
%gr	Graphics box, must have metafile_resize or user_resize property.
%ht	Hypertext
%lb & %ls	List box controls (but not dropdown boxes with grave accent)
%ms	Multiple selection boxes
%tv	Tree view
%tx	Text array
%uw	User defined window

Right justifying format

%rj

This format forces everything that follows it (text, buttons, formats etc.), up to the next new line (\n) or form-feed character (\f), to be right justified.

Get and set window position formats

%gp, %sp

The %gp format takes a pair of 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. %sp can also be used for child windows. Each time the window is moved or re-sized the *x*, *y* pair is updated.

For example:

```
#pragma windows 500000,500000
#include <windows.h>
int x,y;
int myfunc()
{
//   Window will be positioned relative to the button
//   control in the main window
     winio("%spHidden!",x-5,y-5);
     return 1;
}
int main()
{
  winio("Press this button to conceal it! %gp%^bt[Press]",
           &x,&y,myfunc);
}
```

Tab position

%<n>tl

This format allows you to change the default tab character distance. It is specified in average characters widths. For example %4tl would take four integer arguments specifying the first four tab positions as multiples of the width of an average character. Subsequent tab positions would follow the default scheme. A complicated window layout may contain several %tl formats. Each %tl format takes effect for subsequent controls and cancels any previous one.

As explained above, tab positions are based on the width of an average character before any changes in font size. While this is fine enough for most purposes, it is occasionally useful to specify tab positions more precisely. To this end, if you use the grave accent modifier, then the tab positions are specified as `double` quantities and can be fractional.

Box close format

%cb

Closes a box opened by %ob.

Box open format

%<n,m>ob[<option list>]

%OB Example

This format can be used to create boxes, a status bar or grids filled with controls. It defines the top left hand corner of a rectangular box. Subsequent objects will be placed to the right and beneath this point until the box is closed with a %cb format. Every open box must be closed before the format (plus any continuations) is complete. A grave accent can be used to indicate that the box and all controls within it should have a darker (shaded) background. The %ob format may be followed by options enclosed in square brackets.

The options available are:

`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)
`no_border`	Create an invisible box which is useful for grouping controls.
`panelled`	This replaces the simple line box with a 3-dimensional panel.
`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.
`thin_panelled`	Similar to panelled but perhaps a slightly more attractive effect.
`scored`	Double line, 3-dimensional shade effect.
`depressed`	Single line, 3-dimensional shade effect.
`raised`	Single line, 3-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.

A name that is to be spliced into the top line of the box is provided as a standard character string.

For example:

```
%ob[named_c][Title goes here].
```

Alternatively the %ob control can take numerical parameters to create a grid. For example %2.3ob would create a grid two items across and three down. The contents of each grid component can be arbitrary and are terminated with a %cb. Thus the above case would require six %cb formats following it to close each component of the grid. Note that if you place %rd controls (say) in the grid, you may well want to use %co[no_data_border] to remove the superfluous data border from the control. %ob can take additional options. Most box options are available however, the box cannot be named.

For example:

```
#pragma windows 500000,500000
#include <windows.h>
     char string1[]="Salford Software Compilers";
     double v


al=1.0;
```

```
int main()
{
     winio("%ca[multipleregions]%ww[no_frame]&”);
     winio(“%2.2ob[panelled]&”);
     winio(“ %7bt[one] %cb %7bt[two] &”);
     winio(“%cb %7bt[three] %cb&”);
     winio(“%10sl[horizontal] %cb\n&",&val,1.0,10.0);
     winio(“%2.1ob[status,thin_panelled]%`4rf%cb%`rs%cb",&val,
     string1,17);
  return 0;
}
```

Note in particular the following possibilities:

- Controls may be easily set out in a grid using the invisible option on a multiple box.
- A status bar can have multiple horizontal components (multiple vertical components will cause an error) to create a segmented status bar. A panelled status bar probably looks best.

Font format

%fn[font name]

%fn (font name) is used to select a font for subsequent text.

For example:

```
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 standard system font. %sf is used to reset all text attributes to the default state i.e. the standard font.

Get font format

%gf

%gf takes one `int*` variable that receives the current font handle. This handle may then be used by any Windows API function that requires a font handle.

For example:

```
int h_font;
winio("%ts%gf",2.0,&h_font);
```

Hypertext

%<n,m>ht

Hypertext windows provide a way of building very easy to use GUI applications. A hypertext window contains text with hypertext links, which enable your users to flip between pages by clicking on highlighted text, images or initiate parts of your program. A hypertext window is created using %ht and can be surrounded by other controls as required. The text itself is supplied as an ASCII file containing a subset of the HTML mark-up codes, commonly used to create documents for the World Wide Web.

A grave accent (`) can be supplied to ensure that the call-back will be called the first time and each time the hypertext document changes. It will also be called every time one of the following `clearwin_string` are set:

CURRENT_TEXT_ITEM, CURRENT_TEXT_DOCUMENT, CURRENT_TEXT_TITLE

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 document called 'introduction'. This would be defined as a resource thus:

```
start HYPERTEXT "myfile.htm"
```

If you compile your resources via the Salford resource compiler, SRC version 1.95 or later must be used (the Microsoft RC compiler will not work for this). The file BROWSE.HTM, which is built into the ClearWin+ browser program, which is available as an example program, is an excellent example 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 which pertains until cancelled by a corresponding mark-up starting with a '/' character. For example, bold face text is preceded by <B> and followed by </B>. Special characters (such as the diamond brackets used to define mark-up codes) are encoded as follows:

<	or	<	produces	‘<‘ character
>	or	>	produces	‘>‘ character
&	or	&	produces	‘&’ character

Hypertext files may contain portions of text which 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. The following HTML codes are currently available:

<TITLE> ... **</TITLE>** - This supplies a title to a document.

This will be placed in the caption of the window using the %ht format.

To supply a title as part of the text use **<H1>** ... **</H1>** - This defines a principle heading to a document. The text will be enlarged and centred. Although the other heading styles (H2 .. H6) can be used, they currently operate the same way as H1. In future versions of the software this will change.

<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 documents. Each document should be surrounded by these tags and contain a DOC tag.

<DOC name="doc_name"> - This defines the document name, and has no analogue in HTML. Each document should have a distinct name enclosed in double quotation marks as shown.

<a HREF="name"> ... **** - This is an HTML anchor which encloses text (or an image) which 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 should either be another document name (defined by DOC), or the name does not correspond to a document and if the %ht format supplied a call-back function, that function will be called. The function can determine the textual value on the anchor by calling `clearwin_info`. In this way, it is possible to execute code in response to a mouse click.

<P> - End of paragraph. Subsequent text will start on a fresh line with a small gap separating it from the previous text.

**
** - on the other hand, will force a line break with no extra gap.

<HR> - Rules a horizontal line across the screen.

<B> ... **</B>** - Applies bold face to text.

<U> ... **</U>** - Underlines 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 may be modified by nesting a colour change inside the link). These are not standard HTML mark-ups.

<IMG SRC="name" ALIGN="align_option"> - Using this mark-up you can insert a bitmap into your text. "name" should be the name of a BITMAP resource you have added is the list of included resources (quotation marks are required), and align_option should be one of TOP, MIDDLE, BOTTOM. The alignment specification is optional, 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 which responds to mouse clicks.

<UL> ... **</UL>** - Text embedded between these mark-up codes is considered to form an unordered list. Each list item is preceded by a **<LI>** mark-up and will start on a fresh line and a bullet mark. Lists embedded in lists will nest to the right. The codes **<OL>** ... **</OL>** will produce an ordered list numbered from 1.

<IF name> ... **</IF>** - See Conditional Hypertext.

A window may contain at most 1 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. The reason why a window may only contain one hypertext control is that the window automatically provides certain services to the hypertext 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 call-back function as available within a window containing hypertext, thus:

"PREVIOUS_TEXT" Causes the hypertext window to 'go back' one link. A menu item attached to this call-back will be automatically greyed if the hypertext is at the top level.

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 document (as supplied by the **<TITLE>**) mark-up separated by a '-'.

Conditional Hypertext

The ClearWin+ browser program itself 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 an integer parameter, "Fortran", set up using the `set_clearwin_info` function. The first case 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) all use `window_update` to ensure that the hypertext is updated.

To get a working example of how this format works, open the `browse.htm` file and the associated program `browse.cpp`.

For example:

```
// in the main program & only called once
     add_hypertext_resource( "HYPERHELP" );
```

```
int Callback_hypertext_help()
{
     winio("%ca[Hyper text help system]%bg[grey]%ww%pv%^60.22ht@"
           ,"INTRODUCTION",CBhypercallback);
     return 2;
}
...

#pragma resource
HYPERHELP HYPERTEXT “helpfile.htm”
```

Character formats

%it, %bf, %ul

These formats are used to begin italic, bold face, and underlined text respectively. They all take the grave accent modifier (`) which is used to terminate the corresponding effect.

For example:

```
winio("Uses %bfbold face%`bf for emphasis");
```

%sf (standard font) can be used after any combination of these formats in order to reset to the default state. %sf may also be used after %fn and %ts.

Sterling pound symbol

%pd

%pd will ensure that a pound '£' symbol appears correctly on the chosen output device. This is required because of the differences between OEM and ANSI character tables which often lends to confusion.

Subscript format

%sd

Before this format code can be used a font must be activated since it will not have any effect on the default font (see %fn). All text then following this format will be in subscript style. To deselect this mode a grave accent (%`sd) can be use to return to the previous font setting.

For example:

```
winio("%fn[arial] Water is H%sd2%`sdO");
```

Standard font

%sf

Standard font format resets to default the text attributes after use of any combination of %bf, %it, %ul, %fn, and %ts. This format may also take a grave accent (`) but this only has effect is the program is running under Windows 95. In Windows 95 it selects the scaleable font that is used in menus and controls. %ts can be used after %`sf but only if the program is running under Windows 95. To make sure that it is safe to call %`sf a call to the function `WINDOWS_95_FUNCTIONALITY` should be made. This function will return with the value 1 if it is running under Win95. For continuous text you may want to follow %`sf with %ts and a scale factor of about 1.15 to produce comfortably readable text.

Superscript format

%su

Before this format code can be used a font must be activated as it will not have any effect on the default font (See %fn). All text following this format will be in superscript format. To deselect this mode a grave accent (%`su) can be use to return to the previous font setting.

For example:

```
winio("%fn[arial]Einstein says E=MC%su2%'su.");
```

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

Text size format

%ts

%ts takes one `double*` 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 in the range from 0 > *n* < 1.0 are used to scale down, whilst values greater than 1.0 are used to scale up. Note that the standard system font can not be scaled so %fn should always be used before %ts.

For example:

```
winio("%fn[Courier New]%tsThis is larger",1.5);
```

Text colour format

%tc[<colour>]

By default a window gives text the colour scheme that the user can select from the Control Panel icon of the Program Manager. This colour can be determined by a Windows API call of the form `GetSysColor(COLOR_WINDOWTEXT)`. The %tc format specifies the text colour for subsequent text (until another %tc overrides it). It takes one integer argument which is typically created using Windows API **RGB** macro.

For example:

```
winio("%tcRed %tcGreen %tcBlack",
        RGB(255,0,0),RGB(0,255,0),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 for details of how to do this). An argument of -1 resets the colour to the system colour selected from the control panel.

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

Variable string format

%<n[.m]>st

This format takes one `char*` argument and lays the string out in a field of *n* characters. In contrast to the %s format descriptor in the `printf` function, the string is re-drawn 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 to force a change to become visible. The variable string format is equivalent to %rs with a grave accent.

Help

Details are given of the "?" help format modifier. %bh and %he can be used to control the manner in which a help string is presented.

Bubble help format

%bh

This format takes an `int*` argument pointing to a control variable. When this variable is non-zero, help text is supplied to the user in the form of a "bubble" whenever the mouse cursor lies over the control in question. This provides a very clear form of help. To enable your users to switch such help *on* and *off*, simply supply a button with a call-back function which toggles the integer control variable.

For example.

```
int help_switch=0;
int toggle();
main()
{
  char hlp[]="Toggles bubble help On/Off";
  winio("%^?bt[Start]@%bh",toggle,hlp,&help_switch);
}

int toggle()
{
  help_switch=(help_switch+1)&1;
  return 2;
}
```

A grave accent can be added which will have the effect of generating a short delay before the help bubble appears.

Help format

%[<n[.m]>]he

This format specifies the location of the help information (associated with the "?" format modifier). Normally the information is simply printed at the bottom of the window. However, using this format the help can (for example) be placed inside a box. In the form %he (without the integer modifiers), the control will be sized to fit the largest help string yet encountered. If further help strings are supplied after the %he format has been encountered, their length should not exceed this size. In practice help boxes are almost always placed at the bottom of a window. However, if necessary, an earlier help string could be padded with blanks to increase its size.

%20he fixes the size of the control 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.

```
int n_ch=0;
char str[]="No of children:";
char hlp[]="How many children have you got?";
winio("%21st%?rd@\n\n%ob%he%cb",str,&n_ch,hlp);
```

Graphics

There are currently two ways of displaying graphics in Cleawin+. The first (%dw) 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.

The second method (%gr) is probably the most useful. This format enables code using Salford graphics primitives (such as `draw_line`) to work with Windows. This method eases the transition of DOS programs to Windows.

Owner draw box format

%dw[<option list>]

This format is used to provide owner draw boxes (graphics). The format takes one argument which is a device context acquired by calling the ClearWin+ function `get_bitmap_dc`. This device context accesses a bitmap that can be written using standard graphics Windows API calls. 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:

```
#include <windows.h>
...
HDC my_dc=get_bitmap_dc(50,50);
MoveTo(my_dc,0,0);
LineTo(my_dc,50,50);
winio("%dw %bt[OK]",my_dc);
// The next line is not necessary unless
// many device contexts will be acquired
release_bitmap_dc(my_dc);
```

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).

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 of course 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, without worrying about deleting used pens. This is performed by calling the ClearWin+ change_pen function. 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 can contain only:

`user_resize`	%dw can take the option user_resize, which allows it to be re-sized as the pivot item of a window. In this form, %dw does not take a handle from `get_bitmap_dc` 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 `clearwin_info` parameter `GRAPHICS_RESIZING` 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 please note: 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.
`full_mouse_input`	The call-back function (if supplied by the programmer) is called each time the mouse is moved, or a button is pressed or released within the region of the control. Normally the call-back function is only called when the

user clicks on the control. This option makes it easy to produce image editing programs. For example:

```
winio("%^dw",my_dc,dw_callback);
```

To obtain the location of the mouse and other information from within the ClearWin+ call-back function call get_mouse_info.

Graphics format

%gr[<option list>]

%GR Example

%GR Example 2

%GR Example 3

Option list can be any valid combination of the following:

[BLACK, BLUE, GREEN, GREY OR (GRAY), RED, YELLOW, METAFILE_RESIZE, USER_RESIZE, RGB_COLOURS, FULL_MOUSE_INPUT, BOX_SELECTION, LINE_SELECTION, USER_SURFACE]

If a program contains calls to Salford graphics routines and the "#pragma windows" compiler directive (or its equivalent) is used, then by default a simple *Format* window will be automatically created by ClearWin+. This is analogous to the automatic creation of a *ClearWin* window for standard I/O via printf etc. Normally, however, the programmer will prefer to have direct control of the window by an explicit call to winio using the %gr format code as follows.

This format takes two arguments that provide the pixel width and height of a rectangular area. Subsequent calls to Salford graphics functions are drawn in this area. The option *black* initially paints the area black.

For example:

```
int ctrl;
  winio("%gr[black]%^bt[OK]%ww%lw",400,300,"EXIT",&ctrl);
  draw_line(100,100,300,200,15);
```

Otherwise the default background colour is used to paint the area.

If more than one screen area is to be drawn to then a grave accent modifier is supplied to %gr and a third argument is required in the form of a user defined handle for the area. The ClearWin+ subroutine select_graphics_object is provided to work with %gr in this context.

The following example creates two graphics areas and draws to each in turn.

```
int ctrl,hnd1=1,hnd2=2;
winio("%`gr&",400,300,hnd1);
winio("%`gr&",400,300,hnd2);
winio("%ww%lw",&ctrl);
select_graphics_object(hnd1);
draw_line(100,100,300,200,15);
select_graphics_object(hnd2);
fill_ellipse(200,150,75,50,2);
```

In a similar way, it is possible to draw to a screen area in parallel with output to a graphics printer or plotter. In the following example the second handle is attached to a graphics printer or plotter.

```
int ctrl,hnd1=1,hnd2=2,j,k;
 winio("%`gr&",400,300,hnd1);
 winio("%ww%lw",&ctrl);
 j=select_graphics_object(hnd1);
 draw_line(100,100,300,200,15);
 k=open_printer(hnd2);
 j=select_graphics_object(hnd2);
 fill_ellipse(200,150,75,50,2);
 k=close_printer(hnd2);
```

open_printer is a ClearWin+ function that generates a standard "Open Printer" dialog box from which the user selects a graphics printer or plotter. Subsequent calls to Salford graphics drawing routines are written to this device and the output is generated when the ClearWin+ routine close_printer or the Salford graphics routine new_page is called.

Notes:

- open_printer etc. can be used independently of winio..
- The standard call-back function gprinter_open can be used to provide a simpler

interface (see page 125).

- In the above example, j and k can be used to test for the success of the associated function calls.
- The option FULL_MOUSE_INPUT can be used with %gr allowing the mouse to be fully used within a graphics region. A call-back function will be called every time the user moves the mouse or presses a mouse button.

Graphics boxes can be made to change dimensions as the enclosing window is re-sized by the user. By default the graphics area will have a fixed size. When resizing is permitted, the programmer has 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 graphics box 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 methods:

1) If the user_resize option is given to the %gr format and a call-back function is supplied (i.e. %^gr[user_resize]) then when the graphics box 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_info using the parameter GRAPHICS_RESIZING. This will return 1 if a re-size is occurring. The GRAPHICS_WIDTH and GRAPHICS_DEPTH parameters can be used to return the new size of the region. 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:

```
int draw();
main()
{
  winio("%ww%pv%^gr[user_resize]",80,25,draw);
}

int draw()
{
  int x,y;
  if(clearwin_info("GRAPHICS_RESIZING")==1)
  {
    x=clearwin_info("GRAPHICS_WIDTH");
    y=clearwin_info("GRAPHICS_DEPTH");
    draw_line(0,0,x,y,7);
  }
  return 1;
}
```

2) An alternative method of re-sizing is to supply the option metafile_resize to the %gr format. 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 amount of graphics being sent to the box must not be unreasonably large since each graphics call will be recorded for re-play. For example, a fractal drawing program which continued to draw more detail for as long as it ran could not use this technique, since it would ultimately run out of memory.

b) The other consideration is that some images may not respond well to such automatic re-scaling. This is because floating point co-ordinates will have been truncated to integers at the time the original graphics were drawn and will now be scaled and 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.

%gr graphics regions can have a 'selection mode'. This can be set using the BOX_SELECTION or LINE_SELECTION options on the %gr routine. The mode can also be changed dynamically using the set_graphics_selection function. This operates on the current graphics area and the argument can take the following values: 0 - default selection, 1 - Box selection, 2 - line selection. When enabled, dragging the mouse with the left button depressed opens an XOR drawn box or line. A program can obtain the co-ordinates using get_graphics_selected_area which takes arguments X1,X2,Y1,Y2.

RGB_COLOUR. Windows has the potential to access true colour depths of up to 32 bits per pixel. Graphics regions have been enhanced to cope with this. The format modifier RGB_COLOUR for the format code %gr will ensure that when anything is written to the region the correct true colour is written. Alternatively the related routine use_rgb_colours() can be used.

USER_SURFACE. This is only available under Win32 (not under Win32S or Win3.1). When this option is used, pointer to char* variable should be passed immediately after the two size parameters.

This variable will receive the address of a buffer containing the graphics contents stored at 3 bytes per pixel. By manipulating this array you can change the graphics surface directly. The 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) by using this direct access method if USER_SURFACE is coded, but at present the surface is only available for direct program manipulation. 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 which ClearWin+ normally used 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 set_pixel. 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:

```
char buffer[width][3];
```

The first index covers the colours as follows:

0 Blue

1 Green

2 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 not ClearWin+!

Note that it is possible (and often very useful) to mix calls to Salford graphics routines and direct manipulation of the pixel buffer. If you do this under Windows NT you must call the SDK function **GGDIFlush()** 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.

Caption format

%ca<title>

Defines the title of the dialog box as part of the format. The title is supplied as a standard character string.

For example:

```
winio("%ca[Error]");
```

Whenever the window is refreshed the caption will be re-drawn. This means that if the title is supplied in the argument list (signified by the "@" character) then the window title can be changed by altering the text that is pointed at.

Control variable format

%cv

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

Window handle

%hw

%HW Example

The %hw format can be used to set a variable to the window handle of the window being created. This will be the same as the handle returned by a call to `clearwin_info("LATEST_FORMATTED_WINDOW");`. though no confusion will arise in cases where several windows are created at the same time. %hw takes one integer variable as an argument. It may then be necessary to cast the resultant integer to a `HWND` handle for further operations.

Leave window open format

%lw

Normally `winio` does not return until the window that it creates is closed. Occasionally it is useful to create a window which remains open while the program takes other actions. The %lw format causes `winio` to return as soon as it has created the window (the return value is not important in this case). The format takes one `int*` argument that will receive the eventual result. This integer, known as the control variable, will be 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 wise to use a variable that is static

The %lw format can also be used to create a child window to be inserted within another window created by `winio` using the %ch format. The grave accent is used to produce this effect. In this case no window is produced until the control variable is passed to the %ch format of a subsequent call to `winio`. See the %ch format for an example of this procedure.

A child window can be used only once. If it is never used it will be destroyed when the program terminates.

A window that has been left open by means of %lw may be closed by setting its control variable to a non-negative value (the signal that normally indicates that a window has closed) and passing the address of the control variable to `window_update`.

For example:

```
int control;
  winio("Processing - please wait%lw",&control);
  do_something_complex();
  control=0;
  window_update(&control);
```

Disable screen saver - %ns

%ns

The %ns format 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 (minutes) calculation, there is a danger that a screen saver will cut in when your program yields, even temporarily. This is because windows 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.

Set window position format

%sp

This format 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:

```
winio("%si?%spWhy is this stuck in the corner",0,0);
```

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.

Create screen saver

%sv

%SV Example

Using the %sv format it is possible to create a screen saver. A window with %sv will close as soon as it receives mouse or keyboard input. The executable should be renamed to a <filename>.SCR file, be placed in your windows directory and then be selected as the current screen saver. Apart from its obvious use, to create a pretty screen saver program which will run whenever the system is idle, it could be designed to perform a lengthy calculation. To ensure the calculation can be 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:

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdlib.h>
#include <dbos\graphics.h>
#define LAST 5
#define OBJX 50
#define OBJY 30
int h_gr=1,xres,yres,l=0,c=0;
int cx[LAST],cy[LAST],cdx[LAST],cdy[LAST],cclr[LAST];
int screen_update()
{
 select_graphics_object(h_gr);
 for(l=0;l<LAST;l++)
 {
   ellipse(cx[l],cy[l],OBJX,OBJY,0);
   cx[l]+=cdx[l];
   if ( cx[l]<0 || cx[l]>(xres-OBJX) ) cdx[l]=-cdx[l];
   cy[l]+=cdy[l];
   if ( cy[l]<0 || cy[l]>(yres-OBJY) ) cdy[l]=-cdy[l];
   ellipse(cx[l],cy[l],OBJX,OBJY,cclr[l]);
   cclr[l]=(++cclr[l])%255;
 }
 return 2;
}
int main()
{
 xres=clearwin_info("SCREEN_WIDTH");
 yres=clearwin_info("SCREEN_DEPTH");
 for(l=0;l< LAST;l++)
 {
  cx[l]=rand() % xres;
  cy[l]=rand() % yres;
  cclr[l]=rand() % 255;
  if ((rand() % 2 )==1) cdx[l]=-2+-rand()%8; else
     cdx[l]=2+rand()%8;
  if ((rand() % 2 )==1) cdy[l]=-2+-rand()%8; else
     cdy[l]=2+rand()%8;
 }
```

```
 winio("%sv%bg[black]%ww[no_caption,no_maxminbox,no_sysmenu,
     no_border,no_frame,topmost]%`gr[black]&",xres,yres,h_gr);
 winio("%dl",1.0,screen_update);
 return 0;
}
```

Set window size format

%sz

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

```
int w,h;
winio("%sz",&w,&h);
```

w and *h* should initially be set to zero 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 order to recreate the window size that was last used. A maximised window can be configured in the same way because specific values are used to represent this state.

Wallpaper format

%wp<bitmap>

This format takes the name of a bitmap resource where <bitmap> is a standard character string and "wallpapers" the window with the bitmap before the various controls are drawn.

For example:

```
#pragma windows 500000, 500000
#include <windows.h>
int main()
{
 winio("%ca[Wall Paper]%wp[smile]\n\n %bt[Button 1]\n\n"
           "%bt[Button2]\n\n");
 return 0;
}

#pragma resource
smile BITMAP "smile.bmp"
```

This is most effective in windows which only contain 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, such as the standard Windows wallpaper bitmaps. A grave accent (`) can be added to ensure that the window will be expanded to hold at least one copy of the bitmap.

Window control format

%ww[<option list>]

This format causes the resultant window to look like a normal application window, rather than a dialog window. The window can be re-sized, maximised, etc. The format can be followed by an option list enclosed in square brackets (use an empty list if the next character is an open square bracket "["). The following options are available:

`casts_shadow`	This option produces a window which casts a shadow. It is most effective when used in conjunction with no_frame and no_caption.
`no_caption`	Omit the caption from the window.
`no_maxminbox`	Omit the maximise and minimise boxes from the window.
`no_maxbox`	Omit the maximise box.
`no_minbox`	Omit the minimise box.
`no_sysmenu`	Omit the box at the top left which can be used to produce the system menu.
`no_border`	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.
`no_frame`	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.
`no_edge`	This option is like no_frame except that it leaves the default border i.e. an empty space around the edge of the window.
`topmost`	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 loosing sight of the window of the main application.
`maximise (or maximize)`	Displays the window as a maximised window.
`minimise (or minimize)`	Displays the window as a icon.

Background colour format

%bg[<colour>]

The %bg format code can be used with or without the grave accent modifier. Without the modifier, %bg is used to change the background colour for the main window. With the modifier, %`bg is used to change the background colour for the next (and only the next) control (used for example with %rd, %rf, %rs, %ed, %lb). %`bg should not be used for buttons since, in this case, it is the button colour rather than the background colour that is significant. For buttons use %bc.

By default a main window uses the background colour that the user can select from the Control Panel icon of the Program Manager. This colour can be determined by an Windows API function call in the form `GetSysColor(COLOR_WINDOW)`. The %bg format specifies that the background colour is different from the default.

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 can take one integer argument which is typically created using the Windows API **RGB** macro.

For example:

```
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:

```
int color,red,green,blue;
color=GetSysColor(COLOR_WINDOW);
red  =GetRValue(color);
green=GetGValue(color);
blue =GetBValue(color);
color=RGB(red,green,blue+50);
winio("%bg",color);
```

will deepen the blue component of the background colour (the new value is assumed to be less that 256).

Attach window format

%aw

This format attaches the window as an MDI child of a parent which must contain an MDI frame (%fr). The %aw format does this and takes a pointer to an integer control variable as an argument. This should be the control variable of the corresponding parent window (which must already exist). If the parent window does not use %lw, and therefore does not have a control variable, use %cv in the parent window to create one. 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.

```
#pragma windows 500000,500000
 #include <windows.h>
 #define NAMESIZE 128
 int ctrl;
 int open_func();
 main()
 {
   char fname[NAMESIZE];
   winio("%ww[no_border]%mn[&File[&Open,E&xit]]%fr%lw",
     "FILE_OPENR",fname,NAMESIZE,"+",open_func,"EDIT_FILE",fname,
     "EXIT",400,300,&ctrl);
 }

 int open_func()
 {
   winio("%aw%20.8eb[no_border]",&ctrl,NULL,0);
   return 1;
 }
```

Child window format

%ch

This format inserts a child window created by %`lw in a previous call to `winio`. The format takes a single `int*` argument pointing to the control variable of the 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 (like other window controls). Here is a simple example:

```
static int control_var;
winio("%ww[no_frame]%`lw",&control_var);
winio("This is a parent window\n\n%ch",&control_var);
```

Note that the variable *control_var* in the above example was declared static. Although not essential, it is recommended that all control variables be `static` or `public` (i.e. not `automatic` variables) because when a function containing automatic variables returns, the memory associated with those variables is re-used.

Note that %fr and %aw can be used to produce a Multiple Document Interface (MDI) application.

Colour palette

%cl

When this format is used a colour selection window appears. This allows any of the standard windows colours to be selected or it allows the user to redefine an entry so that they can get the exact shade of colour they actually require. It must be noted that this option is dependant of the video display type and mode being used. A 16 colour only mode will attempt to produce further colours by placing two other colours in an alternating pattern (dither) that confuses the eye into seeing a third colour.

