Winteracter Starter Kit

Revision G

P. O. Box 6091
Incline Village, NV 89450

Printed on 50%
recycled paper

Lahey Computer Systems, Inc.
865 Tahoe Boulevard
P.O. Box 6091
Incline Village, NV 89450-6091
(775) 831-2500
Fax: (775) 831-8123

Technical Support
(775) 831-2500
support@lahey.com
www.lahey.com

Table of Contents

Introduction

W*interacter* is a portable Fortran 9x dedicated user-interface and graphics development toolset. It combines *INTERACTER*-compatible graphics with GUI components based on the Win32 or Motif API's. In addition to a Fortran 9x subroutine library, W*interacter* also provides visual user interface design tools.

The W*interacter* Starter Kit (W*i*SK) is derived from the full version of W*interacter*. It includes W*i*SK-specific versions of the 'resource editor' (the menu/dialog/image designer) plus a library of subroutines organised in five categories :

Window Management

Input Handling

Dialog Management

High Resolution Graphics

General Functions

Each category is sub-divided into groups, which are identified by two-character codes, e.g., GT for graphics text manipulation, MH for message handling and so on. The following sections provide a general summary of the facilities provided by each of these subroutine groups. Before starting to use W*i*SK it is recommended that you browse through the following sections to familiarise yourself with the range of features on offer and more importantly, where to find them. The subroutine group summaries presented here follow the same order as the subroutine reference sections later in this manual.

Window Handling

W*interacter* supports a single root window and multiple child windows.

WM: Window Management

These routines open and close root and child windows. Windows can be hidden or combined with dialogs, if required. The current output window is also selectable. Status bar control is provided. Window size/position/state control is provided. All or part of the current window can be cleared.

Input Handling

This group provides the fundamental message delivery mechanism plus menu handling facilities.

MH: Message Handling

All input is reported to the program via the message delivery routines in this group. In particular, a typical W*interacter* program will revolve around an event loop which repeatedly calls the `WMessage` routine.

MN: Menu Handling

Menu layouts are defined separately in a resource file, using the W*interacter* resource editor. Menu selections are reported via the message handling routines in the MH group. This group therefore deals with menu activation and updating the state of individual menu items (e.g. setting the ‘checked/unchecked’ state).

Dialog Management

Dialog layouts are defined externally in a resource script, using the W*interacter* resource editor. The routines in the Dialog Manager are therefore mostly concerned with activating, controlling and interrogating these dialogs.

DM(1): General Form Creation & Editing

Dialog activation and selection is controlled through this group.The state of individual fields can also be controlled.

DM(2): Assign/Retrieve Field Contents

The individual contents of each dialog field are accessed via these routines.

CD : Common Dialogs

Pre-defined dialogs are accessible for file selection and message boxes.

High Resolution Graphics

These routines provide high resolution graphics output facilities.

GG: General Graphics

A number of general graphics facilities are grouped together under this heading, such as target drawable selection, pixel colour interrogation and area/co-ordinate system selection.

GS: Graphics Style Selection

Control is provided over color, line-type, fill style and plot mode.

GD: Graphics Drawing/Movement

These are the basic drawing primitives. In addition to simple move and draw functions, polygon and circle fill routines are provided.

GT: Graphics Text

Graphics text output supports both driver-specific and software fonts. Under Windows, any TrueType font is selectable. Various vector and outline software fonts sets are provided. Full control is provided over font style, size and orientation.

General Functions

The remaining routines provide a variety of functions which are likely to be required in interactive applications.

IF: Information

A set of functions are provided which enable a program to interrogate the current state of the routines in the W*interacter* library and the hardware on which the program is currently running.

OS: Operating System

Environment variable access and termination with exit code routines are provided.

MI: Miscellaneous

The most important routine in this group is `WInitialise`, the W*interacter* initialisation routine. OpenGL support is also enabled via this group. The mouse cursor is selectable

CH: Character Manipulation

These routines provide string manipulation facilities which are useful in interactive applications.

OB: Obsolete Routines

A number of obsolete routines are retained for compatibility with earlier releases.

1 Supplied Files

This chapter summarises the files which you receive with your W*interacter* Starter Kit. These are installed as part of the compiler installation procedure. Refer to *"Building a WiSK Program"* on page 3 for information on how to use and set up W*interacter* once it has been installed.

The W*interacter* Starter Kit file set includes the following:

- The W*interacter* Resource Editor (`bin` directory)
- The W*i*SK Application W*izard* (`bin` directory)
- On-line help covering various topics including the W*i*SK FAQ (`help` directory)
- W*interacter* Starter Kit library (`lib` directory)
- `WINTERACTER` module (`lib` directory)
- Lahey Video Graphics Library emulation source code (`src` directory)
- Various W*i*SK demo programs (`examples` directory)
- Various OpenGL demos (`examples` directory)

Several demonstration programs are provided in the `WiSK` sub-directory of the LF95 `examples` directory. They illustrate various aspects of W*interacter* user interface and graphics programming. Each sub-directory contains a single demo consisting of a source file (with a `.f90` extension), a resource script (`resource.rc`) and a program icon (`winter.ico`). Some directories will contain additional files dependent on the purpose of the demo. The W*i*SK OpenGL demos are organised in a similar manner. Alternatively, the W*i*SK Application W*izard* (`wiskwiz`) can be used to create a tailored template W*interacter* program.

2 Building a W*i*SK Program

A W*interacter* program will consist of Fortran source code and a resource script. The latter describes menus, dialogs, etc. required by that program. Building a W*interacter* program therefore consists of three distinct tasks:

(1) Compiling the calling Fortran application source code to create object files.

(2) Compiling the resource script (`.rc`) to produce object files.

(3) Linking the resulting object files from steps (1) and (2) with the W*interacter* library.

Command Line

The LF95 driver will handle these tasks automatically if you specifiy the `-wisk` (Win32) or `--wisk` (Linux) switch on the command line, along with the Fortran source file name(s) and the resource script name. (Note : In the following command line examples, `<id>` refers to your compiler installation directory.