This format takes one argument that holds the value of the selected colour and must be of type `long`. This is in the same format as the result of the `RGB(r,g,b)` function.

```
#pragma windows 500000,500000
#include <windows.h>

int  main()
 {
  long v;
  winio("%ca[colour palette] %cl",&v);
  winio("%ca[colour mix result] The colour was %wd.“,v);
  return 0;
 }
```

ClearWin window format

%<n.m>cw[<option list>]

Using this format you can embed a *ClearWin* window (as described in chapter 6) in a formatted window. The window is created *n* characters wide and *m* deep but this is changeable with the pivot control format modifier. This format takes one argument which is a pointer to a `FILE` structure (as defined in `stdio.h`). The `FILE` structure is set up for use in the standard library functions `fprintf` etc. Alternatively, if a `NULL` pointer is passed as the argument, the *ClearWin* window is created as the default stream. By default the window has no caption and is not scrollable. The options *vscroll* and *hscroll* may be used to provide for scrolling. 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.

A grave accent (`) can be used to obtain the window handle which is returned via an integer argument. This can then be used in call to routines e.g. `set_max_lines`.

Although the *ClearWin* window can be used for both input and output, it is normally wise to perform input using other formats and use the *ClearWin* window for scrolling results. These 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.

Create MDI frame format

%fr

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

A grave accent (`) can be added to this format to make it return a handle to the currently selected child window. An extra integer argument must be supplied to hold the returned handle. . The integer will be zero if there is no child window. The child widow's handle can be obtained by using the %hw format. A comparison check can then be made on the handle updated by %fr with those previously defined with %hw.

Create a Property sheet

%<n>ps

%PS Example

Property sheets represent a way of presenting two or more 'sheets' of data in a format which resembles a card index. Each sheet is set up as a separate window using `winio` and the %sh format. This produces a child window which is hidden from view until connected to a property sheet control using %ps. The property sheet can also have a call-back function which is called each time the visible sheet is changed. The `SHEET_NO` parameter is provided (see `clearwin_info`) so that it is possible to determine which sheet is now topmost. When the sheet is first displayed the `SHEET_NO` is set to 1 and the call-back is also generated.

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

```
#pragma windows 500000,500000
#include <windows.h>
#include <dbos/graphics.h>
int g1=1,g2=2,wh1,wh2,col1=1,col2=5,loop,ctrl;

int redraw1() // callback
{
  select_graphics_object(g1);
  for(loop=0;loop<200;loop++)
     draw_line(0,loop,200,loop,col1=(++col1)%255);
  return 2;
}

int redraw2() // callback
{
  select_graphics_object(g2);
  for(loop=0;loop<200;loop++)
     draw_line(0,loop,200,loop,(loop+col2)%255);
  col2=(col2+=4)%255;
  return 2;
}

int main() // main code
{
     winio("%sc%ca[window1]%`gr[black]\t\n\t %7^bt[Redraw]%sh",
     redraw1,200,200,g1,redraw1,&wh1);
     winio("%sc%ca[window2]%`gr[black]\t\n\t %7^bt[Redraw]%sh",
     redraw2,200,200,g2,redraw2,&wh2);
     winio("%ca[Property Sheet Example]%2ps\t%8^bt[EXIT]%lw",
     &wh1,&wh2,"EXIT",&ctrl);
 return 0;
}
```

Embed in property sheet

%sh

This format code is very much like the %lw code. Its should be placed at the end of a window definition and an argument should be provided so that a handle to the window can be returned. See %ps for an example as %ps and %sh can only be used in combination.

User window

%uw

This format enables existing Windows code to be interfaced with ClearWin+ in many instances. . Given a registered Windows class (which will have its own Windows procedure), a window may be embedded in a *ClearWin* window using the %uw (user window) format. . The format takes the following arguments:

1) A standard string containing the name of the class.
2) `int*` width in pixels.
3) `int*` height in pixels.
4) Window style. This will be OR-ed with WS_CHILD. . In general no border or caption should be specified, as these will normally be applied to the main window (e.g. using %ca).
5) Extended window style bits (typically zero).
6) An `int*` to receive the HWND handle for the resultant child window.

Closure control format

%cc

%CC Example

Sometimes, especially when an edit box is displayed, it is desirable to control the closure of a window. The %cc format takes a pointer to a call-back function as its argument. This function is called before the window closes (for whatever reason). If it returns a non-zero value the window is kept open.

Delayed auto recall

%dl

This format allows a call-back routine to be called at regular intervals. This takes a floating point argument which is the number of seconds and fraction of a second to wait between call-backs.

Note: the timing messages are only received while the program is idle awaiting input, or when the `yield_program_control()` function is called.

The following code displays its message for two seconds before closing:

```
winio("%dl%si!Testing",2.0,"EXIT");
```

File selection format

%fs<path name>

%FS Example

The file selection format, together with the related %ft format, changes the default file selection used by a subsequent `FILE_OPENR` or `FILE_OPENW` call-back function. It takes a standard character string as argument, which can either be the path name of a directory to be used rather than the current working directory (e.g. `C:\TEST`), or it can be a complete file selection specification (e.g. `C:\TEST\*.EXE`). In the latter case the file filter information is replaced (as displayed at the bottom of the file selection box). For more control over this box see the %ft format.

```
#pragma windows 500000,500000
#include <windows.h>

int disp_name();
char filenm[100]="";

main()
{
    winio("%fs@@%mn[&File[&Open,E&xit]]",
       "c:\\dbos.dir\\*.wri","FILE_OPENR",
       filenm,100,disp_name,"EXIT");
}

int disp_name()
{
  winio("%ws %`bt[ok]",filenm);
  return 1;
}
```

File filter format

%ft<filter name><filter>

%FT Example

The file filter format is used to change the default filter for a subsequent use of the `FILE_OPENR` and `FILE_OPENW` call-back functions. Two standard character strings should be supplied. The first is the name of the filter (e.g. 'Executable files'). The second is the specification (e.g. '`*.EXE`').

For example:

```
winio("%ft@@","Executable files","*.EXE");
```

If the grave accent modifier is used, the filter information replaces the existing information, otherwise it is added (to the front) of any existing information. The default filter for existing information selects all files.

Start-up call-back

%sc

This format 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/screen.

```
int first_time()
{
 winio("%ww[no_border]\n\t This is a test program\t\n&”);
 winio("to demonstrate the percent sc format\n\n\t%bt[ok]\n");
 return 1;
}
int main()
{
 winio("%ca[hi]%sc\nThis is the main window\n",first_time);
 return 0;
}
```

Edit box format

%<n[.m]>eb[<option-list>]

%EB Example

This format edits text and is by far the most complex format available. However, many applications will not require any of the embellishments detailed below.

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 (`CHR(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 for 60 characters on one line (which is the same as %60.1eb). %60.20eb denotes an edit box with the same width and 20 lines. Note that it is simpler to use %rs for a one line edit box because %rs does not require the character string to be null terminated.

The text format is that of a normal DOS text file (except that a `NULL` terminator must be added). In order to read/write data to a file in this format the file should be opened in binary mode.

For example:

```
char buffer[1000];
FILE* f=fopen("myfile","rb");
int fsize=fread(buffer,1,999,f);
buffer[fsize]='\0';
fclose(f);
winio("%60.20eb",buffer,1000);
f=fopen("myfile","wb");
fwrite(buffer,1,strlen(buffer),f);
```

The edit box can also handle text lines separated by carriage return characters only, providing this is used consistently.

<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. The following options are currently defined:

`hscrollbar`	Supply and control a horizontal scroll bar.
`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.
`hook_focus`	Causes the call-back to be called every time the box gains or looses the focus. Note that the over use this option can cause annoying effects for the user and more importantly an infinite loop can 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_border`	Omit the blank border that by default is placed between the text and the surrounding box.
`read_only`	Prevents the user from changing the text. The cursor can still be used to scroll through the text.

The edit box maintains a set of variables in an `EDIT_INFO`(defined in *`clearwin.h`*, see below) which control 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 a pointer to an `EDIT_INFO` structure. This structure has the following format:

```
struct EDIT_INFO  {
 int  h_position;   //Cursor horizontal character position
 int  v_position;   //Cursor vertical character position
 int  last_line;    //Total no of lines in the buffer
 char *buffer;      //pointer to buffer
 int  buffer_size;  //Size of buffer contents
        //(excluding nul terminator)
 int  max_buffer_size;   //Size of memory block
 char *current_position; //Buffer position corresponding to
                  //h_position/v_position
 char *selection;   //Points to selected text if any
 int  n_selected;   //No of selected characters
 int  vk_key;       //Set to VK... if this handles a key press
 int  vk_shift;     //Shift state corresponding to key
 int  active;       //Set when call-back invoked, reset afterwards
 int  modified;     //Set to 1 each time the buffer is modified
 int  closing;      //Set when buffer is about to be closed
 char reserved[40]; //For future enhancements
};
```

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.

```
  char buffer[1000];
  EDIT_INFO info;
  char fname[]="myfile";
  FILE* f=fopen(fname,"rb");
  int fsize=fread(buffer,1,999,f);
  buffer[fsize]='\0';
  fclose(f);
  winio("%60.20`eb",buffer,1000,&info);
  if(info.modified)
  if(winio("%cnSave changes?\n\n%6bt[Yes]  %6bt[No]")==1)
  {
    f=fopen(fname,"wb");
    fwrite(buffer,1,strlen(buffer),f);
    fclose(f);
  }
```

Note that, when the user supplies the edit buffer, the following fragment of code would give the position of any selected text in the window relative to the beginning of the file.

```
char buffer[1000];
EDIT_INFO info;
int position;
winio("%60.20`eb",buffer,1000,&info);
position=info.selection-buffer;
```

Note also that, if the user supplies the edit buffer, a call-back procedure can be used to monitor the size of the text in the buffer and replace the buffer (e.g. using the C function `realloc`) if it is getting full. If the buffer actually overflows, the system will generate a fatal error.

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 function (but see the next section for a simpler method of specifying these operations). The last function informs the edit control that the contents of the buffer have been replaced. In each case the function returns 1 if successful, and zero otherwise.

An edit format may use the caret (^) edit modifier to indicate that a pointer to 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.

Because an edit box may contain valuable information which the user will not wish to lose by mistake, it is a good idea to use the %cc format to restrict the closure of the main window.

The associated call-back function can (for example) interrogate the `modified` flag in the `EDIT_INFO` structure and use another call to `winio` to ask the user to confirm closure.

Parameter box

%<n.m>pb

%PB Example

A parameter block is a control which 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 can take 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 formats thus:

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

One use for the user parameter is to activate another parameter block to achieve a hierarchical effect.

The various parameter types are illustrated by the following example:

```
#include <windows.h>
int kh=0,kv=0;

char* textures[]={"Sticky","Messy","Dirty","Greasy","Slimy","Very
slippery",NULL};
int action()
{
       winio("No action today\n\n%cn%bt[Thank you]");
       return 2;
}
main()
{
    char name[20]=" Black n Sticky 123";
    int k1=1,k2=2,k4=4,k5=5;
    int texture_type=4;
    double v=4.5,p=200,a=25.2;
    winio("%ww[casts_shaddow]Double click on an item to change"
            "it\n\n&");
    winio("%30.8pb[sorted]&");
    winio("%dp[test1]&",&k1);
    winio("%dp[test2]&",&k2);
    winio("%ep[Oil texture]&",textures,&texture_type);
    winio("%dp[Temperature (Deg C)]&",&k4);
    winio("%dp[Carbon monoxide (%)]&",&k5);
    winio("%fp[Curent (Amps)]&",&a);
    winio("%10.3fp[Voltage]&",&v);
    winio("%fp[Oil pressure (PSI)]&",&p);
    winio("%?tp[Oil name][What is the official name of this"
                "oil?]&",name,20);
    winio("%up[Special action]&",action);
```

```
    winio("");
}
```

Text array

%<n.m>tx

%TX Example

%TX Example 2

Text arrays provide a convenient way to display a grid of characters with 'attributes' to achieve an effect analogous to those achieved under DOS by writing to the screen buffer. A text array can respond to keyboard and mouse input via its call-back function and `clearwin_info@` (see below), updating the array as necessary. The text array format %n.mtx[options] takes two character strings and two integer modifiers n, and m.

For example:

```
char text[800],attr[800];
winio("%60.8TX",text,attr,80,10);
```

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 variable if it is preceded by a pivot in a variable sized window. The attr array contains attribute numbers stored as. Zero is the default attribute, others are defined by subsequent %ty[**bgcolour**] %tc[**fgcolour**] format pairs. The first %ty%tc pair define attribute type CHAR(1) , etc. The **bgcolour** is the background colour of the text, and **fgcolour** is the foreground. They may be specified as an RGB value in the argument list if the colour in brackets is omitted. Changes in font, underlining, etc. are not permitted.

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

`TEXT_ARRAY_CHAR`	value of key pressed
`TEXT_ARRAY_DEPTH`	holds new value on resizing
`TEXT_ARRAY_MOUSE_FLAGS`	mouse button press information
`TEXT_ARRAY_RESIZING`	text array window dimension change
`TEXT_ARRAY_WIDTH`	holds new value on resizing
`TEXT_ARRAY_X`	location x when the mouse is pressed in tx
`TEXT_ARRAY_Y`	location y when the mouse is pressed in tx

The `TEXT_ARRAY_RESIZING` parameter is set to 1 only in a call-back responding to a re-sizing event. The next two parameters only have meaning in this context. They supply the new size of the control in average character widths.

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, which contain information about which mouse button (if any) is depressed, are defined as in %dw. The option `FULL_MOUSE_INPUT` can be specified to force call backs whenever the mouse moves over the control.

By default the call-back function is called when a keyboard character is sent while the control is in 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 windows.h). 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 can 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.

Unless an explicit font has been selected prior to %tx, the `SYSTEM_FIXED_FONT` is used, since a text array is supposed to be textually aligned both vertically and horizontally. You can select an explicit font for the array, but you must select a fixed pitch font.

Note carefully that the text array is not re-dimensioned if the control is re-sized, re-sizing simply changes the region which is made visible.

The Visual Basic Custom Control interface

%vb

ClearWin+ format codes feature many useful controls for programmers and engineers. It is unlikely that we will be able to satisfy every ones needs with these codes. To help alleviate this problem an interface to Visual Basic Custom Controls has been added. This opens up a whole new world to ClearWin+ programmers.

Most (but not all) of the custom controls that we have tested seem to be well behaved. If you find a control which does not function as expected please contact us with information about your program and the control in question. Please bear in mind that the vast number of controls available mean that we may not be able to make the control work correctly with ClearWin+.

About VBXs

A VBX is an encapsulated object which provides some form of functionality for a windows application. This may simply be a modified edit control or may be as sophisticated as a modem control module. You can think of a control as a C++ class or Fortran 90 module. The control has properties which control the way in which the control acts and/or appears. These properties may also be interrogated to accretion the current state of the control. A control also generates events which indicate to your program that some action or system event has occurred. VBXs are available from commercial companies and also from the public domain.

Loading a VBX

Before a VBX can be used it must be loaded by ClearWin+. The act of loading a VBX registers it with both ClearWin+ and the operating system. A VBX is loaded by issuing a call the function `load_vbx` as follows:

```
int result;
result = load_vbx("gauge.vbx");
```

The return code of the function indicates if the VBX could be loaded successfully. It is possible to give the full path name of the VBX to be loaded. If the file name only is given, the VBX is assumed to be in the windows system directory. The above line of code will attempt to load the gauge control VBX which is supplied with Visual Basic version 3.0 from the windows system directory.

A return code of 0 indicates that the function has failed to find the requested file. It is important to check the return code from this function is case the user is running the program from a system which does not have the required custom controls present.

The loading of the custom control will register the class name of the control with the application. This is not necessarily the same as the file name for the control. The above example loads the file GAUGE.VBX but the class name for the control is BIGAUGE. It is important that you know the class name for the controls that you are intending to use in your application.

Adding a VBX to a ClearWin+ window

A VBX is added to a window by using the %vb format code. This will search the list of registered VBXs for the requested control. If the control is found it will be added to the window at the current position. If the control cannot be found an error will be generated and the application will terminate. The following line of code will add the gauge control to the currently open formatted window.

```
int result, vbx_handle;
result = winio("%50.1vb[BIGAUGE]&", &vbx_handle);
```

Remember, the control is added by using the class name of the control and not the file name. So the above line adds the gauge control to the window and dimensions the control to be 50 characters long and 1 line deep. It may be necessary to refer to the control later to set properties or to inquire about events that have been generated. Each control is assigned a handle to allow your application and ClearWin+ to identify the control you are referring to uniquely. This is returned in the variable *vbx_handle*.

Setting properties

Properties may be set when the control is added to the ClearWin+ window or dynamically at run-time. You will usually want to use a combination of the two methods.

The first method, setting properties when the control is added to the window, is achieved using the '[' and ']' characters. So to set the background colour to colour 6 we can write the following:

```
int result, vbx_handle;
result = winio("%50.1vb[BIGAUGE][BackColour=6]&", &vbx_handle);
```

Multiple properties may be specified with each being separated by a comma.

```
int result, vbx_handle;
result = winio("%50.1vb[BIGAUGE][BackColor=6,BevelStyle=1]&", &vbx_handle);
```

It is also possible to set the properties of a control using the contents of a variable rather than a fixed constant string format.

```
int result, vbx_handle, colour;
colour = 6;
result = winio("%50.1vb[BIGAUGE][BackColor]&", &vbx_handle,
                colour);
```

The omission of the "**=**" sign informs ClearWin+ to look for a variable name in the list of arguments to the winio function.

The final method of setting a property for a Visual Basic control is dynamically under program control. This may be to respond to some event. The gauge control illustrates this. It would not be much use to have the gauge control permanently fixed at the same value. You will want the value to change to represent some action nearing completion etc. It is not desirable to have to close and redisplay the window to achieve this.

The properties of a VBX may be of three types, integer, real and string. The following function allow you to set properties of each type:

```
set_vbx_integer_property(handle, property, value)
set_vbx_real_property(handle, property, value)
set_vbx_string_property(handle, property, value)
```

The function names should indicated the action of the function. The handle parameter is the handle returned to you when you add the VBX to the window (*vbx_handle* in the above examples). *property* is a string containing the name of the property whose value is to be modified. *value* is the actual value to be assigned to the property. So the above example for setting the background colour to 6 could have been coded thus:

```
int result, vbx_handle;
result = winio("%50.1vb[BIGAUGE]&", &vbx_handle);
set_vbx_integer_property(vbx_handle, "BackColor", (int) 6); int
```

Events

Events are generated in response to some system or user action such as a mouse button press, a drag and drop action etc. A call back is assigned to a VBX control in the same manner as all other ClearWin+ formats. One additional requirement is the events that the call back is to respond to. The following code responds to the single and double click events on the gauge control.

```
int result, vbx_handle;
result = winio("%^50.1vb[BIGAUGE][^=Click,^=DblClick]&",
                  &vbx_handle,GaugeCallBack);
```

The "%^50.1vb" part of the format string specifies a call back function to be associated with this particular control. Note that only one call back function is allowed per format code. The end part of the string "[^=Click,^=DblClick]" specify the events that will generate a call back to the specified function. The setting of properties may be mixed with the specification of the events which will cause the call back function to be entered.

As it is only possible to have one call back per VBX it is necessary to determine which event has generated the call back function to be entered. You can ascertain this by interrogating the string VBX_EVENT from within the call back function.

Events may have data associated with them. It is not worth catching a key press event if you cannot determine which key has actually been pressed. Each event may have one or more parameters which contain useful information. They are the parameters that would be passed to a Visual Basic subroutine when the event is generated.

You can find the type of the argument by calling the `clearwin_info@` routine passing the argument `VBX_PARAMETER_TYPE`. The return types are summarised in the following table:

Return Value	Parameter Type
0	No more parameters
1	Integer number
2	Floating point number
3	String

It is not recommended that this approach is adopted. You should consult your manuals for the VBX you are using to determine the parameter types and the meaning of the parameter. Three functions have been provided to get the value of a parameter.

They are:

`clearwin_integer("VBX_INTEGER_PARAMETER")`	Returns an integer parameter
`clearwin_float("VBX_FLOATING_PARAMETER")`	Returns an floating point parameter
`clearwin_string("VBX_STRING_PARAMETER")`	Returns a string parameter

The values of these parameters may be set by calling the associated `set_clearwin` functions.

When using these functions to get/set the values of the parameters you must take care that

1) You do not modify the value of a parameter before you have obtained its value.

2) You do not attempt to obtain a parameter which does not exist

Call back functions

Whilst a VBX event is being processed, no other VBX events are allowed to occur. The events may be generated by the user say clicking on a control but they will be ignored. This is particularly important when you make calls to winio or to yield_program_control. If your application performs calculations of some other activity that does not involve interacting with windows then any events will be queued and any call back functions will be activated when you return from the function.

Example

The following function illustrates the use of the Visual Basic gauge control to display a horizontal bar which acts as a 1 minute timer. The stop watch is activated by double clicking on the horizontal bar. It is stopped in the same way.

```
//  Stop.cpp
//
//  This program illustrate the use of the gauge VBX to simulate a
//  simple second counter/stopwatch.
//  This application uses the gauge control as supplied with
//  Visual Basic
//  version 3.0.  This control is not shipped with ClearWin+.
//
//  (C)opyright Salford Software Ltd 1995.

#pragma windows 500000, 500000
#include <windows.h>
#include <stdlib.h>
#include <time.h>

//  Preprocessor macros.
#define TITLE "Gauge Control Example"
#define INTEGER_PROPERTY(x, y, z) set_vbx_integer_property(x,y,z)

//  Now for the global variables for ClearWin+.
int counting;
//  Pause for 1 second.  Yield control to the system for a while.
void WaitOneSecond()
{
    time_t  start, now;
    start = time(NULL);
    do
        {
         yield_program_control(1);  // Don't hog the  system.
         now = time(NULL);
        }
        while (difftime(now, start) < 1);
}
 //  Reprt a fatal program error and abort the program run.

void FatalError(char *message)
{
    #pragma silent 287
    int     result;
    result = winio("%ca[" TITLE "]&");
    result = winio("Error : %ws&", message);
    result = winio("%dn%8bt[&Ok]");
    exit(-1);
}
```

```
//  Use a double mouse click to turn the stop watch on/off.
//  This is a toggle function.

int DoubleClick()
{
    if (counting) counting = 0; else counting = 1;
    return(2);  // use 2 as 1 refreshes the whole window
}

//  Now for the main program loop.
#pragma silent 288

void main(void)
{
    #pragma silent 287
    int     result, vbx_handle, value, control;
    result = load_vbx("gauge.vbx");
    if (!result)
        FatalError("Cannot load the Gauge control");
// Now build up the output window.The window has a title, a single
// gauge in the window with a button to exit the program.
// All controls are centred.
// Note the use of the standard call back function "STOP" to
// terminate the program.
    result = winio("%ca[" TITLE "]&");
    result = winio("Double clink on the bar to start/stop the
                counter\n\n&");
    result = winio("%cn%^50.1vb[BIGAUGE][^=DblClick]\f\n&",
                &vbx_handle, DoubleClick);
    result = winio("%cn%^8bt[&Exit]%lw", "STOP", &control);

    //  Set up some properties for the VBX.
    INTEGER_PROPERTY(vbx_handle, "Min", 0);
    INTEGER_PROPERTY(vbx_handle, "Max", 59);

    //  Now for the main stopwatch functionality
    counting = 0;
    value = 0;
    while (1)
        {
        WaitOneSecond();
        if (counting)
            {
            value++;
            value = (value % 60);
            INTEGER_PROPERTY(vbx_handle, "Value", value);
            }
```

```
            }
        }
```

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. The purpose of each modifier will first be described in general terms. Then a number of illustrative examples are given in order to present the details in a palatable form.

[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.

[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 can be 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.

Output a single character

%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.

Output integer values

%wd

%wd is used for the output of integer values.

format	output for i=123	comment
%wd	`123`	minimum width
%6wd	`123`	minimum of 6 characters, right justified
%6.4wd	`0123`	show 4 digits, padding with zeros
%-6wd	`123`	left justify
%06wd	`000123`	pad with zeros on the left
%+6wd	`+123`	force a + sign when not negative

Output floating point exponent

%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	`1.260000e-02`	minimum of 14 characters, right justified
%10.2we	`1.26e-02`	10 characters, 2 decimal places
%6.0we	`1e-02`	6 characters, no decimal point
%-9.1we	`1.3e-02`	9 characters, 1 decimal place, left justified
%010.3we	`01.260e-02`	right justified, padded with zeros
%+10.2we	`+1.26e-02`	force a + sign when not negative
%#8.0we	`1.e-02`	force a decimal point for zero precision

Output floating point values in decimal form

%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	1.260000	minimum of 10 characters, right justified
%6.2wf	1.26	6 characters, 2 decimal places
%2.0wf	1	2 characters, no decimal point
%-6.1wf	1.3	6 characters, 1 decimal place, left justified
%07.3wf	001.260	right justified, padded with zeros
%+6.2wf	+1.26	force a + sign when not negative
%#4.0wf	1.	force a decimal point for zero precision

Output floating point value

%wg

%wg is used for floating point values and has essentially the same effect as %wf unless the supplied value can not 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`	`0.000126`	exponent is -4
%.3wg	`126.7`	`126.7`	exponent is less than the precision
%9.2wg	`1.267e-5`	`1.27e-05`	exponent is less than -4
%7.2wg	`100.0`	`1e+02`	exponent is equal to the precision

Output character strings

%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.

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.

Output hexadecimal values

%wx

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

Save settings to .INI file

%ss[name]

Saves the supplied variables to an .INI file that is located automatically in the windows directory, not the current working directory. When the program is rerun the variables stored in the ini file will automatically be restored. The file name string (without the extension .INI) must be supplied in the square brackets followed by a (/) character and the field name that will appear in the .INI file.

An control argument must also be supplied of type `int`, this will be a save control flag designed to stop the auto save of settings if set to zero. This is useful when an abort/abandon path is taken throughout the program rather than a clean exit, thus preventing erroneous, incomplete data to be stored. The %ss format should be included in your options dialog.

For example:

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdlib.h>

unsigned  inumber=12345,n1=5,h=85;
unsigned  save_control=1;

main()
{
  winio("front %ss[winio9/blue] inum %rd NNum %rd&",
         &save_control,&inumber,&n1);
  winio("%ss[winio9/green] inum %rd",&save_control,&h);
}
```

Call-back functions

Call-back functions consist of functions with no arguments which return an integer result. A returned zero value causes the parent window to close. A returned value of 1 leaves the window open and updates the whole display area to reflect any changes in the data. Under normal circumstances it is not the most desirable return code as frequent returns will cause unpleasant screen flickers. A returned value of 2 leaves the window open but does not update it and is therefore the most suited to repetitive display updates. A return value of 2 however, does not manage any updates itself so you must update and parts of the window with the function window_update.

A number of frequently used call-back functions can be specified as character strings.

(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 will verify that 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 fault.)

Some of these built-in functions consume additional arguments from the `winio` argument string. The following functions are available:

ABOUT

This takes one `char*` 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 inbuilt into windows.

For example:

```
winio(" %^bt[PRESS]" , "+" , "BEEP[MBOK]",
cb_openfile );
```

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 the 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 an edit box and the clipboard. Typically this is used as the call-back function for an edit-copy/cut/paste menu item. The menu or button is automatically greyed when the operation is not possible (any grey-control integer is ignored). These call-back functions consume no extra arguments.

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 takes one `char*` argument containing a file pattern (e.g. *.TXT) and a second character for the resultant file name (of size 129). A file-open window is displayed and the resultant file is read into a buffer and displayed in an edit box, replacing any text already there. This should only be used with a window containing one and only one 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.

EDIT_FILE

Takes one `char*` argument that is the file name of the file to be opened. Unlike `EDIT_FILE_OPEN`, the standard open dialog is not presented.

EDIT_FILE_SAVE, EDIT_FILE_SAVE_AS

Operates on an edit box like `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 a file-save dialog box. Each call-back takes two `char*` arguments one for 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.

EXIT

This closes the window. If a window closure call-back (%cc) has been specified, this is used.

FATAL

This call-back function 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. Only use this if the program is likely to be corrupt, so that performing a normal cleanup might hang the system.

FILE_OPENR[<*caption for open dialogue*>**]**

This takes a `char*` argument for the returned file name followed by an integer holding the number of characters in the string. A call-back function should follow, which is called if the user selects a file (as opposed to pressing cancel, etc.).

For example:

```
winio("%^bt[Open file]", "FILE_OPENR[Open]", str_file,
str_len, cb_process);
```

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 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. For an example of the use of the `FILE_OPENR` and "+" call-back functions see the description of the %aw format.

FILE_OPENW[<*caption for save dialogue*>**]**

This is analogous to `FILE_OPENR` but allows the user to type in a new name, since it is assumed that the file is required for writing.

GPRINTER_OPEN

This call-back function is used to produce output on a graphics printer or plotter. It operates in the same way as `printer_open` but differs from that function in two respects. 1) It takes only one argument, namely a call-back function supplied by the programmer. 2) This call-back function should call Salford graphics drawing routines. Note that graphics printing and plotting can also be carried out using open_printer.

`HELP_CONTENTS`

This takes one `char*` 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 necessary.

HELP_ON_HELP

Provides the standard help-on-help information. This also takes a `char* pointer` containing the name of a help file.

PRINTER_OPEN

This displays a dialog box enabling the user to select a printer. It takes an integer identification argument (described below) and another call-back function. If the printer is successfully selected and opened the supplied call-back function is invoked and its return value is returned to the window to determine if it remains open. If no printer is selected, this function returns 1 to keep the window open. The identification integer is used in calls to `pr_printf` and `pr_close` (these are undocumented ClearWin+ functions from *clearwin.h*). Unless you wish to open more than one printer at a time, this value may be set to zero. Because an open printer locks down a Windows device context, it is very important to call `pr_close` to close a printer which has been opened in this way.

SET

This call-back takes two integer arguments the first is an `int*` and the second an `int`. Each time the

call-back is called the first argument is set to the value of the second and the set call-back returns 1 so that any necessary updates take place. This call-back can be used to simplify many aspects of window control.

SOUND

If you require more than a system beep (see BEEP above) and you have installed a sound card in windows then you will be able to include small samples, stored in standard wave format (*.wav). Any wave file must be included in the resource section of your program and be defined as follows:

```
MYSOUND  SOUND  dognoise.wav
```

The sample can then be called as follows:

```
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 displayed on the entire screen. Programs which use this should provide an accelerator key or other means to exit this mode (i.e. by definition there will be no menus visible).

TEXT

This is a quick way of displaying text in a window. The text should follow the call-back text argument. This should not be invoked from a window that does not contain hypertext.

TEXT_HISTORY

When this is called a text box is displayed that shows the hypertext history and allows the user to select a previous selection.

TOGGLE

This call-back can be used to toggle an integer variable between 0 and 1. The variable follows the call-back argument.

+

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 result of the "+" call-back function is the result of the second call. For an example of the use of the "+" and FILE_OPENR call-back functions see the description of the %aw format.

The following example uses a number of the above call-back functions. It implements a simple editor.

```
char file[129];
char new_file[129];
winio("%mn[&File[&Open,&Save,Save &As,E&xit],
      &Edit[&Copy,Cu&t,&Paste],
      &Help[&Contents,&Help on help]]
      %60.20eb",
      "EDIT_FILE_OPEN",   "*.*",file,
      "EDIT_FILE_SAVE",   "*.*",new_file,
      "EDIT_FILE_SAVE_AS","*.*",new_file,
      "EXIT",
      "COPY","CUT","PASTE",
      "HELP_CONTENTS","c:\\edithelp.hlp",
      "HELP_ON_HELP", "c:\\edithelp.hlp",
      NULL,0);
```

Updating windows

Many of the format codes of the `winio` function take pointers to data which 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:

```
char myform[50]="TEST 1";
winio("%ca@&",myform);
```

The address of the variable `myform` has been passed to `winio`, and if the contents of the array `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

```
void window_update(void *)
```

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 `winio`. For example, in the above example, it would be of no use passing the address of *myform*+1, even though this points to part of the caption array.

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, 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 should close.

Note that in general `window_update` will only update those objects which 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, no more than once per second. Very frequent updates can produce unpleasant strobing effects with the screen refresh frequency.

Related routines

The following routines are used in conjunction with the `winio` function.

activate_bitmap_palette

Purpose

Ensures that palette is set correctly for the current window.

Syntax

```
activate_bitmap_palette( int 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
get_colours
set_colours
use_rgb_colours

add_graphics_icon

Purpose

Places an icon on top of a graphics window.

Syntax

```
int add_graphics_icon(char *name, int *x, int *y, int width, int depth)
```

Description

A graphics region can have an bitmap or icon resource ‘attached’ to it. The icon can be freely moved around the graphics window with the mouse. It is under the control of the call-back function attached to the %gr window with the caret modifier (^).

The `NAME` is that used when the resource is included in you program. The x and y values, initially hold the location the image is draw to. They also hold the new position it the image is moved. The `clearwin_info` parameter ‘`DRAGGED_ICON`’ is set to the value of the handle returned by the function to indicate that a dragging action is in progress. The call-back function can modify the x and y values if required. The purpose of this may be to ‘lock’ the image onto the horizontal or vertical plane.

When the icon is dropped `clearwin_info` parameter ‘`DROPPED_ICON`’ is set to the handle. 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 only using the top-left 16x16 area and filling the remaining unused area with the ‘transparent’ colour. If you do not specify the correct size of the icon in a 16x16 (or less) icon then the unused area will be detected by the mouse.

See also

remove_graphics_icon

add_hypertext

Purpose

Add some text to the hypertext system.

Syntax

```
add_hypertext( char *buffer, int size,
char *htextname )
```

Description

`add_hypertext` adds, to the hypertext resource base, a hypertext document containing one or more hypertext sections. 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.

See also

add_hypertext_resource

add_hypertext_resource

Purpose

Opens a hypertext resource for a program that is included in a resource file. See %ht format and the browse example code included with the ClearWin+ release.

Syntax

```
add_hypertext_resource( char *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 at the end of the program:

See also

add_hypertext

Example

```
#pragma resource
text  HYPERTEXT  book.htm
```

add_menu_item

Purpose

Allows for dynamic menu definitions.

Syntax

```
add_menu_item( int handle, char *name, int &grey, int &check, int (*)
()callback_function)
```

Description

Menus items can be added dynamically to menus. This is of greatest use when you wish to show a file history or a list of open windows.

In the %mn definitions an '*' must be added where a text description would previously have been placed and a integer argument must be provided to hold the HANDLE.

For example:

```
winio("%mn[&Window[*]] &",handle)
```

The HANDLE can then be passed to this routine.

The NAME is the new string to put into the menu i.e. the new menu item. The GREY variable will control the greying out of the item and the CHECK variable will add or remove a tick character. Final a call-back function must be provided so that ClearWin+ can act upon a menu selection.

See also

remove_menu_item

%mn

attach_bitmap_palette

Purpose

To attach a palette to a DC.

Syntax

```
attach_bitmap_palette( int hdc, int p )
```

Description

To attach a colour palette to a device context. It is not normally needed as `set_colours` provides sufficient flexibility. On success a 1 will be returned. A -1 will indicate that the screen type can not provide the palette requested due to too few palette entries available. A 0 indicates that the display is in full colour mode and therefore does not use a palette.

See also

get_colours
set_colours
use_rgb_colours

bold_font

Purpose

To make a graphics region text bold.

Syntax

```
bold_font( int active )
```

Description

When using the function `draw_text` with any font selected other than a Hershey font, in a graphics region, a call to this routine with a value of 1 will make any further text output bold. A further call with a value of 0 will deactivate this function.

See also

rotate_font
scale_font
select_font
italic_logfont
underline_font
select_logfont

change_pen

Purpose

To change the pen in a device context acquired through `get_bitmap_dc`.

Syntax

```
int change_pen( int hdc, int pen_style, int width,
int 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, the system keeps track of which pens have been created, re-uses them where possible and finally deletes them automatically. The style, width, and colour arguments are as supplied to the Windows API function CreatePen.

clear_bitmap

Purpose

To clear a bitmap device context acquired by `get_bitmap_dc`.

Syntax

```
clear_bitmap( int hdc )
```

Description

The bitmap associated with the given device context is cleared to the current Windows background

See also

create_bitmap
export_bmp
export_pcx
get_bitmap_dc
import_bmp
release_bitmap_dc

clearwin_float

Purpose

To interrogate the state of the ClearWin+ environment.

Syntax

```
double clearwin_float( char *param )
```

Description

This function returns the value of the specified floating point parameter. Parameters are specified as case insensitive character strings.

See also

clearwin_string
clearwin_version
set_clearwin_string
set_clearwin_info

clearwin_info

Purpose

To interrogate the state of the ClearWin+ environment.

Syntax

```
int clearwin_info( char *param )
```

Description

This function returns the value of the specified parameter. Parameters are specified as case insensitive character strings, and the following are currently 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.

CALL_BACK_WINDOW

This option provides the call-back routine with the handle of the window generating the call-back.

DRAGGED_ICON

Is set to the value of the handle for the dragged icon, indicating that a icon is currently being dragged.

DROPPED_ICON

Is set to the value of the handle of the handle for the recently dropped icon.

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)

GAINING_FOCUS

Returns 1 when an (%eb) edit box gains the focus.

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 graphics region. It will return zero if there is no current graphics area. Using this handle you can write additional information to a graphics area using SDK functions. The HDC points to a bitmap so it is also possible to extract information form this area. Do not use this HDC 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 graphics area will be re-sized (replacing its HDC) or closed. It is therefore important to obtain the HDC again in such circumstances. In general, obtain the HDC, perform the actions you require, and discard the HDC.

GRAPHICS_RESIZING

This returns 1 if the window has been resized making it necessary to call 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

The flags associated with a mouse event in a graphics (%gr) or owner-draw (%od) control.

The flags should be considered as a set of bits thus:

1 Left button down
2 Right button down
4 Shift key held down during mouse action
8 Control key held down during mouse action
16 Middle button down

The following mask should be used so that any other bits are ignored:

```
MouseData=clearwin_info(GRAPHICS_MOUSE_FLAGS) & 31;
```

GRAPHICS_MOUSE_X

The X co-ordinate of a mouse event in a graphics (%gr) or owner-draw (%od) control.

GRAPHICS_MOUSE_Y

The Y co-ordinate of a mouse event in a graphics (%gr) or owner-draw (%od) control.

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.

LOOSING_FOCUS

Returns 1 when an edit box (%eb) looses 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.

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_COPIES

This only has meaning after the user has been interrogated by a printer-open box. It returns the value entered into the "number of copies" field (default 1). The `winio` function does nothing with this field other than return its value in this call.

PRINTER_FIRST_PAGE

This only has meaning after the user has been interrogated by a printer-open box. It returns the start of the page range selected by the user (default 1). The `winio` function does nothing with this field other than return its value in this call.

PRINTER_LAST_PAGE

This only has meaning after the user has been interrogated by a printer-open box. It returns the end of the page range selected by the user (default = largest integer). The `winio` function does nothing with this field other than return its value in this call.

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. 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

Holds the character code when a key is pressed.

TEXT_ARRAY_DEPTH

Holds the new value on resizing

TEXT_ARRAY_MOUSE_FLAGS

Mouse button pressed information

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

The following mask should be used so that any other bits are ignored:

```
MouseData=clearwin_info(GRAPHICS_MOUSE_FLAGS) & 31;
```

TEXT_ARRAY_RESIZING

Text array window dimension change flag

TEXT_ARRAY_WIDTH

Holds the new value on resizing

TEXT_ARRAY_X

Location x when the mouse is pressed in %tx (text) box

TEXT_ARRAY_Y

Location y when the mouse is pressed in %tx (text) box

TREEVIEW_ITEM_SELECTED

Flags 1 when there has been an item selected from the treeview control.

See also

clearwin_string
clearwin_version
set_clearwin_string
set_clearwin_float

clearwin_string

Purpose

Obtain string information from ClearWin+.

Syntax

```
char *clearwin_string( char *str )
```

Description

ClearWin+ maintains a set of system strings that contain information from the system. The following have been defined:

CURRENT_TEXT_ITEM

It is set during a call-back generated by a hypertext link. The string is the HREF parameter of a hypertext anchor which does not correspond to an another hypertext document. By examining this string and taking action depending on its contents you can create hypertext objects which respond by executing portions of your program.

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.

DROPPED_FILE

Is a string containing the full path and filename of the object dropped into the window defined by %dr.

VBX_EVENT

If the VBX call-back function is called, it is because one of your specified events has fired. You can determine which by examining the ClearWin+ string 'VBX_EVENT' from within the call-back function. Events can have associated parameters of integer, floating point, or string type. The number and type of such parameters is fixed for a given type of event.

See also

clearwin_info
clearwin_version
set_clearwin_info
set_clearwin_float

clearwin_version

Purpose

Gets the current ClearWin+ version information.

Syntax

```
int clearwin_version()
```

Description

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.

See also

clearwin_info
clearwin_string
set_clearwin_info
set_clearwin_string

Example

```
#pragma windows 500000, 500000
#include <windows.h>
main()
{
  int ver;
  winio("%ca[Version information]&");
  version=clearwin_version();
  winio("\nMajor %wd minor %wd\n",ver>>8,ver&255);
}
```

clipboard_to_screen_block

Purpose

To allow any bitmap information in the clipboard to be copied to a screen block.

Syntax

```
int clipboard_to_screen_block( int &dib )
```

Description

This function copies the DIB out of the clipboard which can be displayed with a call to `dib_paint`. A return value of 0 indicates a failure.

See also

copy_to_clipboard
edit_clipboard_cut
copy_from_clipboard
edit_clipboard_copy
edit_clipboard_paste
graphics_to_clipboard

close_cd_tray

Purpose

To close an open CD-drive tray.

Syntax

```
close_cd_tray()
```

Description

Simply by calling this routine any open CD drive with a mechanical drawer will close.

Example

```
winio(“ %^bt[close cd]”,cbclosecd);
...
int cbclosecd()
{    close_cd_tray(); return 2; }
```

See also

open_cd_tray
play_audio_cd
set_cd_position
stop_audio_cd

close_metafile

Purpose

Closes the previously opened metafile.

Syntax

```
close_metafile( int handle )
```

Description

If `handle` is zero then the current graphics block is being used otherwise a graphics block handle should be supplied.

See also

open_metafile
print_graphics_page
do_copies

close_printer

Purpose

To output to a graphics printer or plotter.

Syntax

```
int close_printer( int handle )
```

Description

See `open_printer`. This function returns 1 for success or zero for failure.

See also

do_copies
open_printer
select_printer
open_metafile
close_metafile

copy_from_clipboard

Purpose

Copy data out of the clipboard.

Syntax

```
int copy_from_clipboard( char *buffer, int num,
int type )
```

Description

Use this routine to copy anything from the clipboard. The following data types can exist in the clipboard:

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_DSPTEXT	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.

A 1 is returned on success or a 0 on failure.

See also

copy_to_clipboard
edit_clipboard_cut
clipboard_to_screen_block
edit_clipboard_copy
edit_clipboard_paste
sizeof_clipboard_text

Example

```
char *buffer;
int size_text = sizeof_clipboard_text();
if (size_text == -1) return 0;
if (!(buffer=(char *)malloc(size_text))) return 0;
copy_from_clip( buffer, size_text, CF_TEXT );
```

copy_graphics_region

Purpose

Copies screen blocks in graphics regions defined by %gr.

Syntax

```
int copy_graphics_region(int dest_gr, int dx, int dy,
  int dwidth, int dheight, int src_gr,int sx,int sy,
  int swidth, int sheight, int copy_mode )
```

Description

You must have at least one graphics region open (see %`gr). The grave accent (`) must be used so that you can obtain a handle to the graphics region. The DEST_GR is the handle of the destination graphics region and the SRC_GR is the source graphics region however, the handles specified can be the same. If you set either or both to zero then the current graphics region is assumed for source, destination or both.

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 however, there are any differences then the image will be 'stretched' accordingly.

The COPY_MODE defines the method of copying. Windows defines 255 different copies (ROPS). The most useful are described below:

Hex Value	Name	Logical Description
CC0020	SRCCOPY	src (most common - a direct copy)
8800C6	SRCAND	src and dest
660046	SRCINVERT	src xor dest
440328	SRCERASE	src and not dest
EE0086	SRCPAINT	src or dest
330008	NOTSRCCOPY	not src
1100A6	NOTSRCERASE	not (src or dest)
BB0226	MERGEPAINT	not src or dest
550009	DSTINVERT	not dest
000042	BLACK	0
FF0062	WHITE	1

See also

scroll_graphics
select_graphics_object

copy_to_clipboard

Purpose

Copies data into the clipboard for use by other applications.

Syntax

```
int copy_to_clipboard( char *buffer, int num,int 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.

A 1 is returned on success or a 0 on failure.

See also

copy_from_clipboard
clipboard_to_screen_block
graphics_to_clipboard
sizeof_clipboard_text

create_bitmap

Purpose

Creates a bitmap from bitmap data.

Syntax

```
int create_bitmap( int ptr )
```

Description

Creates a bitmap from a pointer to the data in a .BMP file. When the program is terminated the bitmap will be deleted. It returns a handle to the new bitmap.

See also

attach_bitmap_palette
activate_bitmap_palette
clear_bitmap
export_bmp
export_pcx
get_bitmap_dc
import_bmp
release_bitmap_dc
make_bitmap

create_cursor

Purpose

Creates a cursor from the data in a .CUR file.

Syntax

```
int create_cursor( int ptr )
```

Description

Creates a cursor from the data in a .CUR file. When the program is terminated the cursor will be deleted. On success a handle to the new cursor is returned.

define_file_extension

Purpose

Allows application registry under Windows 95 (Win32 only).

Syntax

```
VOID define_file_extension( char *extname, char *path, char *description, int
icon_index, int New_option )
```

Description

Under Windows 95 it is possible to register an application with the system so that if a data file is opened in the explorer your program 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 DESCR. 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 the ICON_INDEX the second icon resource will be used. The NEW option should be set to a non zero value to activate the file type addition to Windows 95.

For example:

```
define_file_extension(".ico", argv[0], 'icon file
editor", 0, 1);
```

In C you can obtain the file passed by Windows 95 to your program by using argc and argv in you main program section.

For example:

```
int main (int argc,  char *argv[] )
{
      if (argc == 2)
      {   // a file has been passed
      Fpointer=fopen( argv[1], "rb" );
      ...
      }
}
```

dib_paint

Purpose

To display a DIB.

Syntax

```
int dib_paint( int horiz, int vert, int hdib, int function,
int mode)
```

Description

Displays the DIB with handle HDIB on the current device with horiz & 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.

A value of 1 is returned if successful otherwise it is a 0.

See also

attach_bitmap_palette
activate_bitmap_palette
clear_bitmap
export_bmp
export_pcx
get_bitmap_dc
import_bmp
release_bitmap_dc
make_bitmap

display_popup_menu

Purpose

To activate a popup menu.

Syntax

```
display_popup_menu()
```

Description

This subroutine will activate a pre-defined popup menu. This is of most use when the right mouse button is pressed over a graphics region that has FULL_MOUSE_INPUT activated.

do_copies

Purpose

Produces multiple copies.

Syntax

```
do_copies( int handle, int num )
```

Description

Sends *num* copies the printer. A metafile must have been created with a call to `open_metafile` and also closed by `close_metafile` before the `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

export_bmp

Purpose

Exports a DIB to a .BMP file.

Syntax

```
export_bmp(int dib_handle, char *filename, int error )
```

Description

Writes a DIB to a bitmap file specified by *filename*. Any errors are returned by error which may be any of the following:

0	SUCCESS
1	BAD_OPEN
3	BAD_WRITE

See also

export_pcx
import_pcx
import_bmp

export_pcx

Purpose

Exports a DIB to a .PCX file.

Syntax

```
int export_pcx( int dib_handle, char *filename,
int error )
```

Description

Writes a DIB to a bitmap file specified by *filename*. Any errors are returned by error which can be any of the following:

0	SUCCESS
1	BAD_OPEN
3	BAD_WRITE

See also

export_bmp
import_bmp
import_pcx

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

```
int get_bitmap_dc(int h_bits, int v_bits )
```

Description

The bitmap is created of size h_bits x 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.

See also

attach_bitmap_palette
activate_bitmap_palette
clear_bitmap
create_bitmap
release_bitmap_dc

get_colours

Purpose

To access the palette.

Syntax

```
get_colours( int first, int numregs, char *rgbarray )
```

Description

This subroutine loads the `rgbarray` with the values contained in the windows palette registers. `first` indicates the first palette entry to be obtained in the range 0..255. `numregs` is the number of palette entries to be obtained thereafter. The `rgbarray` is an array of bytes. Three bytes are used for one palette entry as they relate to the red, green and blue values.

`rgbarray` now contains a copy of four palette entries starting at entry 20 and ending at entry 24. Each entry is three bytes long.

See also

set_colours

Example

```
get_colours( 20, 4, rgbstore )
```

get_current_dc

Purpose

Yields the current graphical device context (screen, printer, ...).

Syntax

```
get_current_dc( int &dc )
```

get_graphical_resolution

Purpose

Determining the width and height of the current graphics region.

Syntax

```
get_graphical_resolution( int &width, int &height )
```

Description

width and *height* are set to the current graphics region values which is defined with the format code %gr.

get_font_name

Purpose

Returns the name of loaded fonts.

Syntax

```
get_font_name( char *name, int num )
```

Description

The name of installed fonts 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 a empty string will be returned.

Example

```
char name[20];
     int k=1;
     do {
           get_font_name(name,k);
           window_update(name);
           k++;
        }
     while( name!=NULL );
```

get_graphics_selected_area

Purpose

Gets the current graphics block information.

Syntax

```
get_graphics_selected_area(int &x1, int &y1, int &x2,
int &y2 )
```

Description

The four parameters are set to the values of the current graphics selection block which is defined by the ‘rubber-band’ line activated with a call to `set_graphics_selection`

Note: that this function should not be used when selection mode is 0.

See also

set_graphics_selection

%gr

get_im_info

Purpose

To obtain information in graphics files.

Syntax

```
get_im_info( char *filename, int width, int height,
int  nb_colours, int nb_images,
char *format, int 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. On failure *error* will contain:

10	BAD_FILE
15	UNKNOWN_FORMAT

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

```
get_mouse_info( int &x, int &y, int &flags )
```

Description

This function 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 owner-draw format. flags contains the following flags (the parameters will be found in *windows.h*):

`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

See also

clearwin_info

get_nearest_screen_colour

Purpose

To find the closest colour.

Syntax

```
COLORREF get_nearest_screen_colour( COLORREF colour )
```

Description

`get_nearest_screen_colour` selects the closest palette colour that is the best match to the supplied colour.

See also

get_rgb_value
set_colours
use_rgb_colours

get_rgb_value

Purpose

Sets the *value* to the colour value of a pixel.

Syntax

```
get_rgb_value( int hor, int ver, int &value )
```

Description

The horizontal and vertical co-ordinates locate the pixel in the current graphics region. The value is set to the pixels current value and is a compatible format to that returned by `RGB`.

See also

get_nearest_screen_colour
set_colours
use_rgb_colours

get_vbx_integer_property

Purpose

To obtain a VBX integer property (see %vb).

Syntax

```
int get_vbx_integer_proprty( HCTL hctl, char *name )
```

Description

This routine will allow you to obtain an integer for a VBX routine identified by the *hctl*.

See also

get_vbx_floating_property
load_vbx

get_vbx_floating_property

Purpose

To obtain a VBX floating point property (see %vb).

Syntax

```
double get_vbx_floating_property(HCTL hctl, char *name)
```

Description

This routine will allow you to obtain a floating point for a VBX routine identified by the *hctl*.

See also

get_vbx_integer_property
load_vbx

graphics_to_clipboard

Purpose

To allow the interchange of graphics between graphics regions and standard windows programs.

Syntax

```
int graphics_to_clipboard( int x1, int y2, int x2,
int y2 )
```

Description

Places a region of a graphics window, defined by %gr, into the clipboard so that other windows programs can use the graphic image. On success this routine will return a 1 and a 0 will be returned on failure.

x1, y1 is the top left corner of the region.

x2, y2 is the lower right corner of the region.

See also

copy_from_clipboard
copy_to_clipboard
clipboard_to_screen_block
edit_clipboard_cut
edit_clipboard_paste
edit_clipboard_copy

import_bmp

Purpose

Reads in a BMP file.

Syntax

```
int import_bmp( char *filename, int &error )
```

Description

Reads in a BMP file and returns a DIB handle to the image. The handle can then be used with the with a suitable format code e.g. %`bm. On error the error parameter will contain one of the following:

2	BAD_READ
10	BAD_FILE
5	NOT_BMP

See also

clear_bitmap
create_bitmap
export_bmp
export_pcx

import_pcx

Purpose

Reads in a .PCX file.

Syntax

```
int import_pcx( char *filename, int &error )
```

Description

Reads in a PCX file and returns a DIB handle to the image that is compatible with windows bitmaps allowing it to be used with the %`bm format code. On error the error parameter will contain one of the following:

2	BAD_READ
10	BAD_FILE
6	NOT_PCX

See also

clear_bitmap
create_bitmap
export_bmp
export_pcx

italic_font

Purpose

To set italic font style in a graphics region.

Syntax

```
italic_font( int ital )
```

Description

This routine sets or resets the italic property of the current graphics region (defined by %gr). Any subsequent call to the `draw_text` routine will be affected. This has no effect upon the HERSHEY font.

See also

rotate_font
scale_font
select_font
select_logfont
underline_font
bold_font

load_vbx

Purpose

Loads a VBX library.

Syntax

```
load_vbx( char *filename)
```

Description

To use a VBX control, install the VBX file in the Windows system directory and call the function `load_vbx`with the VBX file name as argument. For example, to access the popular GRAPH.VBX control:

```
load_vbx(“graph.vbx”);
```

This loads the VBX and makes it available to subsequent %vb formats. VB may be preceded by %pv if desired. K will be non-zero if the system finds and loads the VBX file. It is important to check this value, because your user may well move the .VBX file by mistake.

See also

get_vbx_integer_property
get_vbx_floating_property

make_bitmap

Purpose

Embedding bitmaps in program code.

Syntax

```
int make_bitmap( char *BMP_DATA, int xres, int 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 Windows handle to a bitmap will be returned. This provides a speedy method of designing simple graphics that would otherwise require image editing software.

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	
l	Light grey	
	B	Blue
	R	Red
	G	Green
	Y	Yellow
	M	Magenta
C	Cyan	
	W	White

See also

make_icon

make_icon

Purpose

Embedding icons in program code.

Syntax

```
int make_icon( char *icon_data )
```

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. This is useful when designing icons that you do not want to have a square appearance. A windows handle to the icon is returned which may be used with any other icon manipulation routine. The icon is automatically discarded at program termination.

Example

```
#pragma windows  500000,500000
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
```

```
char icondata[]=
{ 'R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R'
,'R','R','R','R','R','R','B','R','R','R','R','R','R',
//
// Note: 30 similar lines have been omitted from this guide.
//
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
```

```
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
'R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','
R','R','R','R','R','R','B','R','R','R','R','R','R'
};

int main()
{
  int hicon=make_icon( icondata );
  winio("%ca[Make icon]\n%`ic\n",hicon);
  return 0;
}
```

See also

make_bitmap

open_cd_tray

Purpose

To open a CD-drive tray.

Syntax

```
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_metafile

Purpose

To record graphics sequences so that they can be replayed to the printer.

Syntax

```
int open_metafile( int h )
```

Description

Opens a metafile for the current printer graphics area. A metafile is used for recording graphics calls so that they can be replayed to the printer device. The example shows clearly how this can be achieved.

See also

close_metafile
open_printer
close_printer

open_printer

Purpose

To begin output to a graphics printer or plotter.

Syntax

```
int open_printer( int 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` or the Salford graphics routine `new_page` is called.

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. 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

Purpose

To begin output to a graphics printer or plotter.

Syntax

```
int open_printer1( int handle )
```

Description

`open_printer` does not generate a standard "Open Printer" dialog box which means that printing can resume to the current printer after a close printer. This avoids the need to repeatedly select the same printer. Subsequent calls to Salford graphics routines are be written to this device. The printer or plotter is activated when `close_printer` or the Salford graphics routine `new_page` is called.

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. 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

perform_graphics_update

Purpose

To refresh a graphics regions display.

Syntax

```
perform_graphics_update()
```

Description

The current graphics region, 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 only displayed.

See also

select_graphics_region

play_audio_cd

Purpose

Plays the audio CD from the current position.

Syntax

```
int play_audio_cd( int 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.

See also

close_cd_tray
open_cd_tray
set_cd_position
stop_audio_cd

Example

```
set_cd_position( 1, 0);
play_audio_cd( -1 );
```

play_sound

Purpose

Sends a sample to the audio output device.

Syntax

```
int play_sound( short int &left, short int &right,
int samples )
```

Description

The *left* and *right* parameters are arrays that contain the 16 bit sample data. The number of samples is contained in samples. 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. A 1 is returned on success and a 0 on failure.

See also

write_wave_file
record_sound
sound_playing
sound_recording
select_sampling_rate
play_sound_resource

play_sound_resource

Purpose

Sends a resource SOUND to the current audio device.

Syntax

```
int play_sound_resource( char *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 name of the resource to this routine.

For example:

Include in the resource section of you program:

```
MYWAVE    SOUND     "dogbark.wav"
```

In the code section:

```
res=play_sound_resource("MYWAVE");
```

On success a 1 is returned otherwise a -1 is returned.

See also

write_wave_file
play_sound
record_sound
sound_playing
sound_recording
select_sampling_rate

record_sound

Purpose

Records sound from the sound input device.