Under Windows, try the following in the `\<id>\examples\WiSK\example` directory:

```
lf95 example.f90 resource.rc -wisk
```

which is equivalent to the following commands:

```
lf95 example.f90 -win -c -mod .;\<id>\lib
rc /i \<id>\src resource.rc
res2obj resource.res resource.obj
lf95 example.obj resource.obj -win -lib \<id>\lib\winter.lib
```

Under Linux, try the following in the `/<id>/examples/WiSK/example` directory:

```
lf95 example.f90 resource.rc --wisk
```

which is equivalent to:

```
lf95 example.f90 -c -I /<id>/lib
rc resource.rc
lf95 -o example example.o resource.o -L/<id>/lib -L/usr/X11R6/lib
-lwint -lXm -lXt -lX11
```

These commands would compile and link the supplied `example` demonstration program, along with the accompanying `resource.rc` resource file. Both are simply examples. There is no special significance to the choice of these file names. The `example` program is discussed in some detail in the next chapter.

The W*interacter* library is called `winter.lib` (Win32) or `libwint.a` (Linux) and is installed in the `lib` sub-directory. This directory will also contain the `WINTERACTER` module, which defines data types, interface definitions and symbolic names. All W*interacter* based programs `USE` this module.

Note : The W*interacter* library and the associated `WINTERACTER` module should be considered a pair. Trying to mix a library and module from different releases should therefore be considered an error. Hence, objects compiled with an earlier release of W*interacter* must be recompiled when upgrading to a new release, since those objects will otherwise use definitions based on an out of date module. This is not a W*interacter* issue as such, but is inherent in the implementation of Fortran modules.

Each W*interacter* application will have an accompanying resource script. This defines menus, dialogs, etc. used by the program. It will contain an include statement of the form:

```
#include "winparam.h"
```

Under Windows, the file `winparam.h` contains various standard parameter declarations and is installed in the installation's `src` directory. If you are not using the `-wisk` switch, then a command line argument should be used to identify this directory to the RC resource compilerc. Alternatively, RC will search the include path specified by the `INCLUDE` environment variable.(*Note* : `winparam.h` is not required or supplied under Linux. However, the `#include` statement should still be present in your resource file for portability reasons).

W*interacter* programs need to reference identifiers in the resource script. These identifiers are normally declared as `PARAMETER` values in a Fortran 90 module or include file. The W*interacter* resource editor can generate both file types. Where this file is saved as a module it should be compiled before the program which `USE`'s it.

Win32 executables built with W*i*SK will run on Intel systems under Windows 9x/Me or NT/2000/XP. Linux W*i*SK executables will run on whichever Intel Linux distributions are supported by the current LF95 release.

Linux users should be aware that X/W*i*SK relies on functions provided by the X Windows (Xlib and Xt) and Motif (Xm) libraries. All Linux distributions include Xlib and Xt, but Motif may not be included. Refer to the X/W*i*SK Getting Started Guide (`xgetstart.htm`) which is supplied in HTML format in LF95's `help` directory.

ED for Windows

ED4W can be used to build Win32 W*i*SK programs too. The main issue here is that ED4W normally assumes that your program has only a single source file. However W*interacter* programs consist of two source files, a Fortran program and a resource script, which must both be compiled and linked together. If you use a hardwired name for your resource script (specifically `resource.rc` as used in the `Examples\wisk` demo directories) here's one solution. Install the following batch file somewhere on your system :

```
LF95 %1.f90 resource.rc -wisk
```

Then, in ED use `Tool|Programs|Add` to add a new program. The command line should specify the full path of the above batch file, followed by <name>. This tells ED4W to substitute the name of the current program as argument number 1 of the batch file.

3 Writing W*interacter* Programs

This chapter aims to introduce you to writing software using W*interacter*. It assumes that you are already familiar with how to compile and link W*interacter* programs, as described in the previous chapter.

The first section summarizes some basic rules. Later in this chapter you will find a worked example which provides a gentle introduction to writing a W*interacter* application.

Basics

Certain basic principles apply regardless of which W*interacter* features you use.

Initialization

All W*interacter* programs must call `WInitialise` before opening a window. `WindowOpen` must be called before any window or dialog processing.

Fortran I/O

All screen I/O should be performed via W*interacter*. While you may find that console I/O works (e.g. `WRITE(*,..)`, `READ(*,..)`, `WRITE(6,..)` etc.), this will cause an extra output window to be opened under Windows. The program will have no direct control over this window. This will look untidy, at the very least. Fortran I/O is freely available on all other channels.

The `WINTERACTER` Module

A Fortran 90 module called `WINTERACTER` is supplied which provides three facilities:

- Type definitions for W*interacter* specific data structures.
- Interface definitions for W*interacter* routines.
- `PARAMETER` definitions for numerous symbolic names.

Use of the `WINTERACTER` module is required in any program unit which calls W*interacter* routines. i.e. add the following statement at the beginning of every program unit which uses W*interacter*:

```
USE WINTERACTER
```

When upgrading from an earlier version of W*i*SK, you must recompile all program units which `USE` the `WINTERACTER` module, before relinking your application. Interface or type definitions may change between releases. While such changes will be transparent to the calling program once recompiled, old objects which use an out-of-date module may not be relinkable. This recompilation requirement is fundamental to the way that Fortran 90 modules work.

Type Definitions

Various W*interacter* specific data types are defined in the `WINTERACTER` module, most notably, the `WIN_MESSAGE` structure used by `WMessage`.

Interface Definitions

As an aid to checking the number and type of arguments supplied to W*interacter* routines, the `WINTERACTER` module contains an extensive set of interface block definitions. These define the type and intent of all W*interacter* subroutine arguments and functions. They will cause the compiler to check the number and type of arguments in each W*interacter* call, potentially saving many hours of debugging.