Syntax

```
int record_sound( short int &left, short int &right,
int samples)
```

Description

The *left* and *right* parameters are arrays of short int, 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*. 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. A 1 is returned on success and a 0 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

```
release_bitmap_dc( int 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
%dw

remove_graphics_icon

Purpose

To remove an icon from a %gr graphics region.

Syntax

```
remove_graphics_icon( int handle )
```

Description

To remove an icon from a %gr graphics region call this routine and supply the handle to it that was returned from the call to `add_graphics_icon`.

See also

add_graphics_icon

remove_menu_item

Purpose

Removes a dynamically attached menu item.

Syntax

```
remove_menu_item( int *handle)
```

Description

Use this routine to remove a menu item that has previously been attached with a call to `add_menu_item`.

See also

add_menu_item

rotate_font

Purpose

To rotate the font being used in a graphics region.

Syntax

```
rotate_font( int rot )
```

Description

This routine will rotate the selected font about the bottom-left most corner of the text. It is only for use in a graphics region defined by %gr. The rotation is anti-clock wise and *rot* is specified in degrees.

See also

italic_font
scale_font
select_font
select_logfont
underline_font
bold_font

scale_font

Purpose

Scales the font being used in a graphics region.

Syntax

```
scale_font( int size )
```

Description

Rescales the font being used in the current graphics region.

See also

rotate_font
select_font
select_logfont
underline_font
italic_font
bold_font

scroll_graphics

Purpose

Allows you to scroll a region of a graphics window defined by %gr.

Syntax

```
scroll_graphics( int dx, int dy, int left, int top,
int right, int bottom, int switch, int colour)
```

Description

This routine allows you to scroll a section of a graphics region. 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 then 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 graphics region or a sub region within the whole.

The 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. 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

see_treeview_selection

Purpose

Ensure that the current item is visible.

Syntax

```
see_treeview_selection( int point)
```

Description

This routine ensures that the treeview (defined by %tv) 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 have ensured that any necessary node are marked as expanded ('E') to make the selected item visible.

select_font

Purpose

Selects a new font type for a graphics window %gr.

Syntax

```
select_font(char *fontname )
```

Description

This will become the font for the current device. The default values are loaded for the size.

See also

rotate_font
scale_font
italic_font
select_logfont
underline_font
bold_font

select_graphics_object

Purpose

To select a graphics area for use with %gr and `open_printer`. Where more than one graphics region is being used at once.

Syntax

```
select_graphics_object( int handle )
```

Description

For an full explanation of how to use this routine see the description of %gr and `open_printer`. The handle is setup when the format code %`gr is processed.

See also

scroll_graphics

select_printer

Purpose

To configure and select the active printer.

Syntax

```
int select_printer( char *dev, char *port )
```

Description

A call to `select_printer` will provide a standard Windows dialogue box from which the user is able to select and configure any of the attached printer devices.

On return the *dev* string will contain the name of the printer device that has been selected and the *port* string will contain the name of the port it is connected to.

A character array of around 20, in each case, should be sufficient to hold the returned data. On success `select_printer` will return with a 1 and on a failure a 0.

See also

open_printer

close_printer

Example

port	LPT1: , COM3: , FAX: ...

set_cd_position

Purpose

Sets the position of the CD read head to *track* + *milliseconds* (offset).

Syntax

```
int set_cd_position( int track, int milliseconds)
```

Description

The CD read head will be moved to the track plus the number of specified milliseconds into that track. On successes a 1 will be returned however, on failure a 0 will be returned.

See also

close_cd_tray
open_cd_tray
play_audio_cd
stop_audio_cd

set_clearwin_float

Purpose

Permits the creation and modification of a `clearwin_float`

Syntax

```
set_clearwin_string(char *string, double value)
```

Description

This function has two uses. It can either be used to modify existing `clearwin_floats` that have been provided to enhance the functionality of this product or it can be used to create new `clearwin_floats` that can be used to pass information around call-back functions.

See also

set_clearwin_string
set_clearwin_float
clearwin_info
set_clearwin_info
clearwin_version

set_clearwin_info

Purpose

Alters the value of a `clearwin_info`string.

Syntax

```
set_clearwin_info( char *string, int value )
```

Description

This function will create a new named parameter in necessary. User defined parameters can be very useful to communicate information between modules in a program (especially between DLL's under Win32).

See also

clearwin_string
set_clearwin_float
clearwin_info
set_clearwin_info
clearwin_version

set_clearwin_string

Purpose

Permits the creation and modification of `clearwin_strings`.

Syntax

```
set_clearwin_string(char *string, char *value)
```

Description

This function has two uses. It can either be used to modify existing `clearwin_strings` that have been provided to enhance the functionality of this product or it can be used to create new `clearwin_strings` that can be used to pass information around call-back functions

See also

set_clearwin_string
set_clearwin_float
clearwin_info
set_clearwin_info
clearwin_version

set_clearwin_style

Purpose

Modifies the styling controls for the window.

Syntax

```
int set_clearwin_style( int newstyle )
```

Description

The default setting for style is (WS_OVERLAPPEDWINDOW + WS_HSCROLL + WS_VSCROLL). This can be changed to any valid combination of windows styles contained in the `windows.h` file so that any subsequent widows 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_DLGFRAME
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

The previous settings are returned and should be stored so that they can be reset if the effect you wish to generate is only temporary.

set_colours

Purpose

Sets a section or all a palette.

Syntax

```
set_colours( int first, int num, char *rgbarray )
```

Description

For those developing modern applications it is possible to use the modifier `rgb_colours` with %gr. For example `%gr[rgb_colours,...]`. This will provide access to the true colour range of the display surface which can have a maximum colour resolution of 24 bits per pixel.

To `first` variable indicates which is the first entry that will be modified. The valid range is 0..255. `num` indicates how many following entries are to be modified.

For example:

If FIRST = 100 and NUM = 3 then entries 100, 101, 102 will be modified.

The new palette information is stored in the `rgbarray` each palette requires three `BYTES` of information that correspond to the red, green and blue components of the new colour. For each colour component only six out of the eight bits in the byte have any effect. It has been arranged so that the MSB (most significant) 6 bits are copied into the palette. The 2 LSBs are discarded.

For the above example the `rgbarray` will need to be populated with the following 9 BYTES:

```
RED, GREEN, BLUE, ( palette 100 )
RED, GREEN, BLUE, ( palette 101 )
RED, GREEN, BLUE   ( palette 102 )
```

See also

get_rgb_value
get_nearest_screen_colour
use_rgb_colours

set_graphics_selection

Purpose

Gives user feed back when selecting areas or drawing lines.

Syntax

```
set_graphics_selection( int mode )
```

Description

The *mode* variable can be any of the following:

0	No Selections
1	Rectangle Selection box
2	Line drawing

This routine is used when mouse input is being used within a %gr graphics region. It remains local to the current graphics region selected with `select_graphics_object`.

Mode 1 make 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 graphics region also until the mouse button is released. By processing the mouse pressed and released information it is possible to determine the co-ordinates of the selected region.

Mode 2 is similar to mode 1 but a direct line joins the first point to the second whilst the left mouse button is held down.

Mode 0 deactivates the box and line hi-lighting mechanism. All the lines are draw onto 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 required. If it is placed in the mouse call-back the top-left corner will be reset each time the mouse moves to co-ordinate (0,0). It is possible to select which type of graphics selection is required when the graphics region is opened.

See also

get_graphics_selected_area

set_line_style

Purpose

Sets the style for the line.

Syntax

```
set_line_style( int value )
```

Description

This routine changes the line style within the current graphics region.

See also

set_line_width

set_line_width

Purpose

Sets the line width.

Syntax

```
set_line_width( int value )
```

Description

Changes the pixel thickness of a line when drawing in a graphics region. The default value is 1.

See also

set_line_style

set_old_resize_mechanism

Purpose

To revert to a previous window resizing strategy.

Syntax

```
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, or applied a pivot to a control which could not use it. We have changed the specification slightly because this could result in many bizarre effects, and did not produce anything useful.

To obtain the old specification you should call `set_old_resize_mechanism`. The new scheme is as follows:

In the absence of a pivot a window will not re-size. A pivot may only be placed on a control which can actually re-size in response. It is an error to attempt to pivot any other type of control. The following controls will accept the pivot:

`%eb` Edit boxes

`%ht` Hypertext

`%cw` Embedded *ClearWin* windows

`%fr` Frame for MDI child windows

`%lb` and `%ls`
List box controls (but not drop down boxes with grave accent)

`%ms` Multiple selection boxes

`%gr` Graphics box, must have metafile_resize or user_resize property.

`%tx` Text array.

`%dw[user_resize]` Owner draw box format - provides owner draw boxes

`%ht` Hyper-text boxes

set_sound_sample_rate

Purpose

Sets the sampling speed.

Syntax

```
set_sound_sample_rate( int 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) and 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).

You should also be aware that this will require a large of amount storage.

See also

write_wave_file
play_sound
sound_playing
sound_recording
play_sound_resource
sound_sample_rate

set_vbx_floating_property

Purpose

Modifies a vbx floating point (real) value.

Syntax

```
set_vbx_floating_property(HCTL  handle,char *name,double value )
```

Description

This function allows you to send floating point values in a previously loaded VBX. The `HANDLE` is the handle returned by the `load_vbx` call. The `name` is the string name of the VBX variable and the `value` is the new floating point to write to the VBX.

See also

set_vbx_integer_property
set_vbx_string_property

set_vbx_integer_property

Purpose

Modifies a vbx integer value.

Syntax

```
set_vbx_integer_property(HCTL handle,char *name,int value )
```

Description

This function allows you to set integer values in a previously loaded VBX. The `HANDLE` is the handle returned by the `load_vbx` call. The `NAME` is the string name of the VBX variable and the `VALUE` is the new integer to write to the VBX.

See also

set_vbx_floating_property
set_vbx_string_property

set_vbx_string_property

Purpose

Modifies a vbx string value.

Syntax

```
set_vbx_string_property( HCTL handle,char *name, char *value )
```

Description

This function allows you to send strings to a previously loaded VBX. The `HANDLE` is the handle returned by the `load_vbx` call. The name is the string `NAME` of the VBX variable and `VALUE` is the new string to write to the VBX.

See also

set_vbx_integer_property
set_vbx_floating_property

size_in_pixels

Purpose

Set font size in pixels.

Syntax

```
size_in_pixels( int height, int 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

Set font size if points.

Syntax

```
size_in_points( int height, int width )
```

Description

Sets the *height* and *width* of a font that is being used in a graphics region.

See also

size_in_pixels
select_font

sizeof_clipboard_text

Purpose

Returns the size of the CF_TEXT/CF_OEMTEXT held in the clipboard

Syntax

```
int sizeof_clipboard_text()
```

Description

This routine returns the length of the Windows clipboard text array. If there is no text in the clipboard a value of -1 is returned. If there is text available then the length of the text is returned. The length value is 1 greater than the actual length of text as this accounts for a *NULL* character which is placed at the very end of the string. This character has an ASCII value of 0. It is therefore necessary to allocate a buffer one character less than that returned by this function.

See also

copy_from_clipboard

sound_playing

Purpose

Indicate ongoing sound output.

Syntax

```
int sound_playing()
```

Description

This function returns 1 while there is sound output and 0 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.

See also

write_wave_file
record_sound
play_sound
sound_recording
select_sampling_rate
play_sound_resource
sound_sample_rate

sound_recording

Purpose

Indicate a recording is being made.

Syntax

```
int sound_recording()
```

Description

This function returns 1 while the sound system is recording and 0 when it is idle.

See also

write_wave_file
record_sound
sound_playing
play_sound
select_sampling_rate
play_sound_resource
sound_sample_rate

sound_sample_rate

Purpose

Returns the current sound sampling rate.

Syntax

```
int sound_sample_rate( )
```

See also

write_wave_file
record_sound
play_sound
sound_recording
select_sampling_rate
play_sound_resource

stop_audio_cd

Purpose

Stops audio playback from the CD.

Syntax

```
stop_audio_cd()
```

See also

close_cd_tray
open_cd_tray
play_audio_cd
set_cd_position

temporary_yield

Purpose

To replace the `yield_program_control`.

Syntax

```
temporary_yield
```

Description

This routine replaces the `YIELD_PROGRAM_CONTROL@` routine that required a value of 1 in most cases. No parameters are required.

underline_font

Purpose

To add an underline to graphics region text.

Syntax

```
underline_font( int active )
```

Description

When using the function `draw_text` with any font selected other than a Hershey font, in a graphics region, a call to this routine with a value of 1 will make any further text output underlined. A further call with a value of 0 will deactivate this function.

See also

rotate_font
scale_font
select_font
italic_logfont
bold_font
select_logfont

use_rgb_colours

Purpose

Switches between palette entry and `RGB` colours when drawing in graphics regions.

Syntax

```
int use_rgb_colours( int handle, int bool )
```

Description

If `handle` is zero then the current graphics region is assumed. The `BOOL` can be either 1 or 0. If it is 1 then the `RGB` colours definition will be used else colour numbers will be treated as a palette reference.

See also

%gr

write_wave_file

Purpose

Store any recorded data.

Syntax

```
write_wave_file( char *filename, int chan, int sample,
short int *left, short int *right )
```

Description

The data contained in the *left* and *right* variables is stored to disk in a wave file format. The *filename* should be set to the desired name. There can be either 1 (mono) or 2 (stereo) channels. The sample is the length of either the left or right data array because they should both be of identical length.

See also

sound_recording
record_sound
sound_playing
play_sound
play_sound_resource
set_sound_sample_rate

ClearWin+ tutorial

This tutorial shows you how to develop a simple Windows application using Salford C++ and ClearWin+. It takes you through a step-by-step process leading to a program that interactively factorises integers. 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.cpp*.

The tutorial is based on a number of files called *factor1.cpp*, *factor2.cpp*, etc. that can be found in the \cwp4demo\winio\tutorial directory 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.cpp* and then read the explanation for this file. Then proceed to *factor2.cpp* and so on.

{button Begin ClearWin+ tutorial,Next()}

ClearWin+ tutorial: Step 1

Begin by loading your version of Windows. Open a DOS box and load DBOS in it. You can do this by double clicking on the DBOS icon in the ClearWin group of the Program Manager or [START] [PROGRAMS] [CLEARWIN] for Windows 95. If you want to run Windows 3.1(1) executable files from the DOS box command line, you should also type:

```
HOTKEY77 /W
```

Windows 95 will allow you to start any compatible executable program from the command line.

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

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

```
1  // factor1.cpp
2  #pragma windows 300000,500000,"cwp_ico.rc"
3  #include <windows.h>
4
5  int main()
6  {
7     winio("%ca[Number Factoriser]");
8     return 0;
9  }
```

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

```
scc factor1 /link
```

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

```
factor1
```

from the DOS box command line, provided that you have HOTKEY77 loaded as described above. If HOTKEY77 is not loaded then you can type

```
RUN77 FACTOR1
```

Windows 95 will allow you to run the executable without first starting HOTKEY77.

Alternatively in windows 3.1(1) you can run the executable from the Program Manager. To do this select File and then Run... from the Program Manager menu. Use the Browse... button to find the executable that you want to run and then click on the OK button.

Windows 95 allows you to similarly run programs by just selecting [START] [RUN]. You should the either enter the program with its full path or use the browse function.

factor1.cpp simply opens a window with a caption.

The examples throughout this manual have all been executed under Windows 95. If you are using other versions of windows the display will not be totally identical.

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 2 provides information for the linker that is called when you use /LINK on the SCC command line and it causes the linker to produce a Windows executable. The value 300000 (decimal) specifies the maximum stack size and 500000 (decimal) specifies the maximum heap size used by the application. In C++, the stack is used for local (dynamic) variables whilst the heap is used by some ClearWin+ functions and when you allocate memory using standard functions like `malloc` and `new`. The size of these values is usually not critical so we set them to be suitably large. An error will be reported in due course if the selected values have been made too small.

cwp_ico.rc is the name of what is called a resource script file that can be found in the same directory as *factor1.cpp*. 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 so that the application can be installed in the Program Manager.

Line 2 must be the first line in the file that is not a comment

Line 3 is a call to include the file *windows.h*. The diamond brackets indicate that the file is located in the system directory (called *drive:\dbos.dir\include* by default). This file contains type declarations for all the Windows routines and parameters. The only information of relevance to this particular main program is that `winio` returns a result of type `int`.

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 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 nightmare of writing a GUI application based on calls to the standard API library combined with a detailed resource script and complex call-back functions.

Experienced C programmers will soon recognise the similarity between `winio` and `printf`. 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 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 the two letter format code. Additional information can be 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.

{button Next step,Next()}

ClearWin+ tutorial: Step 2

Here is the text of *factor2.cpp*.

```
1  // factor2.cpp
2  #pragma windows 300000,500000,"cwp_ico.rc"
3  #include <windows.h>
4
5  int number=1;
6
7  int main()
8  {
9     winio("%ca[Number Factoriser]&");
10    winio("%il&",1,2147483647);
11    winio("Number to be factorised: %rd",&number);
12    return 0;
13 }
```

Note that `NUMBER` has been added to line 5 and & has been inserted towards the end of line 9. Lines 10 and 11 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 & in line 9 is simply a device to allow the description of the window to continue in the next call to `winio` (on line 10 in this case). By using this device we can avoid long format strings with long lists of associated arguments.

Line 10 contains the format code %il. This specifies the *integer limit* for the input on line 11. %il takes two (`int`) 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. Integer arguments for `winio` are always taken to be of type `int` whilst floating point values are taken to be of type `double`. The & allows continuation in the next `winio` call.

In line 11 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 the application.

The %rd format code is used for user integer input. It creates an edit box and displays the value of the `int` variable (in this case `number`) that is provided as the next argument of `winio`. The initial value of `number` in line 5 has been chosen to be in the permitted range. The user will find that he is 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.

{button Next step,Next()}

ClearWin+ tutorial: Step 3

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

```
1  // factor3.cpp
2  #pragma windows 300000,500000,"cwp_ico.rc"
3  #include <windows.h>
4
5  int factoriser();
6  int number=1;
7
8  int main()
9  {
10    winio("%ca[Number Factoriser]&");
11    winio("%il&",1,2147483647);
12    winio("Number to be factorised: %rd&",&number);
13    winio("\t%`^bt[Fac&torise]",factoriser);
14    return 0;
15 }
16
17 int factoriser()
18 {
19    return 2;
20 }
```

Note that lines 5, 13 and 17 to 20 have been added to *factor2.cpp*. Also an & has been inserted after %rd on line 12. We need to look at line 13 in detail. The other new lines simply declare a new function which for the moment merely returns the value 1.

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

Line 13 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 & 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 can be used with %bt, but the number of modifiers that a format code can take and the effect each modifier has varies from one 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.

Finally we note that a tab (represented by \t) has been placed before the %bt format code. This is used to space out the button from the text before it.

A call-back function may return any of the following values:

0 Closes the window and returns to the calling `winio` call.

1 The window remains open and the whole screen is refreshed to allow any changes to be displayed. If this values is ‘over used’ it will result in a flashing effect from the display.

2 The window is left open with no refreshes. If anything needs to be updated before the call-back returns a call to `window_update` will refresh individual components of the display.

{button Next step,Next()}

ClearWin+ tutorial: Step 4

The next thing we need is a child window in order to present the results of the factorising process. *factor4.cpp* 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.cpp*:

```
1  // factor4.cpp
2  #pragma windows 300000,500000,"cwp_ico.rc"
3  #include <windows.h>
4  #include <stdio.h>
5
6  int factoriser();
7  int number=1;
8  char str[50]="";
9
10 int main()
11 {
12    winio("%ca[Number Factoriser]&");
13    winio("%il&",1,2147483647);
14    winio("Number to be factorised: %rd&",&number);
15    winio("\t%`^bt[Fac&torise]&",factoriser);
16    winio("\n\n%ob%42st%cb",str);
17    return 0;
18 }
19
20 int factoriser()
21 {
22    sprintf(str,"%d",number);
23    window_update(str);
24    return 21;
25 }
```

Lines 4, 8, 16, 22 and 23 have been added and as before an & has been placed at the end of the format string on line 15.

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 definition of *number* as a global variable on line 7 provides a means of passing this variable to the *factoriser* function. You will recall that call-back functions like *factoriser* do not have any arguments. Line 22 stores the value of *number* as a character string pointed to by *str*. *str* is also global because it is used for output on line 16:

```
winio("\n\n%ob%42st%cb",str);
```

On line 16, \n provides a line feed so with two of them the child window that follows is spaced out 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 at 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 23 has the effect of updating the string on the screen in order to reflect a change in the contents of `str`.

{button Next step,Next()}

ClearWin+ tutorial: Step 5

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

```
7  int about();

13    winio("%ca[Number Factoriser]&");
14    winio("%mn[&File[E&xit]]&","EXIT");
15    winio("%mn[&Help[&About Number Factoriser]]&",about);
16    winio("%il&",1,2147483647);

30 int about()
31 {
32    winio("%ca[About Number Factoriser]&");
33    winio("%fn[Times New Roman]%ts%bf%cnTutorial&",2.0);
34    winio("%ts\n\n\n\n&",1.0);
35    winio("%cnProgram written to demonstrate\n\n&");
36    winio("%ts%tc%cn%bfClearWin+&",1.5,RGB(255,0,0));
37    winio("%tc%sf\n\n%cnby&",-1);
38    winio("\n\n%cnSalford Software&");
39    winio("\n\n%cn%9`bt[OK]");
40    return 1;
41 }
```

A new call-back function called `about` has been added at line 7 and the code for this function has been added to the end of *factor4.cpp* in lines 30 to 41. The new menu bar has been included as lines 14 and 15 after the caption in line 13.

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 14 and 15 for the menu bar.

```
14    winio("%mn[&File[E&xit]]&","EXIT");
15    winio("%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 & symbol 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 `winio` . The first one "EXIT") refers to 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 37. %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 argument of type `double`. The value 2.0 on line 33 doubles the standard text size and then the standard size is restored on line 34 with the value 1.0. Line 36 uses a text size of one and a half times the standard and then the standard size is restored with %sf

on line 37.

%bf on line 33 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 (\n). On line 36, %tc changes the colour of the text to red. The colour is provided by the call to the API macro called `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 1.5 argument on line 36. %tc is also used on line 37. In this case the argument (-1) restores the system text colour. This is the colour that the user can set from the Control Panel in the Program Manager.

Finally line 39 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.

{button Next step,Next()}

ClearWin+ tutorial: Step 6

*factor6.cpp*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 C code are not of direct interest to us in this tutorial but here is the code for those who want to read it.

```
int factoriser()
{
   int k,n=number;
   char val[10];
   sprintf(str,"The factors of %d are: 1",number);
   while(n>1)
   for(k=2;k<=n;++k)
   {
     if(n%k==0)
     {
       sprintf(val,",%d",k);
       strcat(str,val);
       if((n/=k)>1) break;
     }
   }
   window_update(str);
   return 2;
}
```

This brings us to the end of the tutorial. In this one demonstration program you have been introduced to some of the most 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+.

Compiler options and directives

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). The other method is to use one of the SCC command line options /LINK and /LGO and to insert the "#pragma windows" compiler directive into the program code before the main program.

The #pragma directive takes the form:

```
#pragma windows [<stack_size, heap_size>][,<resource_file>]
```

The /WINDOWS command line option becomes redundant when this directive is used. When used, this should be the first line in the main program file that is not a comment. *stack_size* and *heap_size* are specified as decimal values (note that corresponding instructions within LINK77 described below, use hexadecimal values for the stack and heap size). The default values for *stack_size* and *heap_size* are 64Kb (10000 hex) and 320Kb (50000 hex) respectively. The resource script file is optional and if specified, the SRC command will be issued to compile the resource script. If not specified, the application will simply be marked as a Windows application.

All C/C++ programs should include the line

```
#include <windows.h>
```

The /WINDOWS command line option becomes redundant when this directive is used. When used, this should be the first line in the main program file that is not a comment. stack_size and heap_size are specified as decimal values (note that corresponding instructions within LINK77 described below, use hexadecimal values for the stack and heap size). The default values for stack_size and heap_size are 64Kb (10000 hex) and 320Kb (50000 hex) respectively. The resource script file will be issued to compile the resource script. If not specified, the application will simply be marked as a Windows application.

All C/C++ programs should include the line

```
#include <windows.h>
```

.if they make reference to any of the Windows API or ClearWin+ functions. Note the use of "<" and ">" to denote that the default directory for system include files should be searched.

LINK77 commands

If the /LINK and /LGO compiler command line options are not used, LINK77 must be called from the command line in order to link programs.

The first line must contain the WINDOWS_STACK (you can use the abbreviation WS) command to tell LINK77 that it is linking a Windows application. The WS command has the following general form:

```
WS [<stack_size>] [<heap_size>]
```

where both *stack_size* and *heap_size* are optional. In the linker, both parameters must be specified in hexadecimal.

The linker command RC will start the Salford Resource Compiler in a similar manner to the WINAPP compiler directives described above. The form of the command is

```
RC <resource_file>
```

This command must always be used whenever a windows executable is being created. The resource file name may be omitted from the command when there is no resource file to compile.

The following example shows the commands required to compile and link a simple Windows application:

```
SCC MYAPP /WINDOWS
LINK77
WS
LO MYAPP
RC MYAPP.RC
FILE
```

The file *myapp.rc* contains a definition of the “resources” used by the program (see SRC).

The fully linked application will not run in the DOS environment and will only run in Windows enhanced mode with the Virtual Device Driver WDBOS.386 present.

The setting of a maximum stack size is not necessary when linking DBOS applications for use in a DOS environment because DBOS programs run in a 2 gigabyte address space with the stack descending from the highest address. A program will run out of total physical memory (giving the DBOS message “page memory exhausted”) well before the stack overwrites the user’s program. This is because under DOS, DBOS supports any number of non-contiguous regions of memory within the address space, and does not require physical memory to be associated with the unused spaces in between.

In contrast, the Windows environment is less sophisticated in this respect, and it is necessary to allocate a contiguous block of memory for the program. The stack is allocated beneath the program, so, if the stack overflows, you will get a sensible diagnostic because the overflowed stack will not overwrite the program code.

Using the Salford Resource Compiler

The RC command within LINK77 creates a temporary file containing the compiled resource. If the resource script is not being changed then it is clearly unnecessary to recompile the resource each time the linker is used. In such a case the Salford Resource Compiler SRC can be called from the command line in order to create a permanent compilation of the resource script. SRC uses the same resource script syntax as the Microsoft Resource Compiler RC.

Once a resource script has be written, it should be stored in a file with the .RC extension. The resource can then 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 a Fortran filename in the same directory and project. (This restriction does not apply when using the RC command from within LINK77.) The object file can then be linked with other modules using LINK77 as follows.

```
LINK77
WS
LO MYAPP
LO RESOURCE
FILE
```

Windows Memory Allocation

When you define the stack and heap sizes for a windows program is it important to remember that any dynamic memory will come from the heap. So that if you define the heap size to be 500000(hex) (5 Mega Bytes) then you `malloc` and `new` will only be allowed to allocate up to this amount of memory regardless of the amount of actual memory installed.

Calling 16-bit DLL functions from C/C++

Routines exported from 16-bit DLL's can be called from Salford C++. The method requires you to call the **create_windows_stub** function which will generate a stub and provide an address of the stub routine which you can then call.

The following code illustrates how this function might be used to call the **MessageBox** function in the USER dynamic link library (called USER.DLL).

```
win_int (*mBox)(HWND,LPSTR,LPSTR,UINT);
win_int rval;

create_windows_stub("MessageBox","USER","SSLLS",mBox);

if (mBox) // stub created
  rval=mBox(hWnd,"It works","Call Succeeded",MB_OK);
else      // could not create stub
  error
```

Clearwin Window functions: Introduction

This chapter contains the definitions of *ClearWin* window functions that can be used in conjunction with the functions described in chapter 4. Other *ClearWin* functions that have been superseded (mostly by the introduction of `winio`) are described in chapter 8.

The functions in this chapter relate to so-called *ClearWin* windows. 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 C/C++ I/O routines `scanf` and `printf`. Note that a *ClearWin* window is not the same as a window created with `winio` (that is a *Format* window). It is, however, possible to embed a *ClearWin* window in a *Format* window.

Graphics can be drawn to a *ClearWin* window by using `get_graphics_dc` (chapter 8) and by calling Windows API drawing functions. However, it is recommended that graphics objects should be draw to a *Format* window.

A *ClearWin* window (that is not embedded in a *Format* window) can be created by calling `create_window` (this function returns a standard *Windows* window 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 `win_printf` can be used to attach a Fortran unit number to a particular *ClearWin* window.

It is strongly recommended that the function `set_all_max_lines` be used before creating any *ClearWin* windows (including default *ClearWin* windows). This limits the amount of store that is consumed.

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:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main()
{
  double d;
  char str[80];
  while(1)
  {
    gets(str);
    d=atof(str);
    printf("%g\n",sqrt(d));
  }
}
```

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. This is 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 the WS command in LINK77 or the "`#pragma windows`" directive can be used to allocate a larger heap. Store 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 FTN77 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, by default, it is created full screen. 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. This example uses a Windows API function to determine the screen dimensions:

```
...#include <windows.h>
int main()
{
   int   win;
   short xsize, ysize, xpos, ypos;

   xsize=GetSystemMetrics(SM_CXSCREEN)/2;
   ysize=GetSystemMetrics(SM_CYSCREEN)/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 may inform the system that your window is the default *ClearWin* window, thus:

```
set_default_window(win);
```

When your program terminates, the *ClearWin* windows that it has created will remain open for viewing, scrolling etc. until closed by the user (by pressing Alt-F4).

Text output to ClearWin windows

Textual window input/output uses, by default, a Windows font called `SYSTEM_FIXED_FONT` and black text. 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:

```
#include <windows.h>
#define RED 255
int main()
{
   int  lfheight,lfwidth,lfescapement,lforientation,
        lfweight,lfitalic,lfunderline,lfstrikeout,
        lfcharset,lfoutprecision,lfclipprecision,
        lfquality,lfpitchandfamily,win;
   char  lffacename[80];
   short newfont;

   get_system_font(lfheight,lfwidth,lfescapement,
                   lforientation,lfweight,lfitalic,
                   lfunderline,lfstrikeout,lfcharset,
                   lfoutprecision,lfclipprecision,
                   lfquality,lfpitchandfamily,
                   lffacename);
/*
  alter the default parameters to double the
  size and specify italic
*/
   lfheight=lfheight*2;
   lfwidth=lfwidth*2;
   lfitalic=1;
   newfont=CreateFont(lfheight,lfwidth,lfescapement,
                      orientation,lfweight,lfitalic,
                      lfunderline,lfstrikeout,
                      lfcharset,lfoutprecision,
                      lfclipprecision,lfquality,
                      lfpitchandfamily,
                      lffacename);
/*
  attach a text descriptor to the window
*/
   add_text_descriptor(win,newfont,RED);
      ...
```

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`.

Graphical output to ClearWin windows

ClearWin+ can also be used to produce graphical output in a *ClearWin* window but the alternative method via a *Format* window is preferred (see for example the %dw and %gr format codes).

To use a *ClearWin* window for graphical output you should call the routine `get_graphics_dc` to obtain a device context to a bit map of the same size as the window. Thereafter, calls to Windows API GDI routines, such as **TextOut**, will update this memory bitmap. When `update_window` is called, or the window is updated for some other reason, the bitmap will be copied to the window before anything else.

In general, when using *ClearWin* windows, it is better to reserve a *ClearWin* window for pure graphics rather than mix graphics with scrolling text. The memory for the bitmap is taken from Windows, rather than from the program heap, so you do not need to increase the second argument of the WS command in the linker (or the "`#pragma windows`" directive) in order to cater for the memory required. When the *ClearWin* window is finally closed (or the program terminates) the device context and bitmap are returned to the system. *Do not attempt to return them explicitly.*

clear_window

Purpose

To clear a given *ClearWin* window to its given background colour.

Syntax

```
#include <windows.h>
void clear_window(clearwin_window 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`.

create_window

Purpose

To create and display a *ClearWin* window.

Syntax

```
#include <windows.h>
clearwin_window create_window(char* title,int x,int y,
int xw,int yw);
```

Description

Creates a window of width *xw*, height *yw* and title *title* at position (*x*,*y*). The value returned by this function is a Windows 3.1 window handle and can be used as such.

Do not confuse this function with the similarly named routine in the Windows API called CreateWindow.

destroy_window

Purpose

To close and destroy a displayed *ClearWin* window.

Syntax

```
#include <windows.h>
void destroy_window(clearwin_window 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`

get_default_window

Purpose

To return the handle of the current default *ClearWin* window for standard I/O.

Syntax

```
#include <windows.h>
clearwin_window get_default_window(void);
```

set_all_max_lines

Purpose

To set the maximum number of lines to be stored for all *ClearWin* windows.

Syntax

```
#include <windows.h>
void set_all_max_lines(int n)
```

Description

This routine is the same as `set_max_lines` but applies to all *ClearWin* windows.

set_max_lines

Purpose

To set the maximum number of lines to be stored for a given *ClearWin* window.

Syntax

```
#include <windows.h>
void set_max_lines(int w,int 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_default_font

Purpose

To set the default standard I/O font for *ClearWin* windows.

Syntax

```
#include <windows.h>
void set_default_font(HFONT hfont);
```

Description

Sets the default standard I/O font 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_to_fixed_font

Purpose

To reset the default standard I/O font for *ClearWin* windows to ‘‘System Fixed’’.**Syntax**

```
#include <windows.h>
void set_default_to_fixed_font(void)
```

Description

This is a mono spaced font which is the default for *ClearWin* window standard I/O.

set_default_proportional_font

Purpose

To set the default standard I/O font for *ClearWin* windows to “System”.

Syntax

```
#include <windows.h>
void set_default_to_proportional_font(void);
```

Description

This is a proportionally spaced font. You can revert to a mono spaced font by using `set_default_to_fixed_font`.

set_default_window

Purpose

To set the default *ClearWin* window for standard I/O.

Syntax

```
#include <windows.h>
clearwin_window set_default_window(clearwin_window& w);
```

Description

All future standard I/O requests will be directed to this window.

update_window

Purpose

To redraw a *ClearWin* window.

Syntax

```
#include <windows.h>
void update_window(clearwin_window 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 Updating Windows) that is used for *Format* windows.

win_coua

Purpose

To output *n* characters from a string to a *ClearWin* window.

Syntax

```
#include <windows.h>
void win_coua(clearwin_window w, char* str, int n);
```

Description

Outputs *n* characters from the string *str* to the ClearWin window with handle *w*.

win_getchar

Purpose

To read a character from the keyboard.

Syntax

```
#include <windows.h>
char win_getchar(clearwin_window w);
```

Description

Changes the input focus to the ClearWin+ window with handle *w* and reads a character from the keyboard

win_getline

Purpose

To read a line of text from the keyboard.

Syntax

```
#include <windows.h>
void win_getline(clearwin_window w,char* line);
```

Description

Changes the input focus to the ClearWin+ window with handle *w* and reads a line of text from the keyboard.

win_printf

Purpose

To output formatted text to a window.

Syntax

```
#include <windows.h>
void win_printf(clearwin_window w, char* format,...);
```

Description

This is functionally the same as `printf` but the output is directed to window *w* rather than to the default window.

Introduction

This chapter contains the definitions of additional C/C++ functions that augment the standard Microsoft Windows API. These functions are presented in alphabetical order after the next section which deals with the subject of yielding control to other concurrent tasks.

Yielding control with ClearWin+

If your program performs a substantial amount of computation it is important to consider yielding control to Windows from time to time. This enables the system to respond to user input, such as requests to switch to another task, move a window on the screen etc.. A call to `yield_program_control` with the flag `Y_TEMPORARILY` as argument will give the system a chance to catch up with any housekeeping tasks before returning to your program.

Usually a little trial and error will show where calls to `yield_program_control` are required. The only awkward case is where a program spends a great deal of time in one tight loop. Putting the call outside the loop is ineffective, and putting the call inside the loop adds an impossible overhead to the program. In cases such as this it may be necessary to adjust the code thus:

The *yield_count* threshold value of 1000 should be adjusted so that the mouse response is adequate but the program overhead is minimal.

create_windows_stub

Purpose

To provide access to third party DLL functions on a dynamic basis.

Syntax

```
#include <windows.h>
int create_windows_stub(char* name,char* module,
char* arguments, void*& fn_pointer);
```

Description

This function dynamically creates a 32-bit stub procedure to enable routines in third party DLLs to be called.

The function takes the following arguments :

name

This is the name of the function in the DLL that you wish to call. Please note that this routine must be exported in the DLL.

module

This is the module name i.e. the name of the DLL of where the above specified function resides.

arguments

This a character array which will represent the arguments and return type of the specified function. Each character in this array denotes a specific size of argument and the first character is the return type.

The valid characters are as follows :-

V this denotes a **void** type and is only valid for the first character, i.e. the return type.

S this denotes a two byte value, the usual cases for this are **int**, **BOOL** and **short**.

L this denotes a four byte value, the usual cases for this are **long**, **LPSTR** etc.

fn_pointer

On return, this will be the address of the stub for the function. If for any reason a stub could not be created, the value 0 (zero) is returned.

ExportProc

Purpose

To return the address of a ClearWin+ function.

Syntax

```
#include <windows.h>
void WIN* ExportProc(void* fn);
```

Description

Returns the address of a ClearWin+ function *fn* in a form that is ready to be passed to a Windows API function. The function should take the standard four arguments of a Windows call-back function (see add_window_callback).

For call-back functions that have a different set of arguments than this, use **ExportProcExtExportProcExt**.

ExportProcExt

Purpose

To return the address of a ClearWin+ function.

Syntax

```
#include <windows.h>
void WIN* ExportProcExt(void* fn,char* args);
```

Description

Returns the address of a ClearWin+ function *fn* in a form that is ready to be passed to a Windows API function. *args* describes the return type of the function and the argument list. Each argument in the argument list is represented by an S or an L depending on whether it is a long argument (L) or a short argument (S). The return type is the first character in the list and is also represented by an S or an L.

Example

The Windows API function **GrayString** requires a callback function with the following specification:

```
BOOL CALLBACK GrayStringProc(HDC hdc,LPARAM lpData,
int cch)
```

To provide your own ClearWin+ function called, for example **MyGrayStringProc** to **GrayString** you would have to create the address using

```
ExportProcExt(MyGrayStringProc,"SSLS")
```

get_system_font

Purpose

To get the attributes of the system font.

Syntax

```
#include <windows.h>
void get_system_font(int& lfHeight,
 int& lfWidth, int& lfEscapement,
 int& lfOrientation, int& lfWeight, int& lfItalic,
 int& lfUnderline,
 int& lfStrikeOut, int& lfCharSet,
 int& lfOutPrecision, int& lfClipPrecision,
 int& lfQuality, int& lfPitchAndFamily,
 char* 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.

GetCurrentTaskLong

Purpose

To return the handle of the current task together with the task queue selector.

Syntax

```
#include <windows.h>
long getCurrentTaskLong(void);
```

Description

This is the same as the Windows API function GetCurrentTask except that it returns a **long**.

The top 16 bits are a selector that points at the start of the task queue (refer to the TDB description in the text “Undocumented Windows” for details, see page 28). You can achieve the same effect by altering the prototype of GetCurrentTask to return a **DWORD**, but this lets you keep both functions.

hInst_to_hTask

Purpose

To return the instance handle for a given task handle.

Syntax

```
#include <windows.h>
HTASK hInst_to_hTask(HINST hInst);
```

Description

This routine can be used to obtain the current task handle following a call to `windows_instance` to obtain the current instance handle. The task handle is required when sending messages to other applications.

hTask_to_hInst

Purpose

To return the task handle for a given instance handle.

Syntax

```
#include <windows.h>
HINST hTask_to_hInst(HTASK hTask);
```

memcpyln

Purpose

To copy *n* bytes of data.

Syntax

```
#include <windows.h>
void memcpyln(void* lptr,unsigned short nptr,int n);
```

Description

Copies *n* bytes from the string pointed to by the 32-bit ClearWin+ pointer *lptr* to the string pointed to by the Windows 16-bit near pointer *nptr*.

memcpynl

Purpose

To copy *n* bytes of data.

Syntax

```
#include <windows.h>
void memcpynl(unsigned short nptr,void* lptr,int n);
```

Description

Copies *n* bytes from the string pointed to by the Windows 16-bit near pointer *nptr* to the string pointed to by the 32-bit ClearWin+ pointer.

nstrlen

Purpose

To return the length of the NULL terminated string.

Syntax

```
#include <windows.h>
short nstrlen(unsigned short nptr);
```

Description

Finds the length of the NULL terminated string pointed to by the Windows 16-bit near pointer *nptr*.

windows_instance

Purpose

To return the instance handle of the application.

Syntax

```
#include <windows.h>
int windows_instance(void);
```

yield_program_control

Purpose

To yield program control back to Windows.

Syntax

```
#include <windows.h>
int windows_instance(void);
```

Description

This functionshould be used to ensure that a program does not totally take over the CPU, thus preventing other applications from running. *option* may be either `Y_PERMANENTLY` or `Y_TEMPORARILY`. The vast majority of times `Y_TEMPORARILY` will be used. Some trial and error will be needed to determine how often **`yield_program_control`** should be called.

Windows is not a pre-emptive operating system. That is, Windows cannot itself transfer between applications to surrender control regularly so that other applications are given a chance of running.

Other ClearWin functions

This chapter contains the definitions of ClearWin 3.1 functions that have not been described in earlier chapters. These functions are considered to have been superseded by the introduction of `winio` in ClearWin+.

add_text_descriptor

Purpose

To add a text descriptor to a window.

Syntax

```
#include <windows.h>
void add_text_descriptor(clearwin_window w,int font,
int colour);
```

Description

Adds a text descriptor to the window with handle *w* (this value is returned by the ClearWin+ function create_window). Text descriptors are activated by using '@tn' where n is a one or two digit number. This is an instruction to switch to the n'th text descriptor for the given window. Text descriptors simplify the task of changing the text style. They are embedded in the string to be output and affect all text subsequently output. If the use of the @ character is inconvenient, then a different escape character may be nominated using `set_win_escape`.

font is a font handle returned, for example, by the API function CreateFont. *colour* is the value returned by the Windows API macro `RGB`. This function can be used in association with standard C/C++ output functions and the ClearWin+ function win_coua.

add_window_callback

Purpose

Adds a call-back function for a given *ClearWin* window and Windows message.

Syntax

```
#include <windows.h>
void add_window_callback(clearwin_window w,int message,
void* userfunc);
```

Description

Adds a call-back function for the window with handle *w* and the Windows message *message*. *w* is the value returned by the ClearWin+ function create_window. message is one of the standard API messages such as WM_COMMAND (these are constants defined in windows. h). The call-back function should take the standard four arguments of a Windows call-back function in the form:

```
long userfunc(HWND,UINT,WPARAM,LPARAM)
```

The most common message to be trapped will be `WM_COMMAND` although other messages are also good candidates for trapping.

add_window_menu

Purpose

To set the menu in a given window.

Syntax

```
#include <windows.h>
void add_window_menu(clearwin_window w,int menu);
```

Description

Sets the menu in the given ClearWin window with handle *w*. In order to use this function, *menu* must have been obtained from a previous call to one of the Windows API functions **LoadMenu** or **CreateMenu**.

Menus can be created dynamically, with some difficulty, using **CreateMenu** or loaded from the program resource using **LoadMenu**. It is, of course, the easiest option (apart from using `winio`) to predefine menus in the resource file and load them from the program resource using **LoadMenu**.

Notes

It is normally simpler to use an alternative function called `attach_menu` which combines the actions of **LoadMenu** and `add_window_menu`.

attach_menu

Purpose

To load a named menu from the program resource.

Syntax

```
#include <windows.h>
void attach_menu(clearwin_window w, char* menu_name,
void* userfunc);
```

Description

This routine loads a menu and attaches it to the ClearWin window *w*. *userfunc* is the address of a call-back function with the standard four arguments (see `add_window_callback`).

get_filename

Purpose

To display a dialog box from which a filename can be selected.

Syntax

```
#include <windows.h>
win_int get_filename(clearwin_window handle, char *title,
BOOL check_exists,char *buffer, const int &buffersize);
```

Description

handle is the handle of the parent window.

title is the caption for the dialog box.

check_exists is `TRUE` or `FALSE` depending on whether you wish the dialog box to check if the selected file exists (mainly used when opening files).

buffer is a character array for initially specifying the directory and wild card. On successful return, this will contain the name of the file selected.

buffersize is size of the buffer.

get_graphics_dc

Purpose

To return a device context to a bitmap.

Syntax

```
#include <windows.h>
HDC get_graphics_dc(clearwin_window w);
```

Description

Returns the handle of a device context which is a bitmap with the same size as the given window. This handle can be used in subsequent GDI functions of the API in order to draw to the window (compare the API function called **GetDC**). The contents of this bitmap will be copied to the associated window each time the window is updated. *w* is the ClearWin window handle returned by `create_window`.

get_window_handle

Purpose

This function is redundant in ClearWin+.

Syntax

```
#include <windows.h>
HWND get_window_handle(clearwin_window w)
```

Description

This function is provided for compatibility with ClearWin 3.0 and simply returns its argument. *ClearWin* window handles and Windows 3.1 window handles have the same value.

identify_window_handle

Purpose

This function is redundant in ClearWin+.

Syntax

```
#include <windows.h>
clearwin_window identify_window_handle(HWND hw);
```

Description

This function is provided for compatibility with ClearWin 3.0 and simply returns its argument. *ClearWin* window handles and Windows 3.1 window handles have the same value.

make_dialog_box

Purpose

To create and display a given dialog box.

Syntax

```
#include <windows.h>
int make_dialog_box(clearwin_window w,char* template,
void* userfunc)
```

Description

Makes and displays a dialog box which is owned by the parent window *template* is the name of a dialog box which already exists in the program resource. *userfunc* is the address of the call-back function. The call-back function is used to handle dialog box controls such as pushbuttons etc. and takes the form:

```
long userfunc(HWND&,UINT&,WPARAM&,LPARAM&)
```

set_win_escape

Purpose

To set the escape character for a specified *ClearWin* window.

Syntax

```
#include <windows.h>
void set_win_escape(clearwin_window w,char esc_char);
```

Description

Sets the escape character for the window specified to *esc_char*. The escape character is used by the ClearWin+ system to indicate a text descriptor change. The format of a text descriptor is <esc_char>tn. The default escape character is @.

The Salford resource compiler

The ClearWin+ release disc includes a copy of the Salford Windows Resource Compiler (SRC). This is the Salford equivalent of the Microsoft Windows Resource Compiler (RC) that is usually required (that is when `winio` is not used) in order to build Windows applications that include user defined *resources* such as cursors, icons, bitmaps, menus, and dialog boxes. SRC is used to compile a *resource script* that contains details of such resources. Resource scripts for SRC use exactly the same syntax as those for RC. Tools to create resources form part of the Microsoft Windows SDK.

If you do not use the recommended approach to Windows programming, based on the ClearWin+ function `winio`(see chapters 3 and 4), all but the very simplest of applications will have a resource script which contains the details of resources that will be used by your application.

For example, the resource script enables you to describe the menu system for your application and the way in which the menus relate to each other. The resource script is compiled with SRC and linked to your application using LINK77 (SRC can also be called from within LINK77) This also marks the application as a Windows 3.1 application. If one of the SCC compiler options /LINK and /LGO are used, then SRC and LINK77 will be automatically invoked provided that the "`#pragma windows`" compiler directive is included in the main program (see Compiler options and directives).

A typical resource script looks like this (the numbers at the end of each line indicate the relevant explanatory note only and do not form part of the script):

```
#include <WINDOWS.H>                                    ( 1)
#define IDM_ABOUT 100                                   ( 2)
t_icon    ICON testprog.ico                             ( 3)
OutputMenu MENU                                         ( 4)
BEGIN                                                   ( 5)
  POPUP        "&Help"                                  ( 6)
  BEGIN                                          ( 7)
    MENUITEM "&About 32-Bit test...",IDM_ABOUT          ( 8)
  END                                                   ( 9)
END                                                     (10)
AboutBox    DIALOG 22, 17, 144, 75                      (11)
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU           (12)
CAPTION "About 32-Bit test"                             (13)
BEGIN                                                   (14)
  CTEXT "32-Bit test program" -1,0,5,144,8              (15)
  CTEXT "Supplied with FTN77" -1,0,14,144,8             (16)
  CTEXT "Version 2.70"        -1,0,34,144,8             (17)
  DEFPUSHBUTTON "OK"       IDOK,53,59,32,14             (18)
END                                                     (19)
```

1) This line is necessary since it contains the **#define**s for DS_MODALFRAME, WS_CAPTION, WS_SYSMENU, IDOK and WS_GROUP.

2) This defines a message *id* number. When the ABOUT dialog box's push button is depressed, the application will receive this value in the *lparam* part of the message.

3) This line defines the icon to be used when the application is shown as a program item in a group window. The icon can also be used when your application is shown minimised but special action has to be taken to do this.

4) This names the menu resource used by the program. The name is used by the application to identify which menu is to be used. Numbers may be used, and very often are, but this requires using the Windows API function **MakeIntResource**

5) Starts the body of the menu definition. As you might expect, BEGIN and END must be paired.

6) The menu has only one item, Help. The H in help will appear underscored since it is preceded by an ampersand. This also defines H as a mnemonic key. When the mnemonic key is used or the item clicked, Windows will bring up another menu without any other action by the application.

7) Starts the definition for the popup menu on the Help item

8) Defines the text, the mnemonic key and the result of the menu item. At the bottom level, the application must interpret for itself what to do when a menu item is selected.

9) Matches the BEGIN at line 7.

10) Matches the BEGIN at line 5.

11) Names the dialog box resource. Note that although we require to link the dialog box to the menu item in the Help popup, the application must handle this itself. The dialog box is at (22,17) and is 144 dialog units wide by 75 dialog units high. For a definition of dialog units, please refer to the Windows API documentation.

12) Specifies the style of the dialog box. This particular combination gives a dialog box with a modal frame, title bar and system menu.

13) This is the title that goes into the title or caption bar.

14) Starts the body of the dialog box.

15) With (16) and (17) is the text appearing in the dialog box. The text is centred (CTEXT) in a box 144 wide by 8 high. Other options are LTEXT and RTEXT for right and left justified text. The -1 is an *id* number for the control, which is not used in this case.

16) Defines a push button containing the text "OK", with the usual form of *position_x*, *position_y*, width and height. The result IDOK will be sent to the application when the button is depressed.

17) Matches the BEGIN at line 14.

Including Resources

Resources can also be specified using:

```
#pragma resource
```

at the end of the program.

The following list contains the resources that are most commonly used with Clearwin+:

```
HYPERTEXT
WAVE
ICON
BITMAP
```

examples of their use are given throughout chapter 4.

Text strings

Text strings may be placed in the resource file using the STRINGTABLE keyword. There must not be more than one string table declaration in the resource script. The strings should be numbered and retrieved using the Windows API **LoadString** function. Call **MakeIntResource** with the string number as argument, as is usual with numbered resources, when fetching a string. The strings need not be numbered consecutively.

For example:

```
STRINGTABLE
{
  1,     "Hello world"
  3,     "Goodbye cruel world"
  5,     "Exit taken"
}
```

Placing text strings in the resource script file may seem like making life difficult just for the sake of it but there is a good reason why you may wish to do so. Namely that changes can be made to the application's text without needing to change the program source. Error messages and window captions are obvious examples. Changes of this kind can be implemented via the resource compiler alone. There is, however, a more compelling reason for using a STRINGTABLE; that of national language independence.

National language independence

Resource scripts can make the task of producing national language versions of your application much easier. All your text strings can be put into the resource script. You will then have a different script file for each language version that you intend to produce. The text strings can be retrieved at run time and formatted as necessary, without the need to make changes to the program source file. It is therefore only necessary to recompile the resource script and attach the resource binary.

You should be aware, however, that producing truly national language independent applications involves many more considerations than merely the text. Keyboard handling in Windows is a major issue if national language independence is required. Several common programming practices fall down. For example, uppercasing by the following method is not a good idea at all.

```
if  (ch >= 'a' && ch <= 'z') ch = ch - 'a' + 'A'
```

Windows provides functions that take care of such matters. In this case the Windows API function **AnsiUpper** will do the job. Even moving through a character string can present difficulties. Incrementing the string pointer may just give the second byte of a multibyte character code. Chapter 3 of Petzold describes the issues involved.

The GUI

If you want your application to have the “look and feel” of many other Windows applications, you should study the volume “The Windows Interface: An Application Design Guide” supplied with the Microsoft Windows SDK. It is also helpful to study the ready-built applications supplied with Windows 3.1 to see how the menu systems work and to see, for example, how and where “check boxes”, “radio buttons”, and so on are used.

Pointers

There are 3 basic pointer types to bear in mind when programming under Windows. Two of these are Microsoft Windows pointer types, the third is the standard Salford 32-bit near pointer.

1) MS Windows near pointers: These consist of a 16-bit offset only. These usually reference items on the Windows local heap. The data selector is the 16-bit data selector given to your application when it is started up. A copy of this selector, if needed, is kept in the GS register. *Do not* destroy or alter the value of this register. The functions `memcpyln` and `memcpynl` are provided to copy to and from near pointers. It is normally possible to avoid near pointers. There are very few situations where near pointers can not be avoided. One is when a Microsoft EDIT style window is used.

2) MS Windows far pointers: These are 16:16 pointers and are the standard MS Windows pointer type. Virtually all Windows pointers are of this type. These are declared as WIN* pointers in the Salford versions of the header files. This is to indicate that a translation from or to this type is required. Although a WIN* pointer can be de-referenced within program code in the normal way, you should not pass a WIN* pointer as an argument to a Salford library routine.

3) Salford 32-bit pointers: this is the only native Salford pointer type, and is a near pointer in that the selector is implicit not explicit. You should not pass a Salford pointer type to Windows without casting it implicitly or explicitly as a WIN* type.

In a majority of cases, the header files will take care of any type casting that needs to be done, except for Windows near pointers. In the case of Windows near pointers, you will have to copy the data pointed to by the near pointer to a area pointed to by a Salford pointer before using the Salford pointer (see memcpynl and memcpyln).

Windows graphic modes

Between any Windows application and the display hardware there exists a program call a display driver. For every different make of video card there is a different display driver which 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 any 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 machines are capable of displaying at this resolution. It is however very low quality any is now hardly used.

800 x 600 x 16 or 256 colours

This is a popular mode as text is readable there is physically more display area to work.

1024 x 768 x 16 or 256 colours

This of most use when your monitor is greater than 14 inches. It is by far the most clear but can suffer from slow updates if the display card is unaccelerated.

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 can be 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 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.

Resource summary

When you create your resource file you will be able to use any of the following types:

SOUND

This type allows you to include sound samples into your program. The file that you give must be of type .WAV and it is recommended that it be less than 100KB in length..

BITMAP

This type allows you to include bitmaps.

HYPERTEXT

Attaches a hypertext document to you program. After compilation if you examine you .EXE file with a binary file viewer you will not be able to see the ASCII text that formed the hypertext document. This is because hypertext uses compression to store the data and so save memory.

ICON

This type allows you to attach icons to your program. If you are creating your own icons with an editor it is recommended that you use a 32 x 32 grid with 16 colours.

CURSOR

This type allows you to attach new cursors into you program. The following types have already been defined under windows.

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

ClearWin+ Help Contents

Contents: Introduction

Contents: ClearWin+ Tutorial

Contents: Formatted windows

Contents: Format codes

Centring format - %cn
Programmer grid format - %gd
Never down and never right formats - %nd, %nr
Pivot format - %pv
Right justifying format - %rj
Get and set window position formats - %gp, %sp
Tab position - %tl

Boxes
Box close format - %cb
Box open format - %ob

Displaying text
Font format - %fn
Get font format - %gf
Hypertext - %ht
Character formats - %it, %bf, %ul
Sterling pound symbol - %pd
Subscript format - %sd
Standard font - %sf
Superscript format - %su
Text size format - %ts
Text colour format - %tc
Variable string format - %st

Help
Help
Bubble help format - %bh
Help format - %he

Graphics
Graphics
Owner draw box format - %dw
Graphics format - %gr

Main window attributes
Caption format - %ca
Control variable format - %cv
Window handle - %hw
Leave window open format - %lw
Disable screen saver - %ns
Set window position format - %sp
Create screen saver - %sv
Set window size format - %sz
Wallpaper format - %wp
Window control format - %ww

Background colour
Background colour format - %bg

Child windows
Attach window format - %aw
Child window format - %ch
Colour palette - %cl
ClearWin window format - %cw
Create MDI frame format - %fr
Create a Property sheet - %ps
Embed in property sheet - %sh
User window - %uw

Formats for call-back functions
Closure control format - %cc
Delayed auto recall - %dl
File selection format - %fs
File filter format - %ft
Start-up call-back - %sc

Edit boxes
Edit box format - %eb
Parameter box - %pb
Text array - %tx

Third party products (VBX)
The Visual Basic Custom Control interface - %vb
Adding a VBX to a ClearWin+ window

Data Output
Data output
Output a single character - %wc
Output integer value - %wd
Output floating point exponent - %we
Output floating point values in decimal form - %wf
Output floating point value - %wg
Output character strings - %ws
Output hexadecimal values - %wx
Save settings to .INI file - %ss

Contents: Related routines

make_bitmap
make_icon
open_cd_tray
open_metafile
open_printer
open_printer1
perform_graphics_update
play_audio_cd
play_sound
play_sound_resource
record_sound
release_bitmap_dc
remove_graphics_icon
remove_menu_item
rotate_font
scale_font
scroll_graphics
see_treeview_selection
select_font
select_graphics_object
select_printer
set_cd_position
set_clearwin_float
set_clearwin_info
set_clearwin_string
set_clearwin_style
set_colours
set_graphics_selection
set_line_style
set_line_width
set_old_resize_mechanism
set_sound_sample_rate
set_vbx_floating_property
set_vbx_integer_property
set_vbx_string_property
size_in_pixels
size_in_points
sizeof_clipboard_text
sound_playing
sound_recording
sound_sample_rate
stop_audio_cd
temporary_yield
underline_font
use_rgb_colours
write_wave_file

Compiling and Linking

Contents: ClearWin window functions

Contents: Additions to the standard API

Contents: Other ClearWin functions

Contents: Resource Scripts

Contents: Using the Windows API

Contents: Appendix

Help Topic not found

The topic you selected was not found.