Symbolic Names

The `WINTERACTER` module also contains a number of `PARAMETER` declarations which define symbolic names for W*interacter* subroutine arguments and function results. These symbolic names can be used in place of numeric subroutine arguments/results and are designed to be meaningful, aiding program readability. A handful of Microsoft-recommended push-button identifiers (e.g. IDOK) are also defined here. All of the `PARAMETER` statements contain type declarations, so no further definition is required outside of the module.

While use of the symbolic names defined in the `WINTERACTER` module files is not obligatory, their use is recommended. They are documented in the subroutine reference section of this manual and form part of the formal definition of W*interacter*.

Subroutine Arguments

All subroutine arguments of type `CHARACTER` may be of any length, except where explicitly documented otherwise. All subroutine arguments of type `INTEGER` are 4-byte integers. Similarly, all `REAL` arguments are single precision 4-byte variables

Subroutine and Common Block Names

All externally callable routines in W*interacter* start with the letter W or I. W*interacter*-specific routines begin with the letter W. Routines which are common to both W*interacter* and *INTERACTER* begin with an I.

Various internal subroutines and `COMMON` blocks are used. All internal subroutines have names starting with the letters XX or YY (e.g. `XXGDRV`). Internal `COMMON` blocks are named `WINTnn` where nn is a 2 digit number (e.g. `WINT01`). Avoid using subroutines or `COMMON` blocks with similar names.

Error Reporting

All error reporting in W*interacter* is performed via a single function called `InfoError`, which is in the IF subroutine group. Whenever W*interacter* encounters an error, it sets a global error flag which can be interrogated using `InfoError`. This global error flag may be over-written by subsequent errors, but is never cleared until `InfoError` is called. It is the callers responsibility to decide when to interrogate and/or clear the error flag and issue any appropriate error messages. When W*interacter* encounters an error it will simply update the error flag and attempt to take suitable default action. W*interacter* will not report errors to the screen, since this may not be appropriate in many applications.

If a routine sets the W*interacter* error flag, the values which it may set it to are documented with the description of the routine. A summary of the W*interacter* error codes is provided later in this manual. Symbolic names for all the possible error codes are defined in the `WINTERACTER` module.

On-line Help

On-line help is provided either as a Windows help file called `wisk.hlp` (under Win32) or in HTML format as `wisk.htm` (under Linux). This contains a variety of useful W*interacter* related information, including brief subroutine argument summaries, an FAQ, details of supplied demos, error codes, a glossary and a description of the resource file format. It also describes supported graphics interfaces such as OpenGL and the supplied emulation of the Lahey Video Graphics Library. The Windows help file can be accessed via the LF95 Start menu options. The Linux HTML file should be viewed using Netscape 4.x. More usefully, Linux users may wish to start browsing at the `index.htm` page which provides access to all of X/W*i*SK’s on-line help, which also includes a "Getting Started" guide.

Elements of a W*interacter* Program

A W*interacter* program consists of two main elements:

- A resource file describing menu structures, dialogs, icons, etc.
- Fortran 90 source code which calls routines in the W*interacter* library.

The resource file is created and managed by our resource editor (`resedit`). The Fortran 90 source code can be created from scratch or adapted from one of the many demonstration programs in the `demos` sub-directory. Alternatively, for a quick start, use the WiSK W*izard* (`wiskwiz`) to create a substantial W*interacter* starter application tailored to your requirements.

Resource Files

Each W*interacter* program requires a resource script to describe its menus and dialogs, along with miscellaneous information such as the program icon. This script must be created and maintained via the supplied resource editor (ResEdit). This tool allows menus, dialogs, icons, cursors and bitmap buttons to be created interactively. The resulting user interface descriptions are saved as resource (`.rc`) files which must be compiled using the resource compiler as described in the previous chapter. The resulting `.obj` (Win32) or `.o` (Linux) file can then be linked with a program which calls W*interacter* routines to produce a GUI application program. The `-wisk` (Win32) or `--wisk` (Linux) compiler command line argument will handle resource compilation automatically.

Normally, there is no need to know about the format of a resource file, since this is handled automatically by the supplied resource editors. However, their format is documented in the on-line help for the sake of completeness.

A W*interacter* program may only contain one compiled resource file. While it is feasible for multiple resource scripts to be `#include`'d into a 'parent' resource script, this will prevent the resource editor from properly maintaining the identifiers associated with that resource. It is therefore strongly recommended that the resource script be maintained as a single file.

Identifiers

Every user interface element in a resource file has a numeric identifier associated with it. Many W*interacter* routines require such an identifier to be specified as an argument. It is therefore important to maintain a separate Fortran module or include file which reproduces these identifier definitions (as `INTEGER PARAMETER` values) to allow the W*interacter* program to refer to resources via symbolic names. This file is generated and updated automatically by the resource editor. It must be `USE`'d or `INCLUDE`'d by the calling program to enable access to menus/dialogs/etc held in the program resource. This file will be referred to as the Symbol Header file. Several commonly used push-button identifiers (e.g. `IDOK` and `IDCANCEL` for OK/Cancel buttons) are defined in the `WINTERACTER` module (see `WMessage`).

As a general rule, identifier values should be non-zero unsigned 16-bit integers, i.e. they should be in the range 1-65535. However, some exceptions apply, mainly under Windows 9x/Me where identifiers for dialogs, bitmap/icon fields and menus should be in the range 1-32767. (***Note*** : Identifiers should still be stored as standard four byte integers, despite the Windows-imposed limit of 16-bit ranges.)

Remember that identifiers must be valid Fortran parameter names. Hence, they should not include characters such as ! , + - * ( ) > < etc. The resource editor will reject attempts to use such characters in identifier names.

If a resource file is amended manually (not normally recommended), the associated module/ include file can be regenerated by loading the resource file into the resource editor and resaving it.

Message Loop

Most program input is reported via a message queue (also known as an event queue in some other windowing systems; events and messages are the same thing). While the underlying windowing system reports many messages, W*interacter* only passes a much reduced subset of these messages up to the calling program, greatly simplifying the volume and type of messages which must be processed.