Windows applications

The Windows 3.1, 3.11 and 95 environments provide an important opportunity for many software developers. Windows offers an environment in which several DOS programs can be run concurrently with programs known as “Windows applications” which have been written specifically for Windows.

Windows 3.1 and 3.11 provide three modes of operation: real, standard and enhanced. All these modes allow you to run one or more 16-bit application programs, each in a DOS window (commonly called a “DOS Box”). To the DBOS programmer, only the Windows enhanced mode of operation is of real interest, as this is the mode that offers full access to the functionality of the 32-bit Intel chip. However, since Windows is only a 16-bit environment, a protected mode (32-bit) program, such as a DBOS application, cannot run *unaided* in a DOS Box. Moreover, such a program cannot be directly converted to a Windows application by using the features of Windows that are made available to 16-bit programs by the Microsoft Windows API.

Salford Software has provided two products in order to allow you to run a DBOS protected mode program in a DOS box and to create Windows applications using FTN77 or Salford C++. These are respectively the WDBOS virtual device driver, and the ClearWin+ library.

In Windows 95 and Windows NT it is now possible to create programs that are native 32 bit via Win32. Programs can be compiled with the Win32 version of the compiler.

The WDBOS virtual device driver

Unlike some other multi-tasking environments, Windows does not offer DOS programs access to the Virtual Control Program Interface (VCPI) for switching to protected mode. One of the functions of the WDBOS virtual device driver is to provide a subset of these missing functions. Using WDBOS it is possible to run one or more copies of DBOS concurrently and to switch between DBOS programs and other Windows applications within a Windows session.

WDBOS is supplied on the release diskette with version 2.60 and later of DBOS.

The ClearWin+ library

The interface between Windows and the end user may be pretty, but the same cannot be said for the interface with the programmer! If you write a Windows application, you will want to access the range of Windows features and services in the Graphical User Interface (GUI) environment. In order to allow you to do this, Microsoft has defined an API that can be accessed by Windows applications.

ClearWin+ provides an interface layer between an application and the API and supplies a library of functions and other features that enable the programmer to avoid the complexities of programming with Windows API functions. This applies both to 16-bit Windows (e.g. Windows 3.1) and to 32-bit Windows (e.g. Windows NT & 95).

For 16-bit Windows that use the so-called WIN16 API, ClearWin+ also provides an interface to enable the Windows API to be accessed from within the 32-bit code produced by the Salford family of compilers, thus making it possible to run 32-bit ClearWin+ applications under 16-bit Windows.

The ClearWin+ interface may be used at three levels:

At the first and simplest level, DOS programs that interact with the user via standard C/C++ input and output statements (for example `scanf` and `printf`) 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+'s user accessible management routines (the ClearWin+ library) may 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 quickly and easily produce one or more fully featured GUI windows (called *Formatted* windows). This one function effectively 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 call-back functions in order to control the interface. The effect is that much of the complexity of using the full API becomes transparent to the user. At this level the programmer will find that he rarely needs to use the Windows API functions.

Finally at the most sophisticated level, it is possible for the developer to use ClearWin+ to write programs that only 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. It also includes an applications launcher (called LAUNCHER) that enables any Windows 3.1 application to be run from a DOS box. This also enables the /LGO compiler option to be used for compiling, linking and running Windows applications. Windows 95 is capable of launching programs which removes the need to include LAUNCHER in the start-up.

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 rarely use mixed case names.

How to use this help file

This guide distinguishes between three types of window.

A *Windows* window is produced using only Windows API functions. The term does not imply any added functionality from the ClearWin+ library.

A *ClearWin* window is a *Windows* window that has the added ability to make direct use of standard C/C++ I/O (e.g. `scanf` and `printf`). The *ClearWin* window was the central feature of ClearWin versions 3.0 and 3.1 and is still of interest in the current ClearWin+ version 4.3.

A *Format* window is a *Windows* 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. For example, you can plant a text editor (similar to the Program Manager "Notepad" application) in a *Format* window. However, the main advantage of a *Format* window is that it is relatively easy to program. In other words, a *Format* window provides all of the standard GUI facilities whilst at the same time allowing the programmer to bypass the complexities of the Windows API.

The source level debugging system

In order to improve user efficiency and the usability of Salford products a new range of debuggers has been designed and implemented. There will be three debuggers in the range: one for MS-DOS based applications, one for Windows version 3.1 and above (including Windows for Workgroups and Win16 based Windows 95 applications) and one for Windows NT version 3.1 and above and Win32 based Windows 95 applications. All three debuggers have been designed to function consistently. Detailed information on how to use the debuggers is given the user guides for each of the Salford compilers.

BALLS.CPP

Copy to Clipboard

```
// Clearwin + 4.4 create graphics region
//                copy graphics region
//                delete graphics region
// examples. (c) salford software 1996

//
// need screen res of 800x600 !!!!!!!!!!!!
//




#pragma windows 500000,500000

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <windows.h>
#include <dbos\graphics.h>

#define SRC   0xcc0020
#define SAND  0x8800c6
#define SOR   0xee0086

int width,height,clrs,numimage,err,i;
int imghnd, grhnd=1,mask=2,ball=3,brush=5,back=6;

char fmt[4];


int scrx=20,scry=20,first=1,oldx,oldy,dirx=5,diry=5;


int redrawfunction()
{

 if (!first)
 copy_graphics_region(grhnd,oldx,oldy,50,50,back,0,0,50,50,SRC);
 copy_graphics_region(back, 0,0,50,50,grhnd,scrx,scry,50,50,SRC);
 copy_graphics_region(brush,0,0,50,50,mask,0,0,50,50,SRC);
 copy_graphics_region(brush,0,0,50,50,back,0,0,50,50,SOR);
 copy_graphics_region(brush,0,0,50,50,ball,0,0,50,50,SAND);
 copy_graphics_region(grhnd,scrx,scry,50,50,brush,0,0,50,50,SRC);
 first=0;
 oldx=scrx;
```

```
 oldy=scry;

 scrx+=dirx;
 scry+=diry;

 if (scrx>710) {dirx=-dirx;play_sound_resource("SH");}
 if (scrx<0) {dirx=-dirx;   play_sound_resource("SH");}


 if (scry>510){ diry=-diry; play_sound_resource("SH");}
 if (scry<0) {diry=-diry;   play_sound_resource("SH");}



 return 2;
}




int initfunction()
{

 create_graphics_region(4,width,height);
 select_graphics_object(4);

 dib_paint(0,0,imghnd,0,0);
 copy_graphics_region(grhnd,0,0,760,560,4,0,0,width,height,SRC);
 delete_graphics_region(4);

 create_graphics_region(brush,50,50);
 create_graphics_region(back,50,50);

 create_graphics_region(mask,50,50);
 select_graphics_object(mask);
 fill_rectangle(0,0,50,50,0);
 fill_ellipse(25,25,23,23,RGB(255,255,255));
 create_graphics_region(ball,50,50);
 select_graphics_object(ball);
 fill_rectangle(0,0,50,50,RGB(255,255,255));

 fill_ellipse(25,25,24,24,RGB(0,0,200));
 fill_ellipse(23,23,20,20,RGB(0,100,230));
 fill_ellipse(20,20,15,15,RGB(0,150,220));
 fill_ellipse(16,16,10,10,RGB(0,200,200));
 fill_ellipse(14,14,5,5,RGB(0,250,150));
 ellipse(25,25,24,24,RGB(255,255,0));
```

```
 select_graphics_object(grhnd);


 return 2;
}







main (int argc, char *argv[])
 {

  if (argc<=1)
    {
      printf("You must specify a bmp/pcx file to use!\n");
      exit(0);
    }

  get_im_info( argv[1], &width,&height,&clrs,numimage,fmt,err);

  for(i=0;i<3;i++) fmt[i]=toupper(fmt[i]);

  if ( strcmp(fmt,"BMP")==0) imghnd=import_bmp(argv[1], err);
  if ( strcmp(fmt,"PCX")==0) imghnd=import_pcx(argv[1], err);


  winio("%ca[Bouncing Bitmaps!]%ww[no_border]%bg[gray]&");
  winio("%`gr[rgb_colours]&",760,560,grhnd);
  winio("%sc&",initfunction);
  winio("%dl",0.001,redrawfunction);

 }


#pragma resource

SH SOUND "shield.wav"
```

BROWSE.CPP

Copy to Clipboard

```
#pragma windows 1000000 1000000
#include <windows.h>
#include <string.h>
#define LOGICAL

int c_set=1;
int f_set=0;
int executing=0;
int main_window;


int kh=0,kv=0;

char* textures[]={"Sticky","Messy","Dirty","Greasy","Slimy","Very
slippery",NULL};
int action()
{
       winio("No action today\n\n%cn%bt[Thank you]");
       return 2;
}

void toolbar_example()
{
    static char* options[]={"Active","Passive","Interactive",NULL};
    int k1=0,k2=0,k3=0,kopt=1;
    int ans=winio("%ww[no_border,maximise]%2.1tb %`ls %tb&",
                  "one_1","one_2","one_d",&k1,
                  "two_1","two_2","two_d",&k2,
                  options,&kopt,
                  "query_1","query_2","query_1",&k3);
        ans=winio("%bxPress Alt-F4 to close this window",0.5);
    }

void layout_example1()
{
    winio("%6BT[OK]     %6BT[CANCEL]");
    }

void layout_example2()
{
    winio("This is a vital message\n\n%CN%6BT[OK]     %6BT[CANCEL]");
    }

void layout_example3()
{
    winio("%1SI!This is a vital message\n\n%CN%6BT[OK]     %6BT[CANCEL]");
```

```
   }

void layout_example4()
{
    winio("%OB%10.5CW%CB line 1\n line 2\n line3\f line 4",0);
    };

void listboxes_example()
{
    int k;
    char*
things[]={"Apples","Bananas","Cherries","Grapes","Oranges","Pears","Rasberies
",NULL};
    int ivec[7];
    ivec[0]=0;
    ivec[1]=1;
    ivec[2]=0;
    ivec[3]=0;
    ivec[4]=1;
    ivec[5]=1;
    ivec[6]=0;
    winio("%ca[Selecting things]%`3tl&",15.0,30.0,45.0);
    winio("%tc[red]Simple\tDrop-down\tScrolling\tMultiple
selection\n%tc[black]&");
    winio("%ls\t&",things,&k);
    winio("%`ls\t&",things,&k);
    winio("%10.3ls\t&",things,&k);
    winio("%ms&",things,ivec);
    winio("\f\nNote that the first three boxes are coupled together\n&");
    winio("because they share the result variable k");
    }

void parameter_block_example()
{
    char name[20]="Thick and Sticky";
    int k1=1,k2=2,k4=4,k5=5;
       int texture_type=4;
       double v=4.5,p=200,a=25.2;
    winio("%ww[casts_shaddow]Double click on an item to change it\n\n&");
    winio("%30.8pb[sorted]&");
    winio("%dp[test1]&",&k1);
    winio("%dp[test2]&",&k2);
    winio("%ep[Oil texture]&",textures,&texture_type);
    winio("%dp[Temperature (Deg C)]&",&k4);
    winio("%dp[Carbon monoxide (%)]&",&k5);
    winio("%fp[Curent (Amps)]&",&a);
    winio("%10.3fp[Voltage]&",&v);
    winio("%fp[Oil pressure (PSI)]&",&p);
```

```
    winio("%?tp[Oil name][What is the official name of this oil?]&",name,20);
    winio("%up[Special action]&",action);
       winio("");
    }

void cursor_example_1()
{
    winio("%`dc%ob%`cu%gr%cb\f\n%cn%`bt[OK]",IDC_IBEAM,IDC_WAIT,100,100);
    }

void button_example_1()
{
       int ans;
       ans=winio("This is a test message\n\n%CN%`BT[Ok]    %BT[Cancel]");
       winio("%CA[Return value from WINIO]Value = %wd\n\n%cn%`bt[Ok]",ans);
    }

void button_example_2()
{
       int ans;
       ans=winio("This is a test message\n\n%CN%`6BT[Ok]    %6BT[Cancel]");
       winio("%CA[Return value from WINIO]Value = %wd\n\n%cn%`bt[Ok]",ans);
       }

void menu_example_1()
{
    winio("%mn[&File[&New,&Open,&Save,|,E&xit],&Edit[&Copy,C&ut,&Paste]]",
            "TEXT","New",
            "TEXT","Open",
            "TEXT","Save",
            "EXIT",
            "TEXT","Copy",
            "TEXT","Cut",
            "TEXT","Paste");
    }

char* contents[]={
strdup("ACABook"),
strdup("BCAChapter 1"),
strdup("CCASection 1.1"),
strdup("CCASection 1.2"),
strdup("CCASection 1.3"),
strdup("CCASection 1.4"),
strdup("BCAChapter 2"),
strdup("CCASection 2.1"),
strdup("CCASection 2.2"),
strdup("CCASection 2.3"),
strdup("CCASection 2.4"),
```

```
strdup("BCAChapter 3"),
strdup("CCASection 3.1"),
strdup("CCASection 3.2"),
strdup("CCASection 3.3"),
strdup("CCASection 3.4"),
strdup("BCAChapter 4"),
strdup("CCASection 4.1"),
strdup("CCASection 4.2"),
strdup("CCASection 4.3"),
strdup("CCASection 4.4"),
strdup("BCAChapter 5"),
strdup("CCASection 5.1"),
strdup("CCASection 5.2"),
strdup("CCASection 5.3"),
strdup("CCASection 5.4"),
strdup("BCAChapter 6"),
strdup("CCASection 6.1"),
strdup("CCASection 6.2"),
strdup("CCASection 6.3"),
strdup("CCASection 6.4"),
strdup("BCAChapter 7"),
strdup("CCASection 7.1"),
strdup("CCASection 7.2"),
strdup("CCASection 7.3"),
strdup("CCASection 7.4"),
strdup("BCAChapter 8"),
strdup("CCASection 8.1"),
strdup("CCASection 8.2"),
strdup("CCASection 8.3"),
strdup("CCASection 8.4"),
strdup("BCAChapter 9"),
strdup("CCASection 9.1"),
strdup("CCASection 9.2"),
strdup("CCASection 9.3"),
strdup("CCASection 9.4"),
strdup("BCAChapter 10"),
strdup("CCASection 10.1"),
strdup("CCASection 10.2"),
strdup("CCASection 10.3"),
strdup("CCASection 10.4"),
strdup("BCAChapter 11"),
strdup("CCASection 11.1"),
strdup("CCASection 11.2"),
strdup("CCASection 11.3"),
strdup("CCASection 11.4"),
NULL};
int item=6;
```

```
int tv_fun()
{
/*********************************************************************/
/*                                                                   */
/*    Call-back function sets the icon for each object according   */
/*    to whether it is expanded or not                               */
/*                                                                   */
/*********************************************************************/
       char* str=contents[item-1];
       if(*(str+1)=='E')
       *(str+2)='B';
       else
       *(str+2)='A';
       window_update(contents);
    return 2;
    }

void treeview_example()
{   winio("%mi[BROWSE]&");
    winio("%ww%ob%pv%^`20.15tv%cb\f\n%cn%`bt[OK]",
            contents,&item,"closed_book,open_book",tv_fun);
    }

int fortran_mode=1;
int f90_extensions=0;
int c_mode=0;

int me2_use_fortran()
{
    fortran_mode=1;
    c_mode=0;
    return 1;
    }

int me2_use_c()
{
    fortran_mode=0;
    f90_extensions=0;
    c_mode=1;
    return 1;
    }

int me2_use_f90_extensions()
{
    f90_extensions=(1-f90_extensions);
    return 1;
    }
```

```
void menu_example_2()
{
    winio("%mn[Language[#Fortran,#C,~#F90-extensions]]",
            &fortran_mode,me2_use_fortran,
            &c_mode,me2_use_c,
            &fortran_mode,&f90_extensions,me2_use_f90_extensions);
    }

void ps_example()
{
    int k1,k2,k3;
    int male=1,female=0;
    int alive=1,conscious=1,comfortable=0;
    char report[200]="Mr. Bloggs has a large wart\r\non his left buttock
which\r\nshould be removed with caustic soda\r\n";
    int age=75;
    char name[20];
    strcpy(name,"Fred Bloggs");
//
//     Sheet 1
//
    winio("%ca[Personal]Personal information:\n\n&");
    winio("%obName:\t%rs\nAge:\t%rd\nSex: %rb[Male]   %rb[Female]%2ga%cb%sh",
            name,20,&age,&male,&female,&male,&female,&k1);
//
//     Sheet 2
//
    winio("%ca[Medical]General patient condition\n\n&");
    winio("%ob%rb[Alive]\n%rb[Conscious]\n%rb[Comfortable]%cb%sh",
            &alive,&conscious,&comfortable,&k2);
//
//     Sheet 3
//
    winio("%ca[Report]Doctor's report\n\n&");
    winio("%30.4eb[hscrollbar,vscrollbar]%sh",report,200,&k3);
//
//     The main window
//
    winio("%ca[Patient status]%bg[grey]%3ps",&k1,&k2,&k3);
    }
struct example_table{
       char* name;
       void(*function)();
       };

example_table table[]={
{"menu_example_1",            menu_example_1},
{"menu_example_2",            menu_example_2},
```

```
{"cursor_example_1",          cursor_example_1},
{"parameter_block_example",   parameter_block_example},
{"listboxes_example",         listboxes_example},
{"layout_example1",           layout_example1},
{"layout_example2",           layout_example2},
{"layout_example3",           layout_example3},
{"layout_example4",           layout_example4},
{"button_example_1",          button_example_1},
{"button_example_2",          button_example_2},
{"ps_example",                ps_example},
{"treeview_example",          treeview_example},
{"toolbar_example",           toolbar_example},
{NULL,NULL}
       };

int hypertext_fn()
{
/*************************************************************************/
/*                                                                       */
/*    Call back for hypertext window. This runs the appropriate example  */
/*    program and returns to the window.                                 */
/*                                                                       */
/*************************************************************************/
    char option[50];
    if(executing)
       MessageBeep(0);
       else
       {
       executing=1;
    strcpy(option,clearwin_string("CURRENT_TEXT_ITEM"));
    strlwr(option);
    int k=0;
    while(table[k].name)
        {
        if(strcmp(table[k].name,option)==0)
            {
            (table[k].function)();
       executing=0;
            return 2;
            }
        k++;
        }
    winio("%`sp%bg[grey]%ww[casts_shadow,volatile,no_edge,no_caption]&",0,0);
    winio("Item %ws",option);
       }
       executing=0;
    return 2;
    }
```

```
int set_c()
{
    set_clearwin_info("FORTRAN",0);
    c_set=1;
    f_set=0;
    return 1;
    }

int set_f()
{
    set_clearwin_info("FORTRAN",1);
    f_set=1;
    c_set=0;
    return 1;
    }

main()
{
    add_hypertext_resource("TEXT");
    set_c();
    winio("%mi[BROWSE]&");

    winio("%ca[ClearWin Browser]
%mn[&File[E&xit],&Back,&History,&Language[#&C,#&Fortran]]&","EXIT","PREVIOUS_
TEXT","TEXT_HISTORY",&c_set,set_c,&f_set,set_f);
       if(windows_95_functionality())winio("%`sf%ts&",1.15);
       winio("%hw&",(int*)&main_window);
    winio("%bg[grey]%ww%pv%^70.20ht@","INTRODUCTION",hypertext_fn);
    }
#pragma resource
TEXT HYPERTEXT browse.htm
run BITMAP run.bmp
button BITMAP button.bmp
hptext01 BITMAP hptext01.bmp
browse BITMAP browse.bmp
one_1 BITMAP one_1.bmp
one_2 BITMAP one_2.bmp
one_d BITMAP one_d.bmp
two_1 BITMAP two_1.bmp
two_2 BITMAP two_2.bmp
two_d BITMAP two_d.bmp
query_1 BITMAP query_1.bmp
query_2 BITMAP query_2.bmp
BROWSE ICON browse.ico
closed_book ICON book1.ico
open_book ICON book2.ico
```

CDPLAY.CPP

Copy to Clipboard

```
//
// (c) Salford Software 1996
//
// Clearwin+ CD player example
//

#pragma windows 500000,500000

#include <stdio.h>
#include <windows.h>
#include <dbos\graphics.h>

#define CDPLAY 1
#define CDSTOP 0

unsigned int tick=0, remain=0, cdstate=CDSTOP, total=0;

int track=1;
unsigned int track_length[40],last_track=0;


char num[24];


int CB_tick();
int CB_init_display();
int CB_play();
int CB_stop();
int CB_next();
int CB_prev();
int CB_pause();
int CB_eject();



inline void update_display()
{
 if (cdstate==CDPLAY)
 {
  remain=(total/1000)-tick;
  size_in_pixels(12,6);
  fill_rectangle(5,24,290,36,0);
  sprintf(num,"[%u]",last_track);
  draw_text(num,5,35,10);
  sprintf(num,"<%u>",tick);
  draw_text(num,90,35,10);
```

```
  sprintf(num,"<%u>",remain);
  draw_text(num,200,35,10);
 }
}


//
// MAIN
//
main()
{
  winio("%mi[CDICO]&");
  winio("%ww[no_border,no_frame,no_maxbox]&");
  winio("%mn[CD[Play,Pause,Stop,|,Next,Prev,|,Eject],Exit]&",
     CB_play,   CB_pause,  CB_stop,
     CB_next,   CB_prev,  CB_eject,"exit"  );
  winio("%ca[CD Player]%`gr[black]&",300,50,1);
  winio("%sc&",CB_init_display);
  winio("%dl",1.0,CB_tick);
}


//
// Provide a way of measuring the distance into the track
//
int CB_tick()
{
 tick++;
 update_display();
 return 2;
}


//
// First time screen setup
//
int CB_init_display()
{
 stop_audio_cd();

 rectangle(0,0,297,47,2);
 rectangle(1,1,296,46,10);
 rectangle(2,2,295,45,2);
 select_font("ARIAL");
 size_in_pixels(8,10);
 draw_text("Track",5,20,10);
 draw_text("Time",90,20,10);
 draw_text("Remain",200,20,10);
```

```
 update_display();
 return 2;
}


//
// MENU play cd
//
int CB_play()
{
 int l=1,cnt;

  while( (track_length[l]=get_track_length(l)) >0) l++;
  last_track=l-1;


  // convert time to seconds look up table
  total=0;

  for(cnt=1;cnt<l;cnt++)  total+= track_length[cnt];

  track=1;
  set_cd_position(1,1);
  tick=0;
  play_audio_cd(total);
  cdstate=CDPLAY;
  update_display();

 return 2;
}


//
// MENU stop cd playing
//
int CB_stop()
{
 stop_audio_cd();
 set_cd_position(1,1);
 track=1;
 tick=0;
 update_display();
 cdstate=CDSTOP;
 return 2;
}

//
// MENU play next track
```

```
//
int CB_next()
{
 int jump,l;

 track++;
 if (track>last_track) track=last_track;

 jump=0;
 for(l=track;l<=last_track;l++) jump+=track_length[l];

 stop_audio_cd();
 set_cd_position(track,1);
 play_audio_cd(jump);
 update_display();
 return 2;
}

//
// MENU previous track
//
int CB_prev()
{
 int jump,l;

 track--;
 if (track<1) track=1;

 jump=0;
 for(l=track;l<=last_track;l++) jump+=track_length[l];


 stop_audio_cd();
 set_cd_position(track,1);
 play_audio_cd(total);

 update_display();
 return 2;
}

//
// MENU Pause playing current track
//
int CB_pause()
{
 return 2;
}
```

```
//
// MENU eject the cd
//
int CB_eject()
{
 stop_audio_cd();
 track=1;
 open_cd_tray();
 update_display();
 cdstate=CDSTOP;
 return 2;
}


#pragma resource

CDICO ICON "..\\resource\\cdplay.ico"
```

CI.CPP

Copy to Clipboard

```
#pragma windows 500000 500000

#include <windows.h>
#include <stdio.h>
#include <dbos\graphics.h>

int test()
{
   int x,y,flags;

   x     = clearwin_info("graphics_mouse_x");
   y     = clearwin_info("graphics_mouse_y");
   flags = clearwin_info("graphics_mouse_flags");

   printf(" X=%d, Y=%d, flags=%d\n",x,y,flags);

   fill_ellipse(x,y,6,6,RED);

   return 1;
}
main()
{
   winio("%ob[named_l][graphics area]%`^gr%cb&",300,300,1,test);
   winio("%ob[nameless,bottom_exit]%50.5cw%cb&",0);
   winio("%nl%nl %`bt[ok]%ff Click mouse in the graphics region.");
}
```

CU.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>

main()
{
// n.b. : one can not use IDC_SIZENS

   winio("%`cu&",CURSOR_SIZENS);
   winio("%`gr[black]&",100,100,1);


   winio("%cu[my_cursor]&");
   winio("%`gr[red]&",100,100,2);

   winio(" %`^bt[OK]","EXIT");
}

#pragma resource
#include "cursor.rc"
```

DRAGDROP.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <string.h>

char drop_file[129]="no file";

int drop()
{
  strcpy(drop_file, clearwin_string("DROPPED_FILE"));
  window_update(drop_file);
  return 2;
}

int main()
{
    winio("%ca[Drag and drop]\nThe string should contain%dr&",drop);
    winio(" the path of the dropped file:\n\n[%30`rs]\n",drop_file,65);
}
```

EBOX.CPP

Copy to Clipboard

```
//----  (c) Salford Software Ltd 1996
//----  edit box as mdi child demo

#pragma windows 500000, 500000
#include <windows.h>


//----  Must have this window handle/control
int frctrl;
char buff[32767];
EDIT_INFO edit_ctrl_struct;

/* --------- typedefed in clearwin.h (included in windows.h) ----
    struct EDIT_INFO  {
      int  h_position;
      int  v_position;
      int  last_line;
      char *buffer;
      int  buffer_size;
      int  max_buffer_size;
      char *current_position;
      char *selection;
      int  n_selected;
      int  vk_key;
      int  vk_shift;
      int  active;
      int  modified;
      int  closing;
      char reserved[40];
     };
-------------------------------------------------  */


/**************/
/*  MAIN      */
/**************/
int main()
{
   frctrl=1;

//----  Init the structure
   edit_ctrl_struct.h_position=1;
   edit_ctrl_struct.v_position=1;

//----  MDI Parent
   winio("%ca[Edit Box test]%lw&",&frctrl);
```

```
   winio("%ww[no_border,maximize]%pv%fr",800,400);



//----  MDI Child ------ note above no '&' join character --------
   winio("%ca[The Child Edit Window]&");
   winio("%pv%aw%ww[no_border]
%`60.15eb[hscrollbar,vscrollbar,fixed_font,no_border,use_tabs]",
      &frctrl, buff ,32767,&edit_ctrl_struct);
  return 0;
}
```

FACTOR1.CPP

Copy to Clipboard

```
// factor1.cpp
#pragma windows 300000,500000,"cwp_ico.rc"
#include <windows.h>

int main()
{
   winio("%ca[Number Factoriser]");
   return 0;
}


_
```

FACTOR2.CPP

```
// factor2.cpp
#pragma windows 300000,500000,"cwp_ico.rc"
#include <windows.h>

int number=1;

int main()
{
   winio("%ca[Number Factoriser]&");
   winio("%il&",1,2147483647);
   winio("Number to be factorised: %rd",&number);
   return 0;
}

_
```

FACTOR3.CPP

```
// factor3.cpp
#pragma windows 300000,500000,"cwp_ico.rc"
#include <windows.h>

int factoriser();
int number=1;

int main()
{
   winio("%ca[Number Factoriser]&");
   winio("%il&",1,2147483647);
   winio("Number to be factorised: %rd&",&number);
   winio("\t%`^bt[Fac&torise]",factoriser);
   return 0;
}

int factoriser()
{
   return 1;
}

_
```

FACTOR4.CPP

```
// factor4.cpp
#pragma windows 300000,500000,"cwp_ico.rc"
#include <windows.h>
#include <stdio.h>

int factoriser();
int number=1;
char str[50]="";

int main()
{
   winio("%ca[Number Factoriser]&");
   winio("%il&",1,2147483647);
   winio("Number to be factorised: %rd&",&number);
   winio("\t%`^bt[Fac&torise]&",factoriser);
   winio("\n\n%ob%42st%cb",str);
   return 0;
}

int factoriser()
{
   sprintf(str,"%d",number);
   window_update(str);
   return 1;
}
```

FACTOR5.CPP

```
// factor5.cpp
#pragma windows 300000,500000,"cwp_ico.rc"
#include <windows.h>
#include <stdio.h>

int factoriser();
int about();
int number=1;
char str[50]="";

int main()
{
   winio("%ca[Number Factoriser]&");
   winio("%mn[&File[E&xit]]&","EXIT");
   winio("%mn[&Help[&About Number Factoriser]]&",about);
   winio("%il&",1,2147483647);
   winio("Number to be factorised: %rd&",&number);
   winio("\t%`^bt[Fac&torise]&",factoriser);
   winio("\n\n%ob%42st%cb",str);
   return 0;
}

int factoriser()
{
   sprintf(str,"%d",number);
   window_update(str);
   return 1;
}

int about()
{
   winio("%ca[About Number Factoriser]&");
   winio("%fn[Times New Roman]%ts%bf%cnTutorial&",2.0);
   winio("%ts\n\n\n\n&",1.0);
   winio("%cnProgram written to demonstrate\n\n&");
   winio("%ts%tc%cn%bfClearWin+&",1.5,RGB(255,0,0));
   winio("%tc%sf\n\n%cnby&",-1);
   winio("\n\n%cnSalford Software&");
   winio("\n\n%cn%9`bt[OK]");
   return 1;
}

_
```