Messages are reported via the `WMessage` routine. Typical messages are 'a dialog button was pressed', 'a menu item was selected', 'a window changed size' and so on. The calling program will usually revolve around a `DO` loop which calls `WMessage` then checks the resulting message in a `SELECT CASE` statement. The following is a simplified example of such a loop:

```
USE WINTERACTER
TYPE (WIN_MESSAGE) :: MESSAGE
         !
DO       ! loop until termination
  CALL WMessage(ITYPE,MESSAGE)
  SELECT CASE (ITYPE)
    IWIN = MESSAGE%WIN   ! Originating window
    CASE (MenuSelect)    ! A menu item was selected
          ITEM = MESSAGE%VALUE1
    CASE (Resize,Expose) ! The window was resized or exposed
          CALL DrawMyGraph(IWIN)
    CASE (CloseRequest)  ! The user closed a window
          IF (IWIN==0) EXIT
    END SELECT
END DO
```

The location of the message loop is a matter of program choice. In a small program, it will make sense to place the message loop in the main program. However, this can rapidly lead to a 'top heavy' program in larger applications. It is therefore perfectly allowable to have multiple message loops in a program, provided they are capable of processing all the possible messages which can be reported at a given point in the program's execution.

See the MH group in the subroutine reference section for more information.

Windows

A W*interacter* application consists of a root window and up to 20 child windows. The latter exclude any dialog windows (see the section *"Dialogs"* on page 13). In this context a 'window' is a standard output window which can have any text or graphics written into it. Effectively they are free format output windows.

A root window is always opened first using `WindowOpen`. Child windows can then be opened using `WindowOpenChild`. The graphics routines in the GG/GS/GD/GT groups can be used to draw in these windows. Every window has a handle which is allocated automatically by W*interacter* when the window is opened. Use this to select the window to receive output in a call to `WindowSelect`.

In certain cases (e.g. when opening a child window inside a parent window) an arbitrary co-ordinate system is used which treats a window as being 10000x10000 "window units" square, measured from the top left corner of the window. Graphics output uses a separate REAL user-defined cartesian co-ordinate system (see the section *"Graphics"* on page 14).

When graphics are drawn to a window, the caller is responsible for maintaining the contents of that window. So if another window or dialog overlaps a window, the obscured area of the window will need to be repainted when the overlapping window/dialog is moved or closed. An `Expose` message is reported via `WMessage` in this case.

Windows can either be fixed in size or resizeable. In the latter case, the window co-ordinates are rescaled automatically if the user changes the size of the window. A `Resize` message is reported via `WMessage` in this case. The calling program will normally need to repaint the entire window when this message is received.

Menus

The contents and structure of program menus are defined in resource scripts using the resource editor. Use `WindowOpen`, `WindowOpenChild`, `WMenu` and/or `WMenuFloating` to activate these menus. Main menus remain visible at all times. Floating menus disappear when a selection is made or they are cancelled by the user. In either case, menus are managed automatically once displayed. Menu selections are reported via `WMessage`.

The contents and state of individual menu items can be modified at run-time. For example, menu items can be greyed to prevent them from being selected, by calling `WMenuSetState`.

Dialogs

Dialogs are collections of fields (or 'controls') which are displayed in a dedicated child window. (In other development systems dialogs are also known as 'panels' or 'forms'.) The layout and initial contents of a dialog must be defined in a resource file scripts created using the supplied resource editor.

Two basic types of dialogs are allowed : modal and modeless. Their resource file definitions are identical, but their behaviour when activated is different. A modal dialog will block all other program input until the user terminates the dialog. This makes for simpler program development, at the expense of a slightly less friendly user interface. Modeless dialogs do not block program execution and allow interaction with other windows/dialogs belonging to the same program. A third dialog type is supported by W*interacter*, known is 'semi-modeless'. These are a useful hybrid dialog type which appears modeless to the calling program but modal to the user. Such dialogs eliminate the need to use callback routines.

Under Windows, all dialogs are either 'pop-up' dialogs or 'child' dialogs (Motif dialogs are always 'pop-up'). A pop-up dialog can be moved to any position on the screen either inside or outside of the application window. Child dialogs are restricted to the root window. Child dialogs must be modeless. Alternatively a dialog can be combined with a window. Such dialogs are always modeless.

To activate a dialog call `WDialogLoad` and `WDialogShow`. Multiple simultaneous dialogs are allowed. See the introduction to the DM(1) group for more details.

Dialogs consist of various field types including strings, four styles of menu, push-buttons, radio buttons, progress bars and check boxes. These field types are described in more detail in the Dialogs chapter. The contents of most field types can be assigned and retrieved using the various 'put/get' routines in the DM(2) group.

W*interacter* dialogs use standard Windows or Motif controls so all the normal behaviour is available. Notably clipboard cut/paste is supported via the mouse or the usual keyboard shortcuts (Ctrl/C, Ctrl/V, etc.) Under Windows, the usual shortcuts menu is available via the right mouse button. W*interacter* dialogs also implement Ctrl/A as a shortcut for Select All in string fields.

In addition to application-specific dialogs, some pre-packaged modal dialogs are also available. The most useful of these (file selection and message box) are supported via the routines in the CD group.

Graphics

W*interacter*'s graphics routines are mostly compatible with the earlier *INTERACTER* library. The introduction to the GG group describes the basic principles of W*interacter* graphics programming.

Graphics can appear in any window opened via `WindowOpen` or `WindowOpenChild`. The target window is selectable via `WindowSelect` or `IGrSelect`. The latter routine also allows graphics to be drawn to a dialog field instead of to a window.

The graphics co-ordinate system is fully user definable and is controlled by the `IGrUnits` routine. It remains the same regardless of the type of target drawable.

Graphics drawn to a window must normally be maintained by the calling program. In other words, it is possible for screen graphics to be erased if an overlapping window or dialog is moved/closed. The calling program must process Expose and Resize messages to identify this situation.