FACTOR6.CPP

(File missing)

FS.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>

int disp_name();

char filenm[100]="";

main()
{
   winio("%fs@%mn[&File[&Open,E&xit]]",
         "c:\\windows\\*.wri","FILE_OPENR",
         filenm,100,disp_name,"EXIT");

}


int disp_name()
{
  winio("%ws %`bt[ok]",filenm);

  return 1;
}
```

FT.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>

int disp_name();

char filenm[100]="";

main()
{
   winio("%ft@@&","Write Files","*.wri");

   winio("%mn[&File[&Open,E&xit]]",
         "FILE_OPENR",filenm,100,disp_name,
         "EXIT");
}


int disp_name()
{
  winio("%ws %`bt[ok]",filenm);

  return 2;
}
```

GRAPH.CPP

Copy to Clipboard

```
#pragma windows 2000000 3000000

#include <windows.h>
#include <\dbos\graphics.h>
#include <math.h>

int x_res=700;
int y_res=400;

extern "C" double RANDOM();
extern void load_standard_colours();

COLORREF black=RGB(0,0,0);
COLORREF red=RGB(255,0,0);
#define n_points 1000
double Y[n_points];

int paint()
{
    if(clearwin_info("GRAPHICS_RESIZING"))
        {
        double x_margin,y_margin,total_x,total_y,x0,y0;
        x_res=clearwin_info("GRAPHICS_WIDTH");
        y_res=clearwin_info("GRAPHICS_DEPTH");
        x_margin=x_res*0.05;
        y_margin=y_res*0.05;
        total_x=x_res-2*x_margin;
        total_y=(y_res-2*y_margin)/2;
        x0=x_margin;
        y0=y_res/2;
        draw_line(x0,y0,x0+total_x,y0,black);
        draw_line(x0,y0+total_y,x0,y0-total_y,black);
        int k=0;
        double max_y=0;
        while(k < n_points)
            {
             if(fabs(Y[k]) > max_y)max_y=fabs(Y[k]);
             k++;
            }
        if(max_y==0)max_y=1;
        k=0;
        MessageBeep(0);
        while(k < n_points)
            {
             double x=x0+total_x*k/n_points;
             double y=y0-total_y*Y[k]/max_y;
```

```
            draw_line(x,y,x,y0,RGB(k%255,k%(~255),0));
            k++;
           }
        }
    return 1;
    }




main()
{
    int flag;
    int k;
    double p=0.0;


    for(k=0;k<n_points;k++)
    {
     Y[k]=(sin(p)*10)+(sin(p/2)*5);
      p+=0.0273;
    }
    winio("%ca[User resize window]&");
    winio("%mn[&File[E&xit]]%ww[no_border]
%pv%^gr[rgb_colours,white,user_resize]",
           "EXIT",x_res,y_res,paint);
 }
```

IMG1.CPP

```
// (c) Salford Software Ltd 1995
// simple winio example
#pragma  windows 5000000,500000
#include <windows.h>

int main()
{
winio("%ca[Number Factoriser]"); return 0;

}
```

IMG2.CPP

```
// (c) Salford Software Ltd 1995
// Simple %rd example

#pragma  windows 5000000,500000
#include <windows.h>

int number=1;

int main()
{
winio("%ca[Number Factoriser]%il&",1,2147483647);
winio("Number to be factorised: %rd",&number);
 return 0;

}
```

IMG3.CPP

```
// (c) Salford Software Ltd 1995


#pragma  windows 5000000,500000
#include <windows.h>

int number=1;
int factoriser() {return 1;}

int main()
{
winio("%ca[Number Factoriser]%il&",1,2147483647);
winio("Number to be factorised: %rd&",&number);
winio("\t%`^bt[Fac&torise]",factoriser);
 return 0;

}
```

IMG4.CPP

```
// (c) Salford Software Ltd 1995

#pragma  windows 5000000,500000
#include <windows.h>

int number=1;

char str[50]="";


int factoriser() {

sprintf(str,"%d",number);
window_update(str);
return 1;
}

int main()
{
winio("%ca[Number Factoriser]%il&",1,2147483647);
winio("Number to be factorised: %rd&",&number);
winio("\t%`^bt[Fac&torise]&",factoriser);
winio("\n\n%ob%42st%cb",str);

 return 0;

}
```

IMG5.CPP

```
// (c) Salford Software Ltd 1995
// NUmber factoriser example

#pragma  windows 5000000,500000
#include <windows.h>

int number=1;

char str[50]="";

int about()
{

  winio("%ca[About Number Factoriser]&");
  winio("%fn[Times New Roman]%ts%bf%cnTutorial&",2.0);
  winio("%ts\n\n\n&",1.0);
  winio("%cnProgram written to demonstrate\n\n&");
  winio("%ts%tc%cn%bfClearWin+&",1.5,RGB(255,0,0));
  winio("%tc%sf\n\n%cnby&",-1);
  winio("\n\n%cnSalford Software&");
  winio("\n\n%cn%9`bt[OK]");



 return 1;
}


int factoriser() {

sprintf(str,"%d",number);
window_update(str);
return 1;
}

int main()
{
winio("%ca[Number Factoriser]%il&",1,2147483647);
winio("Number to be factorised: %rd&",&number);
winio("\t%`^bt[Fac&torise]&",factoriser);
winio("\n\n%ob%42st%cb&",str);

winio("%pv%mn[&File[E&xit]]&","EXIT");
winio("%mn[&Help[&About Number Factoriser]]",about);

 return 0;
```

```
}
```

IMG6.CPP

```
// (c) Salford Software Ltd 1995

#pragma  windows 5000000,500000
#include <windows.h>
#include <string.h>


int number=1;

char str[50]="";

int about()
{

  winio("%ca[About Number Factoriser]&");
  winio("%fn[Times New Roman]%ts%bf%cnTutorial&",2.0);
  winio("%ts\n\n\n&",1.0);
  winio("%cnProgram written to demonstrate\n\n&");
  winio("%ts%tc%cn%bfClearWin+&",1.5,RGB(255,0,0));
  winio("%tc%sf\n\n%cnby&",-1);
  winio("\n\n%cnSalford Software&");
  winio("\n\n%cn%9`bt[OK]");



 return 1;
}


int factoriser() {
int k,n=number;

char val[10];



sprintf(str,"The factors %d are: 1",number);

 while(n>1)
 {
 for(k=2;k<=n;++k)
  if (n%k==0)
  {
   sprintf(val,",%d",k);
   strcat(str,val);
   if ((n/=k)>1) break;
```

```
  }
 }


window_update(str);
return 1;
}

int main()
{
winio("%ca[Number Factoriser]%il&",1,2147483647);
winio("Number to be factorised: %rd&",&number);
winio("\t%`^bt[Fac&torise]&",factoriser);
winio("\n\n%ob%42st%cb&",str);

winio("%pv%mn[&File[E&xit]]&","EXIT");
winio("%mn[&Help[&About Number Factoriser]]",about);

 return 0;

}
```

IMG7.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000,500000
#include <windows.h>

int main()
{

 winio("Idiot!\t%bt@","&Sorry");
 return 0;
}
```

IMG8.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000,500000
#include <windows.h>


int func()
{
  return 2;
}

    main()
    {
      winio("Press this to see what happens %^bt[PRESS]", func);
      return 0;
    }
```

IMG9.CPP

```
// (c) Salford Software 1995

#pragma windows 500000,500000
#include <windows.h>

int grey_control;

int open_func()    { grey_control=1;return 2;}
int save_func()    { grey_control=0;return 2;}
int save_as_func() { grey_control=0;return 2;}

int main()
{
        //Only the OPEN button is initially available
        grey_control=0;
        winio("%^bt[Open]  %~^bt[Save]  %~^bt[Save as]",
                open_func, &grey_control, save_func,
                        &grey_control, save_as_func);
}
```

IMG10.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000,500000
#include <windows.h>
int main()
{
 double bar_value;
 int window_ctrl=1;

 winio("%ca[Bar format]Processing .....\n\n%20br%lw",
 &bar_value,RGB(255,255,0),&window_ctrl);

  for(bar_value=0;bar_value<=1;bar_value+=0.003)
    {
      for(int i=0;i<100000;++i);  //do something here
      window_update(&bar_value);
    }
}
```

IMG11.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000,500000
#include <windows.h>
int main()
{

 char* greek[]={"alpha","beta","gamma",NULL};
 int selection=1;
 winio("Select a Greek letter:  %lb",greek,&selection);

}
```

IMG12.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000 500000
#include <windows.h>


int active()
{
 winio(" active icon ");
 return 2;
}


int main()
{
  winio("This is an icon: %ic[clear_win]");


 winio("%2si?%2si*%^2si!%2si#\n\nThese are all the standard icons.
\n\n&",active);
 winio("%cn%7bt[OK]");
 MessageBeep(MB_ICONEXCLAMATION);
 MessageBeep(MB_ICONASTERISK);
 MessageBeep(MB_ICONHAND);
 MessageBeep(MB_ICONQUESTION);

}



 #pragma resource
  clear_win ICON sheep.ico
```

IMG13.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000 500000
#include <windows.h>


int main()
{
  winio("What shall I do now?\n%bt[Continue] %bt[Giveup]\n");

}
```

IMG14.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000,500000
#include <windows.h>
int x,y;
int myfunc()
{
//      Window will be positioned relative to the button
//      control in the main window
        winio("%spHidden!",x-5,y-5);
                return 1;
}


int main()
{
  winio("Press this button to conceal it! %gp%^bt[Press]",&x,&y,myfunc);
}
```

IMG15.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000 500000
#include <windows.h>


int main()
{
 HDC my_dc=get_bitmap_dc(50,50);
 MoveTo(my_dc,0,0);
 LineTo(my_dc,50,50);
 winio("%dw %bt[OK]",my_dc);
 // The next line is not necessary unless
 // many device contexts will be acquired
 release_bitmap_dc(my_dc);

 winio("%ca[Error]");

}
```

IMG16.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000 500000
#include <windows.h>

int main()
{
  int n_ch;
  char* help="How many children have you got?";
  window_printf("No of children %?rd@",&n_ch,help);
}
```

IMG17.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000,500000
#include <windows.h>
int main()
{
  double x=1.0,y=2.5;
    winio("X = \t%wf\nY = \t%wf",x,y);
}
```

IMG18.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000 500000
#include <windows.h>
int main()
{
int p;
winio("%si*Enter an integer:%3rd",&p);
}
```

IMG19.CPP

```
// (c) Salford Software Ltd 1995

#pragma windows 500000 500000
#include <windows.h>
#include <math.h>

double X,Y,R,THETA;
int convert_to_polar()
{
  R=sqrt(X*X+Y*Y);
  THETA=(X==0.0 && Y==0.0)? 0.0:atan2(Y,X);

  return 2;
}

int convert_to_XY()
{
   X=R*cos(THETA);
   Y=R*sin(THETA);
return 2;
}

main()
{
  winio("X = %^rf   Y= %^rf    R = %^rf   THETA = %^rf",
        &X,convert_to_polar,&Y,convert_to_polar,
        &R,convert_to_XY,&THETA,convert_to_XY);
}
```

LCTEST.CPP

Copy to Clipboard

```
// (c) Salford Software 1996
//
// Program to show how usefull a %lc can be
//

#pragma windows 500000, 500000
#include <windows.h>


int  hw_a,hw_b;
int  ax1,ay1,ax2,ay2,bx1,by1,bx2,by2;
int ss=50;


int current,page=2,maxpage=40;


      int lson()
      {
       EnableWindow( hw_a, 0);
       EnableWindow( hw_b, 0);
       return 2;
      }

      int lsoff()
      {
       EnableWindow( hw_a, 1);
       EnableWindow( hw_b, 1);
       return 2;
      }

      int lsvis()
      {
       ShowWindow( hw_a, 0);
       ShowWindow( hw_b, 5);
      return 2;
      }

      int lshid()
      {
       ShowWindow( hw_a, 5);
       ShowWindow( hw_b, 0);
       return 2;
      }

      int cb_button()
```

```
    {
     winio("%ww[no_border]PRESSED");
     return 2;
    }


    int cb_scrollbar()
    {
     move_window(hw_a,ax1,(ay1-ay2)+current);
     move_window(hw_b,bx1,(by1-by2)+current);
     return 2;
    }

    int cb_grow()
    {
     ss+=10;
     resize_window(hw_a,ss+ss,ss);
     resize_window(hw_b,ss+ss,ss);
     if (ss>140) ss=40;
     return 2;
    }




main()
{

    int setup,lsvar;
    setup=1;
    char *string[]={
                 "This is test line 1",
                 "This is test line 2",
                 "This is test line 3",
                 "This is test line 4",
                 "This is test line 5",
                 "This is test line 6",NULL };

    winio("%ca[Testing %lc control]%bg[grey]&");
    winio("%^vs&",page,maxpage,&current,cb_scrollbar);
    winio("%nl %^10bt[grey on]   %^10bt[grey off]%ff&",lson,lsoff);
    winio("%nl %^10bt[LS BOX]   %^10bt[BUTTON]%ff&",lshid,lsvis);
    winio("%nl %^10bt[BIGGER]&",cb_grow);
    winio("%ap%20.4ls%lc&",6,10,string,&lsvar, (int *)&hw_a);
    winio("%ap%^14bt[BIG BUTTON]%lc&",6,10,cb_button,(int *)&hw_b);
    winio("%lw", &setup);

    get_window_location(hw_a,ax1,ay1,ax2,ay2);
```

```
    get_window_location(hw_b,bx1,by1,bx2,by2);

    ShowWindow( hw_a, 5);
    ShowWindow( hw_b, 0);


}
```

LS.CPP

Copy to Clipboard

```
#pragma windows 500000 500000
#include <windows.h>

int k;
char* things[]={"Apples","Bananas","Cherries",
               "Grapes","Oranges","Pears","Raspberries",NULL};
int ivec[7]={0,0,0,0,0,0,0};
main()
{
    ivec[1]=1;
    ivec[4]=1;
    ivec[5]=1;
    winio("%ca[Selecting things]%3tl&",15,30,45);
    winio("%tc[red]Simple\tDrop-down\t"
           "Scrolling\tMultiple selection\n%tc[black]&");
    winio("%lb\t&",things,&k);
    winio("%`ls\t&",things,&k);
    winio("%10.3ls\t&",things,&k);
    winio("%ms&",things,ivec);
    winio("\f\nNote that the first three boxes are coupled together\n&");
    winio("because they share the result variable k");
    }
```

MAKEICON.CPP

Copy to Clipboard

```
#pragma windows  500000,500000
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

char icondata[]= {

'R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','
R','R','R','R','R','R','B','R','R','R','R','R','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',
```

```
'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','
B','B','Y','Y','B','B','Y','Y','B','B','Y','Y','R',

'R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','R','
R','R','R','R','R','R','B','R','R','R','R','R','R'};


int main()
```

```
{
  int hicon=make_icon( icondata );
  winio("%ca[Make icon]\n%`ic\n",hicon);

 return 0;
}
```

META.CPP

Copy to Clipboard

```
#pragma windows 1000000 1000000

#include <windows.h>
#include <dbos\graphics.h>
#include <dbos\devices.h>
#define INT_WHITE 15


#define NB_POL 10
#define FONT_COLOR RED

#define mprintf __mprintf
#pragma silent
extern "C" void mprintf(char*,...);


short background_colour;

void test_graphics_fonts();
int startup_callback_func();


//-----------------------------------------

int meta_to_clip_func()
{
   int res=metafile_to_clipboard();

   return 1;
}

//-----------------------------------------

int meta_from_clip_func()
{
   static int window_control;

   winio("%`gr[BLACK] %ca[2]%ww%lw",
                  640,480,2,&window_control);


   int res=play_clipboard_metafile();

   return 1;
}
```

```
//-----------------------------------------

int startup_callback_func()
{
   test_graphics_fonts();

   return 1;
}
//-----------------------------------------

int clear_screen_func()
{
   clear_screen();

   return 1;
}

//-----------------------------------------

main()
{
   int width=640;
   int depth=480;

   winio("%mn[&File[&New,&Draw,&Exit]]&",
         clear_screen_func,startup_callback_func,"EXIT");
   winio("%mn[&Edit[&Metafile to Clipboard,&From Clipboard]]&",
         meta_to_clip_func,meta_from_clip_func);
   winio("%pv%`gr[BLACK,METAFILE_RESIZE] %ca[Salford graphics]%ww%sc",
                  width,depth,1,startup_callback_func);
}

//-----------------------------------
//-----------------------------------

void test_graphics_fonts()
{

   short x[10],y[10],error;
   short i,j,n=5,nbp;
   short color;

   // h:array of handles
   short h[NB_POL+8];

   char buf[] = "HELLO Girl !";
   short h_end,v_end;
```

```
   draw_line(0,0,200,200,RED);
   draw_line(0,200,200,0,GREEN);


   ellipse(100,100,80,40,RED);
   fill_ellipse(100,100,60,40,YELLOW);



   x[0]=230;
   y[0]=20;
   x[1]=280;
   y[1]=20;
   x[2]=280;
   y[2]=120;
   x[3]=230;
   y[3]=120;

   x[4]=x[0];
   y[4]=y[0];


   polyline(x,y,5,RED);



   x[0]=30;
   y[0]=250;
   x[1]=80;
   y[1]=250;
   x[2]=80;
   y[2]=350;
   x[3]=30;
   y[3]=350;

   x[4]=30;
   y[4]=250;


   for(j=0;j<NB_POL;j++)
   {
      for(i=0;i<5;i++) {x[i] += 10;y[i] -= 5;}
//      create_polygon(x,y,4,h[j],error);
      create_polygon(x,y,5,h[j],error);
   }
```

```
   for(j=0;j<NB_POL;j++) fill_polygon(h[j],GREEN,error);


//winio("%si!%sp Press key : MOVE  %ww%`bt[ok]",200,20);

   // move h[8]
   fill_polygon(h[8],background_colour,error);
   move_polygon(h[8],100,100,error);
   fill_polygon(h[8],RED,error);

//winio("%si!%sp Press key : UNDISPLAY  %ww%`bt[ok]",200,20);

   // undisplay h[1]
   fill_polygon(h[1],background_colour,error);


   // creating star

   x[0]=350;
   y[0]=100;
   x[1]=500;
   y[1]=250;
   x[2]=450;
   y[2]=0;
   x[3]=400;
   y[3]=250;
   x[4]=550;
   y[4]=100;

   x[5]=x[0];
   y[5]=y[0];

//   create_polygon(x,y,5,h[14],error);
   create_polygon(x,y,6,h[14],error);


//winio("%si!%sp Press key : DISPLAY STAR  %ww%`bt[ok]",200,20);

   // display star
   fill_polygon(h[14],YELLOW,error);


//winio("%si!%sp Press key : UNDISPLAY STAR %ww%`bt[ok]",200,20);

   // undisplay star
   fill_polygon(h[14],background_colour,error);
```

```
   x[0]=230;
   y[0]=150;
   x[1]=280;
   y[1]=150;
   x[2]=280;
   y[2]=250;
   x[3]=230;
   y[3]=250;

   x[4]=230;
   y[4]=150;



   for(j=10;j<14;j++)
   {
      for(i=0;i<5;i++) {x[i] += 10;y[i] -= 5;}
//      create_polygon(x,y,4,h[j],error);
      create_polygon(x,y,5,h[j],error);
   }


   nbp=2;
   // combine two rectangles
   combine_polygons(&h[10],nbp,h[15],error);

   // combine two rectangles
   combine_polygons(&h[12],nbp,h[16],error);


   nbp = 3;

   // combine the four rectangles and the star
   combine_polygons(&h[14],nbp,h[17],error);

//winio("%si!%sp Press key : DISPLAY COMBINED POLYGONS %ww%`bt[ok]",200,20);

   fill_polygon(h[17],YELLOW,error);

//winio("%si!%sp Press key : MOVE COMBINED POLYGONS %ww%`bt[ok]",200,20);

   fill_polygon(h[17],background_colour,error);
   move_polygon(h[17],100,50,error);
   fill_polygon(h[17],RED,error);


   // free memory
   for(i=0;i<17;i++) delete_polygon(h[i],error);
```

```
//-----

   for(i=0;i<640;i++) set_pixel(i,260,YELLOW);

   get_pixel(0,260,color);

//winio("%sp LINE COLOR : %wd %ww%`bt[ok]",200,20,color);

   rectangle(290,20,340,70,YELLOW);

   fill_rectangle(350,20,370,70,YELLOW);




//--------
// HERHSEY
//--------

//$$$$$$$$$    set_text_attribute(101,40,0,0);
//$$$$$$$$$    draw_hershey(2034,200,600,15,h_end,v_end);

//-------------
// WINDOWS FONT
//-------------

// 161
   set_text_attribute(155,2.0,90.0,0);
   draw_text(buf,300,350,FONT_COLOR);


//   set_text_attribute(154,2.0,0,1);
   set_text_attribute(1,1.0,0.0,0);
   draw_text(buf,380,400,FONT_COLOR);

   set_text_attribute(1,1.0,90.0,0);
   draw_text(buf,380,400,FONT_COLOR);
}
```

MICO.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <math.h>
#include <stdio.h>
#include <dir.h>
#include <string.h>
#include <dbos\graphics.h>

#define ICON_HOT_X 5

int width=300,height=200;
int x_arrow_r=50-ICON_HOT_X,y_arrow_r=100;

int x_org=50,x_end=250,y_org=100;


// draw a ruler
int start_call()
{
   int i;

   int id_arrow_r=add_graphics_icon("arrow_r",&x_arrow_r,&y_arrow_r,9,26);

   draw_line(50,100,251,100,RGB(0,128,0));

   for(i=50;i<250;i+=5)
   {
      draw_line(i,95,i,100,RGB(255,128,0));
   }

   for(i=50;i<260;i+=50)
   {
      draw_line(i,90,i,100,RGB(0,128,0));
   }



   return 2;
}

int gr_callback()
{
 // lock the arrow Y coord
    y_arrow_r=y_org;
 // allow limited movement on X coord
    if(x_arrow_r < x_org) x_arrow_r=x_org-ICON_HOT_X;
```

```
    else
    if(x_arrow_r > x_end) x_arrow_r=x_end-ICON_HOT_X;

   return 2;
}

void main()
{
    static int ctrl;


    winio("%mn[&Exit]&","EXIT");
    winio("%sc%ca[Moveable Icon]&",start_call);
    winio("%bg[grey]%ob[depressed]&");
    winio("%`cu%^`gr[grey,rgb_colours,full_mouse_input]
%cb%lw",IDC_ARROW,width,
                height,1,gr_callback,&ctrl);
}

#pragma resource
arrow_r   ICON      arrow_r.ico
```

MINMAX.CPP

Copy to Clipboard

```
// (c) Salford Software 1996
// get window state API information
//


#pragma windows 500000, 500000
#include <string.h>
#include <windows.h>


int hwnd;

char SZwinstate[20]="Window NORMAL",SZcaption[20]="Resize Detector";

int cb_checkstate()
{
 // the 'is' functions are windows API calls

 if ( IsZoomed(hwnd) )
 {
  strcpy(SZwinstate,"Window ZOOMED");
  strcpy(SZcaption,"Resize ZOOM");
  window_update(SZwinstate);
  window_update(SZcaption);
 }
 else
 if ( IsIconic(hwnd))
 {
  strcpy(SZcaption,"Win Minimized");
  window_update(SZcaption);
 }
 else
 {
  strcpy(SZwinstate,"Window NORMAL");
  strcpy(SZcaption,"Resize Normal");
  window_update(SZwinstate);
  window_update(SZcaption);
 }

 return 2;
}


main()
{
 winio("%ww[no_border]&");
```

```
 winio("%ca@&",SZcaption);
 winio("%cn%`rs&",SZwinstate,20);
 winio("%dl%hw",0.5,cb_checkstate,(int *)&hwnd);
}
```

MN.CPP

Copy to Clipboard

```
//
// (c) Salford Software 1995
// Dynamic menus %mn using the '*' format modifier.
//

#pragma windows 500000,500000
#include <windows.h>


char *string[]={"FISH ","CHIPS ","PEAS ","PIE "};

int handle,count=0;

int cbchange();
int cbdummy();


int main()
{

  winio("%ca[Dynamic menus]\n&");
  winio("%mn[&Window[*]]&",&handle);
  winio("\n\n %14^bt[change option]\n\n ",cbchange);

 return 0;

}



// called when menu item selected
int cbdummy()
{
 winio("PRESSED!");
 return 2;
}

// called on button pressed
int cbchange()
{
 remove_menu_item( handle );
 add_menu_item( handle, string[count], 0,0,cbdummy);
 count=(++count)%4;
 return 2;
}
```

OB.CPP

Copy to Clipboard

```
// displays various output boxes
//
// (c) Salford Software 1996

#pragma windows 500000,500000
#include <windows.h>

void main()
{
   winio("%mn[E&xit]&","EXIT");
   winio("%bg[grey]&");
   winio("%ob[named_l][named_l option]%cb&");

   winio("%ob[nameless,bottom_exit]A nameless bottom_exit box%cb&");

   winio("%ff%ob[no_border]%fn[courier new]A no_border box%`6bt[ok]%nl&");
   winio("%nl          %6bt[cancel]%cb%ff%nl&");

   winio("%ob[thin_panelled]A thin_panelled box%cb%ff&");
   winio("%ob[panelled,bottom_exit]A normal panelled box%cb%ff&");

   winio("%ob[depressed]A depressed box%cb%nl&");

   winio("%ta%ta%ob1%nl%nl%nl2%cb%ob[bottom_exit]1%cb%ob2%cb%ff&");

   winio("%ob[status] This is a status bar  %cb");
}
```