Legacy graphics code written for the Lahey Video Graphics Library (PLOT, PLOTS, etc.) can be relinked with W*interacter* via the LVGL emulation interface in `lvgl.f90`. See the "Graphics Interfaces" section in the on-line help file for further details.

OpenGL graphics are also supported. These can be displayed in any W*interacter* window. See `WglSelect` and the OpenGL section under "Graphics Interfaces" in the on-line help file.

Color

Several W*interacter* routines accept color arguments. All routines which accept RGB (Red, Green, Blue) color arguments encode such color values in a single integer using the formula :

*Red + Green*256 + Blue*256*256*

where each of the Red, Green and Blue components are 8-bit values in the range 0-255. Hence these RGB values are also commonly referred to as "24-bit" color values.

The `WRGB` function can be used to construct a 24-bit color value and `WRGBsplit` performs the opposite conversion. Eight symbolic names are also pre-defined in the `WINTERACTER` module for the 8 primary colors: RGB_BLACK, RGB_BLUE, RGB_RED, RGB_MAGENTA, RGB_GREEN, RGB_CYAN, RGB_YELLOW and RGB_WHITE.

A Worked Example

This section explains how to write a simple W*interacter* program which uses a small but typical selection of the facilities available, including message handling, common dialogs and graphics. The short program is built up step by step, with newly introduced statements highlighted at each stage by a '*' in the right hand margin. A copy of the complete program along with its associated resource script can be found in the W*i*SK demos `example` directory.

The first thing to do in any W*interacter* program is to initialize the library by calling `WInitialise`. This must be followed by a call to `WindowOpen` to open a root window and initialize the graphics routines. To terminate screen processing, `WindowClose` must be called. A minimal W*interacter* program therefore looks like this:

```
PROGRAM WISK_EXAMPLE                                                     *
USE WINTERACTER                                                          *
IMPLICIT NONE                                                            *
CALL WInitialise()                 ! Initialize Winteracter              *
CALL WindowOpen(MENUID=IDR_MENU1, & ! Open root window                   *
                TITLE='Example Program')                                 *
CALL WindowClose()                 ! Remove program window               *
STOP                               ! Required by Elf90 only              *
END PROGRAM WISK_EXAMPLE                                                 *
```

The program initializes the library then fills a data structure with a description of the root window which is to be opened. `WindowOpen` is then called to open that window. Note that the program `USE`'s the `WINTERACTER` module. In this particular example it will define the `WIN_STYLE` data type, the symbolic names assigned to the `FLAGS` element of the window type argument and the interface blocks for each of the called routines. As noted earlier in this chapter, use of this module is *required*.

So far, all this program will do is open a root window with no menu then immediately close it again. Let's assume we want the program to read some time series data from a file and plot it as a simple line graph. The first task is to introduce some message handling so we can add 'Open' and 'Exit' options to the program. This will allow the user to select the data file to plot and to exit via a program menu option (though the System menu can be used for the same purpose, where enabled).

The following expanded example assumes that a resource file is supplied which defines a menu consisting of 'Open' and 'Exit' options.

```
PROGRAM WISK_EXAMPLE
USE WINTERACTER
IMPLICIT NONE
INTEGER, PARAMETER :: IDR_MENU1  = 30001                                 *
INTEGER, PARAMETER :: ID_OPEN    = 40001                                 *
INTEGER, PARAMETER :: ID_EXIT    = 40002                                 *
TYPE(WIN_MESSAGE)  :: MESSAGE                                            *
INTEGER            :: ITYPE                                              *
CALL WInitialise()                ! Initialize Winteracter
CALL WindowOpen(MENUID=IDR_MENU1, & ! Open root window
                TITLE='Example Program')
DO                                ! Loop until user terminates        *
  CALL WMessage(ITYPE, MESSAGE)                                          *
  SELECT CASE (ITYPE)                                                    *
    CASE (MenuSelect)             ! Menu item selected                *
      SELECT CASE (MESSAGE%VALUE1)                                       *
        CASE (ID_OPEN)            ! Select file to plot               *
          CONTINUE                ! We will load file here            *
        CASE (ID_EXIT)            ! Exit program (menu option)        *
          EXIT                                                           *
      END SELECT                                                         *
    CASE (CloseRequest)           ! Exit program (e.g. Alt/F4)        *
      EXIT                                                               *
  END SELECT                                                             *
END DO                                                                   *
CALL WindowClose()                ! Remove program window
STOP                              ! Required by Elf90 only
END PROGRAM WISK_EXAMPLE
```

Three parameters are now defined which identify the root menu and the two options which it will contain. Normally such PARAMETER statements would be stored in a module generated automatically by the resource editor, but they are shown as part of this program for the sake of clarity. The identifier of the root menu is now specified as part of the root window description, ensuring that Windows will automatically attach that menu to the window.

The main addition to the example program is the introduction of the DO loop which processes Windows messages. It loops continuously until the user selects Exit from the program menu or closes the window via the title bar controls.

The message loop also allows for the user having selected the 'Open' option from the root menu. We will now expand the program to allow a data file to be selected via a common dialog. Data will then be read from the file ready for plotting.

```
PROGRAM WISK_EXAMPLE
USE WINTERACTER
IMPLICIT NONE
INTEGER, PARAMETER  :: IDR_MENU1  = 30001
INTEGER, PARAMETER  :: ID_OPEN    = 40001
INTEGER, PARAMETER  :: ID_EXIT    = 40002
TYPE(WIN_MESSAGE)   :: MESSAGE
INTEGER             :: ITYPE, NVALUE, I                                    *
CHARACTER(LEN=255)  :: FNAME                                               *
REAL, DIMENSION(50) :: VALUES                                              *
CALL WInitialise()                 ! Initialize Winteracter
CALL WindowOpen(MENUID=IDR_MENU1, & ! Open root window
                TITLE=’Example Program’)
FNAME  = 'example.dat'                                                     *
NVALUE = 0                                                                 *
DO                                 ! Loop until user terminates
  CALL WMessage(ITYPE, MESSAGE)
  SELECT CASE (ITYPE)
    CASE (MenuSelect)              ! Menu item selected
      SELECT CASE (MESSAGE%VALUE1)
        CASE (ID_OPEN)             ! Select file to plot
          CALL WSelectFile('Data File|*.dat|', &                           *
                           PromptOn,FNAME,'Load Data')                     *
          IF (WInfoDialog(ExitButtonCommon)==CommonOpen) THEN              *
              OPEN(20,FILE=FNAME,STATUS='OLD')                             *
              READ(20,*)  NVALUE                                           *
              READ(20,*) (VALUES(I),I=1,NVALUE)                            *
              CLOSE(20)                                                    *
          ENDIF                                                            *
        CASE (ID_EXIT)             ! Exit program (menu option)
          EXIT
      END SELECT
    CASE (CloseRequest)            ! Exit program (e.g. Alt/F4)
      EXIT
  END SELECT
END DO
CALL WindowClose()                 ! Remove program window
STOP                               ! Required by Elf90 only
END PROGRAM WISK_EXAMPLE
```

Our program now defines a values array and a filename variable. A common dialog is used to select the input file (a suitable `example.dat` file is supplied in the W*i*SK demos `example` directory).

If the user confirms their file selection (i.e. they don't click on Cancel or press Escape) the name of the selected file is returned in the `FNAME` variable. Data is then read from the chosen file. Note that the file handling contains no error processing, to keep the example simple.

We are now ready to plot the data. A separate routine will be introduced which uses W*interacter*'s graphics routines.

```
PROGRAM WISK_EXAMPLE
USE WINTERACTER
IMPLICIT NONE
INTERFACE                                                                *
    SUBROUTINE DrawGraph(VALUES,NVALUE)                                  *
      IMPLICIT NONE                                                      *
      REAL   , INTENT (IN), DIMENSION(:) :: VALUES                       *
      INTEGER, INTENT (IN)               :: NVALUE                       *
    END SUBROUTINE DrawGraph                                             *
END INTERFACE                                                            *
INTEGER, PARAMETER  :: IDR_MENU1  = 30001
INTEGER, PARAMETER  :: ID_OPEN    = 40001
INTEGER, PARAMETER  :: ID_EXIT    = 40002
INTEGER, PARAMETER  :: ID_STRING1 = 50001
TYPE(WIN_MESSAGE)   :: MESSAGE
INTEGER             :: ITYPE, NVALUE, I
CHARACTER(LEN=255)  :: FNAME
REAL, DIMENSION(50) :: VALUES
CALL WInitialise()                 ! Initialize Winteracter
CALL WindowOpen(MENUID=IDR_MENU1, & ! Open root window
                TITLE='Example Program')
FNAME  = 'example.dat'
NVALUE = 0
DO                                 ! Loop until user terminates
  CALL WMessage(ITYPE, MESSAGE)
  SELECT CASE (ITYPE)
    CASE (MenuSelect)              ! Menu item selected
      SELECT CASE (MESSAGE%VALUE1)
        CASE (ID_OPEN)             ! Select file to plot
          CALL WSelectFile('Data File|*.dat|', &
                           PromptOn,FNAME,'Load Data')
          IF (WInfoDialog(ExitButtonCommon)==CommonOpen) THEN
              OPEN(20,FILE=FNAME,STATUS='OLD')
              READ(20,*)  NVALUE
              READ(20,*) (VALUES(I),I=1,NVALUE)
              CLOSE(20)
              CALL DrawGraph(VALUES,NVALUE)                              *
          ENDIF
        CASE (ID_EXIT)             ! Exit program (menu option)
          EXIT
      END SELECT
    CASE (Expose,Resize)           ! Need to redraw picture              *
      CALL DrawGraph(VALUES,NVALUE)                                      *
    CASE (CloseRequest)            ! Exit program (e.g. Alt/F4)
      EXIT
  END SELECT
END DO
CALL WindowClose()                 ! Remove program window
STOP                               ! Required by Elf90 only
END PROGRAM WISK_EXAMPLE
```

```
SUBROUTINE DrawGraph(YVALUE,NVALUE) ! Draw graph                   *
USE  WINTERACTER                                                   *
IMPLICIT NONE                                                      *
REAL,    INTENT(IN), DIMENSION(:) :: YVALUE                        *
INTEGER, INTENT(IN)               :: NVALUE                        *
!                                                                  *
REAL              :: XMIN,XMAX,YMIN,YMAX,XLEN,YLEN,XPOS            *
INTEGER           :: IX                                            *
!    Calculate X and Y ranges                                      *
XMIN = 1.0                                                         *
XMAX = REAL(NVALUE)                                                *
YMIN = MINVAL(YVALUE)                                              *
YMAX = MAXVAL(YVALUE)                                              *
XLEN = XMAX - XMIN                                                 *
YLEN = YMAX - YMIN                                                 *
CALL IGrUnits(XMIN-0.1*XLEN,YMIN-0.1*YLEN, &                       *
              XMAX+0.1*XLEN,YMAX+0.1*YLEN)                         *
!  Draw simple axes                                                *
CALL IGrMoveTo(XMIN,YMAX)                                          *
CALL IGrLineTo(XMIN,YMIN)                                          *
CALL IGrLineTo(XMAX,YMIN)                                          *
!  Draw line graph                                                 *
CALL IGrMoveTo(XMIN,YMAX)                                          *
DO IX = 2,NVALUE                                                   *
  XPOS = XMIN + XLEN*REAL(IX-1)/REAL(NVALUE-1)                     *
  CALL IGrLineTo(XPOS,YVALUE(IX))                                  *
END DO                                                             *
RETURN                                                             *
END SUBROUTINE DrawGraph                                           *
```

Minimum X and Y values are assigned. `IGrUnits` is then used to define a co-ordinate system which leaves a border around the area in which the graph will be plotted. A simple L-shaped axis is plotted, followed by the data itself. `IGrLineTo` is effectively a Pen-Down and draw operation, so the `DO` loop creates a connected line graph. The graph will also be redrawn if the window size changes or the window becomes partly or wholly exposed.