OUTPUT.CPP

Copy to Clipboard

```
/************************************************************************

    PROGRAM: Output.cpp

    PURPOSE: Output template for Windows applications

    FUNCTIONS:

     WinMain() - calls initialization function, processes message loop
     InitApplication() - initializes window data and registers window
     InitInstance() - saves instance handle and creates main window
     MainWndProc() - processes messages
     About() - processes messages for "About" dialog box

    DESCRIPTION: This program shows how ClearWin+ can be used to write
       programs based on the direct use of the Windows API.
     ExportProc is the only function used in this program that
     is drawn from the ClearWin+ library.  Otherwise this is a
     standard Windows SDK program.

     Although you can write programs in this way, most programmers
     will probably prefer to use Format (Winio) and ClearWin windows.


************************************************************************/
#pragma windows 10000 10000 "output.rc"
#include <windows.h>
#include <stdio.h>
#include "output.h"
#include <stdlib.h>
#include <string.h>

HANDLE hInst;

HPEN hDashPen;                                  /* "---" pen handle
*/
HPEN hDotPen;                                   /* "..." pen handle
*/
HBRUSH hOldBrush;                               /* old brush handle
*/
HBRUSH hRedBrush;                               /* red brush handle
*/
HBRUSH hGreenBrush;                             /* green brush handle
*/
HBRUSH hBlueBrush;                              /* blue brush handle
*/
```

```
/*************************************************************************

    FUNCTION: WinMain(HANDLE, HANDLE, LPSTR, win_int)

    PURPOSE: calls initialization function, processes message loop

*************************************************************************/

win_int WinMain(HANDLE hInstance,HANDLE  hPrevInstance,LPSTR
lpCmdLine,win_int nCmdShow)
{
    MSG msg;

    if (!hPrevInstance)
     if (!InitApplication(hInstance))
         return (FALSE);

    if (!InitInstance(hInstance, nCmdShow))
        return (FALSE);

    while (GetMessage(&msg, NULL, NULL, NULL)) {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
    }
    return (msg.wParam);
}


/*************************************************************************

    FUNCTION: InitApplication(HANDLE)

    PURPOSE: Initializes window data and registers window class

*************************************************************************/

BOOL InitApplication(HANDLE hInstance)
{
    WNDCLASS  wc;

    wc.style = NULL;
    wc.lpfnWndProc = ExportProc(&MainWndProc);
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
```

```
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  "OutputMenu";
    wc.lpszClassName = "OutputWClass";

    return (RegisterClass(&wc));
}


/****************************************************************************

    FUNCTION:  InitInstance(HANDLE, win_int)

    PURPOSE:  Saves instance handle and creates main window

****************************************************************************/

BOOL InitInstance(HANDLE hInstance,win_int nCmdShow)
{
    HWND            hWnd;

        hInst = hInstance;

    hWnd = CreateWindow(
        "OutputWClass",
            "32-BIT Application",
            WS_OVERLAPPEDWINDOW,
            0,
            0,
            GetSystemMetrics(SM_CXSCREEN),
            GetSystemMetrics(SM_CYSCREEN),
            NULL,
            NULL,
            hInstance,
            NULL
            );

    if (!hWnd)
        return (FALSE);

        ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    return (TRUE);

        }

/****************************************************************************

    FUNCTION: MainWndProc(HWND, unsigned, WORD, LONG)
```

```
    PURPOSE:  Processes messages

    MESSAGES:

     WM_COMMAND    - application menu (About dialog box)
     WM_CREATE     - create window and objects
     WM_PAINT      - update window, draw objects
     WM_DESTROY    - destroy window

    COMMENTS:

     Handles to the objects you will use are obtained when the WM_CREATE
     message is received, and deleted when the WM_DESTROY message is
     received.  The actual drawing is done whenever a WM_PAINT message is
     received.

*************************************************************************/

long  MainWndProc(HWND hWnd,unsigned message,WORD wParam,LONG lParam)
{
    FARPROC lpProcAbout;

    HDC hDC;                          /* display-context variable  */
    PAINTSTRUCT ps;                   /* paint structure           */
    RECT rcTextBox;                   /* rectangle around the text */
    HPEN hOldPen;                     /* old pen handle            */



    switch (message) {
     case WM_COMMAND:
         if (wParam == IDM_ABOUT) {
           lpProcAbout = MakeProcInstance(ExportProc(&About), hInst);

           DialogBox(hInst,
               "AboutBox",
               hWnd,
               lpProcAbout);

           FreeProcInstance(lpProcAbout);
           break;
         }
         else
           return (DefWindowProc(hWnd, message, wParam, lParam));

        case WM_CREATE:
```

```
            /* Create the brush objects */

            hRedBrush =   CreateSolidBrush(RGB(255,   0,   0));
            hGreenBrush = CreateSolidBrush(RGB(  0, 255,   0));
            hBlueBrush =  CreateSolidBrush(RGB(  0,   0, 255));

            /* Create the "---" pen */

            hDashPen = CreatePen(PS_DASH,                /* style */
                1,                                       /* width */
                RGB(0, 0, 0));                           /* color */

            /* Create the "..." pen */

            hDotPen = CreatePen(2,                       /* style */
                1,                                       /* width */
                RGB(0, 0, 0));                           /* color */
            break;

        case WM_PAINT:
            {
           TEXTMETRIC textmetric;
           win_int nDrawX;
           win_int nDrawY;
           char szText[300];

           /* Set up a display context to begin painting */

                hDC = BeginPaint (hWnd, &ps);

                /* Get the size characteristics of the current font.  */
                /* This information will be used for determining the  */
                /* vertical spacing of text on the screen.            */

                GetTextMetrics (hDC, &textmetric);

                /* Initialize drawing position to 1/4 inch from the top  */
                /* and from the left of the top, left corner of the      */
                /* client area of the main windows.                      */

                nDrawX = GetDeviceCaps (hDC, LOGPIXELSX) / 4;   /* 1/4 inch
*/
                nDrawY = GetDeviceCaps (hDC, LOGPIXELSY) / 4;   /* 1/4 inch
*/

                /* Send characters to the screen.  After displaying each   */
                /* line of text, advance the vertical position for the     */
                /* next line of text.  The pixel distance between the top  */
```

```
                /* of each line of text is equal to the standard height of */
                /* the font characters (tmHeight), plus the standard       */
                /* amount of spacing (tmExternalLeading) between adjacent  */
                /* lines.                                                  */

                strcpy (szText, "These characters are being painted using ");
                TextOut (hDC, nDrawX, nDrawY, szText, strlen (szText));
                nDrawY += textmetric.tmExternalLeading + textmetric.tmHeight;

                strcpy (szText, "the TextOut( ) function, which is fast and
");
                TextOut (hDC, nDrawX, nDrawY, szText, strlen (szText));
                nDrawY += textmetric.tmExternalLeading + textmetric.tmHeight;

                strcpy (szText, "allows programmer control of placement and
");
                TextOut (hDC, nDrawX, nDrawY, szText, strlen (szText));
                nDrawY += textmetric.tmExternalLeading + textmetric.tmHeight;

                strcpy (szText, "formatting details.  However, TextOut( ) ");
                TextOut (hDC, nDrawX, nDrawY, szText, strlen (szText));
                nDrawY += textmetric.tmExternalLeading + textmetric.tmHeight;

                strcpy (szText, "does not provide any automatic
formatting.");
                TextOut (hDC, nDrawX, nDrawY, szText, strlen (szText));
                nDrawY += textmetric.tmExternalLeading + textmetric.tmHeight;

                /* Put text in a 5-inch by 1-inch rectangle and display it.
*/
                /* First define the size of the rectangle around the text
*/

                nDrawY += GetDeviceCaps (hDC, LOGPIXELSY) / 4;  /* 1/4 inch
*/
                SetRect (
                      &rcTextBox
                    , nDrawX
                    , nDrawY
                    , nDrawX + (5 * GetDeviceCaps (hDC, LOGPIXELSX)) /* 5" */
                    , nDrawY + (1 * GetDeviceCaps (hDC, LOGPIXELSY)) /* 1" */
                );

                /* Draw the text within the bounds of the above rectangle */

                strcpy (szText, "This text is being displayed with a single "
                            "call to DrawText( ).  DrawText( ) isn't as fast
"
                            "as TextOut( ), and it is somewhat more "
```

```
                        "constrained, but it provides numerous optional "
                        "formatting features, such as the centering and "
                        "line breaking used in this example.");
                DrawText (
                      hDC
                    , szText
                    , strlen (szText)
                    , &rcTextBox
                    , DT_CENTER | DT_EXTERNALLEADING | DT_NOCLIP
                                                | DT_NOPREFIX | DT_WORDBREAK
                );

                /*  Paint the next object immediately below the bottom of
*/
                /*  the above rectangle in which the text was drawn.
*/

                nDrawY = rcTextBox.bottom;

                /* The (x,y) pixel coordinates of the objects about to be
*/
                /* drawn are below, and to the right of, the current
*/
                /* coordinate (nDrawX,nDrawY).
*/

                /* Draw a red rectangle.. */

                hOldBrush = SelectObject(hDC, hRedBrush);
                Rectangle (
                      hDC
                    , nDrawX
                    , nDrawY
                    , nDrawX + 50
                    , nDrawY + 30
                );

                /* Draw a green ellipse */

                SelectObject(hDC, hGreenBrush);
                Ellipse (
                      hDC
                    , nDrawX + 150
                    , nDrawY
                    , nDrawX + 150 + 50
                    , nDrawY + 30
                );

                /* Draw a blue pie shape */
```

```
                SelectObject(hDC, hBlueBrush);
                Pie (
                      hDC
                    , nDrawX + 300
                    , nDrawY
                    , nDrawX + 300 + 50
                    , nDrawY + 50
                    , nDrawX + 300 + 50
                    , nDrawY
                    , nDrawX + 300 + 50
                    , nDrawY + 50
                );

                nDrawY += 50;

                /* Restore the old brush */

                SelectObject(hDC, hOldBrush);

                /* Select a "---" pen, save the old value */

                nDrawY += GetDeviceCaps (hDC, LOGPIXELSY) / 4;  /* 1/4 inch
*/
                hOldPen = SelectObject(hDC, hDashPen);

                /* Move to a specified point */

                MoveTo(hDC, nDrawX, nDrawY);

                /* Draw a line */

                LineTo(hDC, nDrawX + 350, nDrawY);

                /* Select a "..." pen */

                SelectObject(hDC, hDotPen);

                /* Draw an arc connecting the line */

                Arc (
                      hDC
                    , nDrawX
                    , nDrawY - 20
                    , nDrawX + 350
                    , nDrawY + 20
                    , nDrawX
                    , nDrawY
```

```
                    , nDrawX + 350
                    , nDrawY
                );

                /* Restore the old pen */

                SelectObject(hDC, hOldPen);

                /* Tell Windows you are done painting */

                EndPaint (hWnd,  &ps);
            }
            break;

     case WM_DESTROY:
            DeleteObject(hRedBrush);
            DeleteObject(hGreenBrush);
            DeleteObject(hBlueBrush);
            DeleteObject(hDashPen);
            DeleteObject(hDotPen);
        PostQuitMessage(0);
        break;

     default:
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}


/****************************************************************************

    FUNCTION: About(HWND, unsigned, WORD, LONG)

    PURPOSE:  Processes messages for "About" dialog box

    MESSAGES:

     WM_INITDIALOG - initialize dialog box
     WM_COMMAND    - Input received

****************************************************************************/

BOOL  About(HWND hDlg,unsigned message,WORD wParam,LONG lParam)
{
    switch (message) {
     case WM_INITDIALOG:
        return (TRUE);
```

```
     case WM_COMMAND:
         if (wParam == IDOK
                 || wParam == IDCANCEL) {
           EndDialog(hDlg, TRUE);
           return (TRUE);
         }
         break;
    }
    return (FALSE);
}
_
```

PB.CPP

Copy to Clipboard

```
#pragma windows 500000,500000

#include <windows.h>
#include <stdlib.h>
#include <string.h>
int kh=0,kv=0;

char* textures[]={"Sticky","Messy","Dirty","Greasy","Slimy","Very
slippery",NULL};

int no_action()
{
   return 2;
}

int action()
{
       winio("No action today\n\n%cn%bt[Thank you]");
       return 2;
}
main()
{
    char name[20]="Thick and Sicky";
    int k1=1,k2=2,k4=4,k5=5;
    int texture_type=4;
    double v=4.5,p=200,a=25.2;
    int defv=1;
    char st[40],st1[20];


    winio("%ww[casts_shaddow]Double click on an item to change it\n\n&");
    winio("%30.8pb[sorted]&");
    winio("%dp[test1]&",&k1);
    winio("%dp[test2]&",&k2);
    winio("%ep[Oil texture]&",&textures,&texture_type);
    winio("%dp[Temperature (Deg C)]&",&k4);
    winio("%dp[Carbon monoxide (%)]&",&k5);
    winio("%fp[Curent (Amps)]&",&a);
    winio("%10.3fp[Voltage]&",&v);
    winio("%fp[Oil pressure (PSI)]&",&p);
    winio("%?tp[Oil name][What is the official name of this
oil?]&",&name,20);
    winio("%up[Special action]&",action);

    itoa(2,st,10);
```

```
strcpy(st1,"default value=");
strcat(st1,st);

winio("%up@&",st1,no_action);
winio("");
}
```

POPUP.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdlib.h>


double v1=0,v2=0,sum=0,mode=0;
int z=1;


int cbadd()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=2;
  return 2;
}

int cbsub()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=3;
  return 2;
}

int cbmul()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=0;
  return 2;
}

int cbdiv()
{
  if ((v1==0.0) || (v2==0.0)) z=0; else z=1;
  mode=1;
  return 2;
}




int math()
{
 switch( mode )
 {
  case 0: sum=v1*v2; break;
  case 1: if ( (v1==0.0) || (v2==0.0) )
```

```
                { z=0; break; }
          sum=v1/v2; break;
  case 2: sum=v1+v2; break;
  case 3: sum=v1-v2; break;
 }
 window_update(&sum);
 return 2;
}



int  main()
{
 winio("%ca[Popup]\n %10sl[vertical] %10sl[vertical]
\n&",&v1,0.0,100.0,&v2,0.0,100.0);
 winio("\n  Value 1 is %df%rf \n\n  Value 2 is %df%rf&",0.1,&v1,0.1,&v2);
 winio("\n\n The sum is %^tt[Do Math]  %`rf&",math,&sum);
 winio("%pm[Multiply,~Divide,|,Add,Subtract]",cbmul,&z,cbdiv,cbadd,cbsub);
 return 0;
}
```

PS.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdio.h>

#include <dbos/graphics.h>


#define say(s) MessageBox(NULL,s,"Yo ...",MB_OK | MB_ICONSTOP);



int g1=1,g2=2,wh1,wh2,ctrl,quit=1,col1=1,col2=5,loop,x,y,f,shhet=0;
char strx[]="0000",stry[]="0000",strf[]="00",shtno[]="01",sheet=1;

int lx[999],ly[999],count=0;


int cb()
{

 select_graphics_object(g2)  ;

  get_mouse_info(x,y,f);

  sprintf(strx,"%d",x);
  sprintf(stry,"%d",y);
  sprintf(strf,"%d",f);

  window_update(strx);
  window_update(stry);
  window_update(strf);

  if (f&1)
  {

   set_device_pixel(x,y,15);

   lx[count]=x;
   ly[count]=y;

   count++;
   if (count>999) count=999;
  }
 return 2;
}
```

```
int redraw1()
{

  select_graphics_object(g1)  ;
    set_graphics_selection(1);
    for(loop=0; loop<400;loop++) draw_line(0,loop,400,loop,(loop+col2)%255);
  col2=(col2+=4)%255;

  return 2;

}

int redraw2()
{
 int l,d;

 select_graphics_object(g2);

 set_line_width(1);

 for(l=0;l<400;l++) draw_line(0,l,400,l,0);

 for(d=1;d<count;d++)
  for(l=0;l<count;l++)
  {
   draw_line(lx[d],ly[d],lx[l],ly[l],14);
  }

  count=0;
  return 2;
}



int pst()

{
  sheet=clearwin_info("SHEET_NO");
  sprintf(shtno,"%d",clearwin_info("SHEET_NO"));
  window_update(shtno);

 return 2;
}
```

```
int main()
{
// sheet 1
 winio("%sc%ca[Colour Scroll]%`gr[black]\t\n\t %7^bt[redraw]
\n%sh",redraw1,400,300,g1,redraw1,&wh1);

// sheet 2
 winio("%sc%ca[Join The Dots]%`^gr[black,full_mouse_input]\t\n\t
%7^bt[redraw]\n\n\n\n\n X=%`rs\n\n  Y=%`rs\n\n
F=%`rs%sh",redraw2,400,300,g2,cb,redraw2,strx,4,stry,4,strf,2,&wh2);

 winio("%bg&",GetSysColor(11));
 winio("%ca[Property Sheet Example]%`bg%^2ps\t%8^bt[EXIT]\n\n sheet
%`rs%lw",GetSysColor(11),&wh1,&wh2,pst,"EXIT",shtno,2,&ctrl);
 return 0;
}
```

RESIZE.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <dbos\graphics.h>

HWND hwnd;
RECT rect;

int draw_func()
{
   // keep the original position of the window
   GetWindowRect(hwnd,&rect);

   draw_line(0,0,200,200,15);

   return 2;
}

//-----------------------------------------------------

int res_func()
{
   int s;


   // get the style of the window
   s = GetWindowLong(hwnd,GWL_STYLE);

   if(!(s & WS_MAXIMIZE))
   {
      // the window is not maximized
      SetWindowPos(hwnd,0,0,0,400,400,SWP_NOZORDER | SWP_NOMOVE);
   }
   else
   {
      // don't call SetWindowPos because :
      // the window is maximized
      MoveWindow(hwnd,rect.left,rect.top,400,400,1);
      SetWindowLong(hwnd,GWL_STYLE,s & ~WS_MAXIMIZE);
   }

   return 1;
}

//-----------------------------------------------------

main()
```

```
{
   static int ctrl;

   // let the window open to be able to resize
   winio("%ww Please select Resize %nl%nl%sc%pv%gr[black,metafile_resize]
%lw&",
         draw_func,200,200,&ctrl);

   winio("%mn[&File[E&xit],&Resize]","EXIT",res_func);

   hwnd = (HWND)clearwin_info("latest_formatted_window");
}

//-------------------------------------------------------
//-------------------------------------------------------
```

SLIDER.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>

double value=5.0;

int main()
{
   winio("%ca[Slider Controls]%bg&",GetSysColor(11));
   winio("%30sl[horizontal]\n\n&",&value,0.0,10.0);
   winio("\n\n %fl%rf",0.0,10.0,&value);
}
```

STOP.CPP

Copy to Clipboard

```
//
//  Stop.cpp
//
//  This program illustrate the use of the gauge VBX to simulate a simple
//  second counter/stopwatch.
//
//  This application uses the gauge control as supplied with Visual Basic
//  version 3.0.  This control is not shipped with ClearWin+.
//
//  (C)opyright Salford Software Ltd 1995.
//
#pragma windows 500000, 500000
#include <windows.h>
#include <stdlib.h>
#include <time.h>

//
//  Preprocessor macros.
//
#define TITLE "Gauge Control Example"
#define INTEGER_PROPERTY(x, y, z) set_vbx_integer_property(x, y, z)

//
//  Now for the global variables for ClearWin+.
//
int counting;

//
//  Pause for 1 second.  Yield control to the system for a while.
//
void WaitOneSecond()
{
    time_t  start, now;

    start = time(NULL);
    do
        {
        yield_program_control(1);           // Don't hog the system.
        now = time(NULL);
        }
    while (difftime(now, start) < 1);
}

//
//  Reprt a fatal program error and abort the program run.
//
```

```
void FatalError(char *message)
{
    #pragma silent 287
    int     result;

    result = winio("%ca[" TITLE "]&");
    result = winio("Error : %ws&", message);
    result = winio("%dn%8bt[&Ok]");
    exit(-1);
}

//
//  Use a double mouse click to turn the stop watch on/off.  This is a
//  toggle function.
//
int DoubleClick()
{
    if (counting)
        counting = 0;
      else
        counting = 1;

    return(1);
}

//
//  Now for the main program loop.
//
#pragma silent 288
void main(void)
{
    #pragma silent 287
    int     result, vbx_handle, value, control;

    result = load_vbx("gauge.vbx");
    if (!result)
        FatalError("Cannot load the Gauge control");
    //
    //  Now build up the output window.  The window has a title, a single
    //  gauge in the window with a button to exit the program.  All controls
    //  are centred.
    //
    //  Note the use of the standard call back function "STOP" to terminate
    //  the program.
    //
    result = winio("%ca[" TITLE "]&");
    result = winio("Double clink on the bar to start/stop the counter\n\n&");
```

```
    result = winio("%cn%^50.1vb[BIGAUGE][^=DblClick]\f\n&", &vbx_handle,
DoubleClick);
    result = winio("%cn%^8bt[&Exit]%lw", "STOP", &control);
    //
    //  Set up some properties for the VBX.
    //
    INTEGER_PROPERTY(vbx_handle, "Min", 0);
    INTEGER_PROPERTY(vbx_handle, "Max", 59);
    //
    //  Now for the main stopwatch functionality
    //
    counting = 0;
    value = 0;
    while (1)
        {
        WaitOneSecond();
        if (counting)
            {
            value++;
            value = (value % 60);
            INTEGER_PROPERTY(vbx_handle, "Value", value);
            }
        }
}
```

SV.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdlib.h>
#include <dbos\graphics.h>
#define LAST 10

int h_gr=1,xres,yres,l=0,c=0,coldx[LAST],coldy[LAST],refresh=500;


int col=0;

int screen_update()
{
 static int x,y;
 static float m,floop;

 select_graphics_object(h_gr);
 x=rand()%xres;
 y=rand()%yres;

 floop=(float)(rand()%8)+1;


 clear_screen();


   for(m=0;m<360;m+=4)
   {
    set_text_attribute(1,floop,m,0);
    italic_font(1);
    draw_text(" +Salford Software",x,y,col);
    col=(++col)%255;
   }



 return 2;
}




int main()
{
 xres=clearwin_info("SCREEN_WIDTH");
 yres=clearwin_info("SCREEN_DEPTH");
```

```
 winio("%sv%bg[black]
%ww[no_caption,no_maxminbox,no_sysmenu,no_border,topmost]
%`gr[black]&",xres,yres,h_gr);
 winio("%dl",10.0,screen_update);

 return 0;
}
```

TRACE.CPP

Copy to Clipboard

```
#pragma windows 500000,500000

#include <stdlib.h>
#include <windows.h>
#include <dbos\graphics.h>

int update_func();
int startup_func();

RECT r={0,0,500,100};
int bg=GetSysColor(11);

//-------------------------------------

int main()
{
   winio("%ca[SCROLLING trace]&");
   winio("%bg&",bg);
   winio("%ob[scored]&");
   winio("%`gr[black]%cb&",500,100,1);
   winio("%dl&",0.1,update_func);
   winio("%sc",startup_func);
}

//-------------------------------------

int update_func()
{
   int l;
   static int old_y,y=50;


   old_y=y;

   y=rand() % 100;

   // paint region to update in red
   scroll_graphics(-25,0,r.left,r.top,r.right,r.bottom,1,0);

   // update suqare pattern
   draw_line(475,0,475,100,7);
   for(l=0;l<100;l+=20) draw_line(474,l,500,l,7);

   // update graphics
   draw_line(475,old_y,500,y,10);
```

```
   return 2;
}

//-------------------------------------

int startup_func()
{
   int l;

   // draw square pattern

   // vertical lines
   for(l=0;l<500;l+=25)  draw_line(l,0,l,100,7);

   // horizontal lines
   for(l=0;l<100;l+=20)  draw_line(0,l,500,l,7);

   return 2;
}

//-------------------------------------
//-------------------------------------
```

TT.CPP

Copy to Clipboard

```
#pragma windows 500000,500000

#include <windows.h>


main()
{

  winio("%ww[no_border]&");
  winio("%ca[TT as tool bar]%tt[File]%tt[Edit]%tt[Search]%tt[Window]
%tt[Help]&");
  winio("%nl %nl ");

 return 0;
}
```

TX.CPP

Copy to Clipboard

```
#pragma windows 500000,500000

#include <stdlib.h>
#include <windows.h>
#include <string.h>

char t1[5][3],t2[5][3];
char td1[6],td2[6];
char a1[6],a2[6];
int pos=0;


// callback for vertical scrolling

int vscroll_func()
{
   int i,j,k;

   // the data displayed in the tx box must be moved
   // by the used dependant on the position of the scroll tab.
   // set text array according to the position of the slider
   for(i=0,j=pos,k=0;i<6;i++,k++)
   {
      if(k==2)
      {
         j++;
         k=0;
      }

      td1[i]=t1[j][k];
      td2[i]=t2[j][k];
   }

   window_update(td1);
   window_update(td2);

   return 1;
}


// main code
main()
{
   static int ctrl;
   int i,j,k;
```

```
// 1st array to scroll
strcpy(t1[0],"a1");
strcpy(t1[1],"b1");
strcpy(t1[2],"c1");
strcpy(t1[3],"d1");
strcpy(t1[4],"e1");

// 2nd array to scroll
strcpy(t2[0],"a2");
strcpy(t2[1],"b2");
strcpy(t2[2],"c2");
strcpy(t2[3],"d2");
strcpy(t2[4],"e2");

// initalise text array with the first 3 lines to display
for(i=0,j=0,k=0;i<6;i++,k++)
{
   if(k==2)
   {
      j++;
      k=0;
   }

   td1[i]=t1[j][k];
   td2[i]=t2[j][k];
}

// attributes array
for(i=0;i<6;i++)
{
   a1[i]=0;
   a2[i]=0;
}

winio("%ww%ca[testing tx routine]\n%lw&",&ctrl);

// page size:1
// range:3
winio("%^vs&",1,3,&pos,vscroll_func);

// tab position at ten
winio("%1tl&",10);

winio("  1%ta  2%nl&");

// size of tx box : 3x3
// text array is 2 (width) x 3 (length)
winio("%ob%3.3tx%cb%ta&",td1,a1,2,3);
```

```
   winio("%ob%3.3tx%cb",td2,a2,2,3);
}
```

VXTX.CPP

Copy to Clipboard

```
#pragma windows 500000,500000

#include <stdio.h>
#include <string.h>
#include <windows.h>


#define GMEMORY 0xFFFF
#define LMEMORY 1600

#define LEN_ONE_LINE 80
// 2 screens of 20,10 text
char gdata[GMEMORY],gattrib[GMEMORY];

char ldata[LMEMORY],lattrib[LMEMORY];

int vscroll,hscroll;
int cb_textarray();
int cb_vxhx();

////////////////////
//
////////////////////

char *string[]={"winio","for","while","int","char","if","else","return",
                "#define","#pragma","windows","#include", NULL};


main()
{
 int l=0,line=0,cnt,cmp,current,found;
 int step=10,maxpage=0,lenoneline=LEN_ONE_LINE;


 FILE *fp;
 char c;

 memset(gdata , 32,GMEMORY);
 memset(gattrib, 0,GMEMORY);
 memset(ldata , 32,LMEMORY);
 memset(lattrib, 0,LMEMORY);



 fp=fopen("data.dat","r");
 if (!fp) { winio("%si! Cannot find the data file. \n"); return 0;}
```

```
while ( ((c=fgetc(fp)) !=EOF) && ( l+(maxpage*LEN_ONE_LINE)<GMEMORY) )
  {
    if (c<32)
    {
     line+=LEN_ONE_LINE;
     l=0;
     maxpage++;
    }
    else
     gdata[line+(l++)]=c;
  }

fclose(fp);

current=0;

// reserved words
while(string[current]!=NULL)
{
 for(cnt=0;cnt<LEN_ONE_LINE*maxpage;cnt++)
 {
   // search
   found=1;
   for(cmp=0;cmp<strlen(string[current]);cmp++)
     if (gdata[cnt+cmp]!=*(string[current]+cmp)) {found=0; break;}
   // highlight
   if (found)
   {
     for(cmp=0;cmp<strlen(string[current]);cmp++)  gattrib[cmp+cnt]=1;
     cnt+=strlen(string[current]);
   }

 }
 current++;
}

// brackets
for(cnt=0;cnt<LEN_ONE_LINE*maxpage;cnt++)
{

 if ( gdata[cnt]>='0' && gdata[cnt]<='9') gattrib[cnt]=5;
  else
 if ( gdata[cnt]=='{' || gdata[cnt]=='}' ) gattrib[cnt]=2;
  else
 if ( gdata[cnt]=='(' || gdata[cnt]==')' ) gattrib[cnt]=3;
  else
 if ( gdata[cnt]=='[' || gdata[cnt]==']' ) gattrib[cnt]=4;
```

```
 }

 memcpy( &ldata,   &gdata,   LMEMORY);
 memcpy( &lattrib, &gattrib, LMEMORY);


 winio("%ca[TX Box With Scroll Bar Example]%bg[grey]&");

 // 80x20 text box with added horiz and vert scroll bars
 winio("%ob[depressed]%^hx%^vx%^80.20tx[full_char_input]&",
        &step,&lenoneline,&hscroll,cb_vxhx,
        &step,&maxpage,&vscroll,cb_vxhx,
        ldata,lattrib,80,20,cb_textarray);

 winio("%tc[RED]%ty[GREY]%tc[BLUE]&"); // colour attrib 1

 winio("%ty[GREY]%tc[BLUE]&");         // colour attrib 2

 winio("%ty[YELLOW]%tc[GREEN]&");      // colour attrib 3

 winio("%ty[GREY]%tc[WHITE]&");        // colour attrib 4


 winio("%ty[GREY]%cb&");  // set the text colour for the next
 winio("%ff\n \n%ob[status,thin_panelled]%tc[black]X%`3rd
Y%`3rd%cb",&hscroll,&vscroll);

 return 0;
}


//////////////////////
//
//////////////////////
int cb_vxhx()
{
int l,vert, verthoriz,shift_dist;
 memset(ldata , 32,LMEMORY);
 memset(lattrib, 0,LMEMORY);

 vert=LEN_ONE_LINE*vscroll;
 shift_dist=LEN_ONE_LINE-hscroll;

 for(l=0;l<20;l++)
 {
  verthoriz=(LEN_ONE_LINE*l)+hscroll+vert;
  memcpy(ldata+(LEN_ONE_LINE*l), gdata+verthoriz,shift_dist);
  memcpy(lattrib+(LEN_ONE_LINE*l), gattrib+verthoriz,shift_dist);
```

```
 }

 window_update(&vscroll);
 window_update(&hscroll);
 window_update(ldata);
 return 2;
}


//////////////////////
//
//////////////////////
int cb_textarray()
{
 return 2;
}
```

WAVE.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>
#include <stdlib.h>
#include <dbos\graphics.h>

int sx=500,sy=200;



int redraw()
{
 int x,rtx;

 fill_rectangle(sx-100,0,sx,sy,0);
 rtx=sx-100;

 for(x=0;x<100;x++) draw_line(rtx+x,sy,rtx+x,sy-(rand()
%sy),RGB(255,100+x,0));

 copy_graphics_region(1,0,0,sx,sy,1,100,0,sx,sy,0xcc0020);

 return 2;
}



int resize()
{
 if (clearwin_info("graphics_resizing"))
 {
   sx=clearwin_info("graphics_width");
   sy=clearwin_info("graphics_depth");
 }

 redraw();
 return 2;
}



main()
{
 winio("%ca[Scrolling Wave]%pv%ww[no_border]
%`^gr[black,rgb_colours,user_resize]&",500,200,1,resize);

 winio("%dl",0.002,redraw);

 return 0;
```

```
}
```

WINIOG.CPP

Copy to Clipboard

```
#pragma windows 500000,500000
#include <windows.h>

int  main()
 {
  long v;
  winio("%ca[colour palette] %cl[]",&v);
  winio("%ca[colour result] \n\nThat colour was %wd (%wd)
\n\n",v,RGB(255,255,255));

  return 0;
 }
```