And finally, to add some annotation to the x axis:

```
    SUBROUTINE DrawGraph(YVALUE,NVALUE)  ! Draw graph
    USE  WINTERACTER
    IMPLICIT NONE
    REAL,    INTENT(IN), DIMENSION(:) :: YVALUE
    INTEGER, INTENT(IN)               :: NVALUE
    !
    CHARACTER (LEN=3) :: STR                                              *
    REAL              :: XMIN,XMAX,YMIN,YMAX,XLEN,YLEN,XPOS
    INTEGER           :: IX,ISTART                                        *
    !    Calculate X and Y ranges
    XMIN = 1.0
    XMAX = REAL(NVALUE)
    YMIN = MINVAL(YVALUE)
    YMAX = MAXVAL(YVALUE)
    XLEN = XMAX - XMIN
    YLEN = YMAX - YMIN
    CALL IGrUnits(XMIN-0.1*XLEN,YMIN-0.1*YLEN, &
                  XMAX+0.1*XLEN,YMAX+0.1*YLEN)
    !  Draw simple axes
    CALL IGrMoveTo(XMIN,YMAX)
    CALL IGrLineTo(XMIN,YMIN)
    CALL IGrLineTo(XMAX,YMIN)
    !  Draw line graph
    CALL IGrMoveTo(XMIN,YMAX)
    DO IX = 2,NVALUE
      XPOS = XMIN + XLEN*REAL(IX-1)/REAL(NVALUE-1)
      CALL IGrLineTo(XPOS,YVALUE(IX))
    END DO
    ! Add annotation to X axis                                            *
    CALL WGrTextOrientation(AlignCentre)                                  *
    DO IX = 5,NVALUE,5                                                    *
      CALL IGrMoveTo(REAL(IX),YMIN)                                       *
      CALL IGrLineTo(REAL(IX),YMIN-YLEN*0.025)                            *
      CALL IntegerToString(IX,STR,'(I3)')                                 *
      ISTART = ILocateChar(STR)                                           *
      CALL WGrTextString(REAL(IX),YMIN-YLEN*0.06,STR(ISTART:))            *
    END DO                                                                *
    RETURN
  END SUBROUTINE DrawGraph
```

Labels and tick marks are added to the x axis using center justified text. For more examples of how to use W*interacter* see the various other W*i*SK demonstration programs.

Application W*izard*

To create a new W*interacter* program, you may wish to consider using the W*i*SK Application W*izard* (`wiskwiz`). This creates a substantial W*interacter* starter application tailored to your requirements.

The W*izard* leads you through a series of 5 dialogs which ask simple questions about the type of application you wish to create. Enter the name of the project and the directory which will hold the application files. Use the Back/Next buttons to move to/fro between the W*izard* dialogs, describing the required appearance of your application and the basic options it is to offer. A preview field will show a mimic of the type of application the W*izard* will create based on your selections. Press Finish when you are ready to generate the files for your application. In fact, you can press Finish at any time, even if you have not worked through all five dialogs (the W*izard* will just fill in a set of default selections). A confirmation window will appear summarising your choices, after which the project files will be created, consisting:

- Fortran 90 source code for the new application.
- A resource script describing the new application’s menus, dialogs, etc.
- A module or include file (selectable) defining resource file identifiers.
- A program icon file.

The generated program will be ready for immediate compilation.

4 Resource Editor

The main W*interacter* user interface design tool is the resource editor. This allows you to create, edit and maintain resource scripts which define dialogs, menus, icons, bitmaps and cursors. The resource editor incorporates dialog, menu and image editors in a single integrated program.

When a resource is loaded, it can be navigated via the Resources window which displays a list of all of the dialogs, menus, etc. in that resource. Click on an item and press Edit to display the required resource component. Double clicking has the same effect. The Delete button in the same window can be used to remove components from a resource script.

This chapter briefly summarises the menus which are common to all parts of the resource editor. The subsequent chapters entitled *Menus*, *Dialogs* and *Icons/Bitmaps/Cursors* provide more information on how to build various types of user interface components using the resource editor. You will also find considerable additional information about the resource editor in its on-line help, via Help→Contents.

File Menu

A new resource script can be created by selecting File→New. Alternatively, an existing resource script can be selected via File→Open or the file name can be specified on the resource editor's command line. The latter option ensures that the Windows version of the editor supports invocation via drag-and-drop or via a file-type association (see 'View→Options→File Types' in Explorer).

The File→Save and File→Save As options save the current resource to disk. When editing a resource script, these options save both the updated script and the associated symbol header file (i.e. the Fortran module or include file which defines parameters for all of the identifiers used in the resource script). The resulting resource script will require external recompilation using the resource compiler as described in the earlier *Building a WiSK Program* chapter.

Once a resource has been loaded, additional resource import/export options also become available on the File menu.

Edit Menu

A standard set of Cut/Copy/Paste/Delete options are available on the Edit menu. The exact meaning of these options depends on which sub-editor is currently active, but typically they operate on resource sub-components such as dialog fields or menu items.

Resource Menu

The Resource menu operates on complete resource components such as a menu or a dialog, offering Add, Copy and Properties options. Use this menu to introduce new components to your resource file or to view/amend the properties of the current component.

Settings Menu

Selected features of the editor's behaviour can be customised, via Settings→Preferences. The Resource List options are saved in the resource editor's initialisation file (`resedit.ini`) and reactivated each time the editor is invoked. The remaining preferences are written in the resource file, allowing the symbol header file name, base identifiers, etc. to be selected on a per resource basis.

View Menu

When a resource script is loaded, this menu provides options to view the identifiers which is uses.

The Identifier Names and Values option shows all of the identifiers used by the entire resource and allows them to be edited

The Used Identifiers option shows only the identifiers used by the current resource component (e.g. the current dialog) and allows a particular sub-component to be selected as the 'current' item, via the Select button.

5 Menus

Overview

Menus are the main method by which most programs will determine the next action to take. They consist of various options which can be chosen by the user using the keyboard or mouse. When an option is chosen the program will take some action related to the chosen option, such as displaying a dialog or plotting a graph. Menu layout is defined in your program's resource script, which is created using the supplied resource editor. Details of how to use this program are given later in this chapter and in online help.

Every menu and menu option used in a given program must have a unique identifier, as set in the resource file. A Fortran module or include file should normally be used to specify PARAMETER values for these dialog/field identifiers. The resource editor creates such a module or include file automatically. This file contains identifier definitions for all resource types. The values in this file will be updated whenever your resource script is saved by the resource editor.

Up to 50 menu definitions definitions can be built into a program executable, as part of the program resource. The main menu for a window is normally specified when opening a window using `WindowOpen` or `WindowOpenChild`. This can be changed or removed later using `WMenu`. Floating menus are displayed by calling `WMenuFloating`, usually in reponse to a right mouse click. Normally only the root window will have a main menu.

When the user selects a menu option a `MenuSelect` message will be reported via `WMessage`. No message will be reported for items which lead to sub-menus or if the user closes the menu without selecting an option. The `MenuSelect` message does not report which type of menu the message came from or what method the user used to select it. Programs should not be concerned with how an option was selected, only that it was selected.

Menu Types

Two types of menu are supported by W*i*SK:

Main Menus

A main menu is attached to the top of a window. Most, if not all, of the menu options in your program should be available via the main menu. The top level of this type of menu is always displayed. The options at this level normally lead to sub-menus. It is possible to have items at the top level which do not lead to sub-menus, however this is unusual in a Windows or Motif application. This type of menu can not be used with a child window which lies within its parent window.

Floating Menus

This type of menu can be displayed at any point on the screen. It is normaly displayed at the position of the mouse cursor in response to a right mouse click. Normally this type of menu will only have a small number of options and sub-menus. Often it will consist of a sub-set of the options available via the main menu. Where a floating menu item performs the same function as a main menu item it should be given the same identifier. In addition to simplifying your menu code this will also cause the states of items in the floating menu to match those of items in the main menu.

Menu Item Types

Three types of item exist in menus:

Selectable Options

These are the options which are used to select the functions available in your program. They consist of a text string describing the function which the item performs. When one of these options is chosen by the user a `MenuSelect` message will be reported via `WMessage`.

Popup Options

These options lead to sub-menus. Selection of this type of option is not reported via `WMessage`. Normally most of the items at the top level of a main menu will be of this type.

Separators

Separators are used to split the options in a menu or sub-menu into logical groups. They have no functional effect on your program, but do improve the appearance and legibility of a menu. Separators do not have an identifier.

Menu Item States

Menu items can be greyed out to prevent them from being selected. They can also be checked to indicate that they have been selected. This is typically used with menu items which toggle the state of an option. Checked items in main and floating menus display a tick next to the item text. The initial state for an item is set in the resource script using the resource editor's

Menu Item Properties dialog (double click on the relevant menu item in the resource editor's menu mimic to view this dialog). The state can be updated in your program using the `WMenuSetState` routine. Items at the top level of a main menu can not be checked.

Menu Help

Brief help on the purpose of a menu item can be provided by means of associated text displayed in the status bar of the window for both main and floating menus. This help text is defined using the resource editor and is displayed automatically when the user highlights an option. No code is necessary to display this help text. However you must specify the presence of a status bar when opening the window. Any previous status bar contents will be restored automatically when the menu is not active.

Keyboard Access to Menus

In addition to selecting menu items using a mouse they can also be accessed via the keyboard. At the most basic level all main menu items can be accessed by using the Alt key to activate the menu then using the cursor keys to navigate the menu. The initial letters of menu items can be used to provide quicker access to the items in a sub-menu. Where multiple items in a sub-menu start with the same letter, the letter to use for an option can be changed by entering an & character into the caption string in the resource editor. It is advisable to specify the letter to be used in this way even when using the initial letter since the & character also causes the letter to be underlined. This is the expected behaviour for a Windows or Motif program.

In addition keyboard shortcuts, known as accelerators, can also be defined to provide direct access to specific main menu options. Such accelerator keys are defined using the resource editor. Defining a keyboard shortcut as an accelerator requires no additional programming effort since those key sequences are then reported as `MenuSelect` (rather than `KeyDown`) messages by `WMessage`. When defining accelerators it is useful to use commonly adopted conventions where possible. For example if your program has a 'New' option it should use Ctrl+N for its accelerator. It is also advisable to avoid using commonly used accelerator combinations for options other than their normal purpose. If the keyboard accelerators for a menu are to be processed while a dialog is active you should avoid accelerators which conflict with standard dialog keystrokes. See the Dialogs chapter for more details.

Creating and Modifying Menus

Menus are created and maintained using the resource editor. See the earlier Resource Editor chapter. You should also refer to the resource editor's on-line help, available via Help→Contents.

A new menu can be created either by starting a new resource script or by adding a menu to an existing resource script. To start a new resource script select File→New, then select Resource File. Next, select Resource→Add and choose 'Menu' to add a new menu structure. Alternatively, open an existing resource script using File→Open. If the resource contains any existing menus, they will be listed in the Resources window. Double click on a menu in the Resources window to display it. If the resource script contains no menus (for example it contains only dialog definitions) or to start a new menu, select Resource→Add as described above,

Adding a new menu will display the Menu Properties dialog. This allows the identifier of the menu to be specified. Selecting OK will create a new menu consisting of a single option with an 'Empty' caption. The individual menu options can now be created, see below. To change the menu identifier after creation select Resource→Properties.

When a resource script contains several menu defintions you can choose which menu to edit via the Resources window. Alternatively, whenever a menu is selected as the current resource type, you can cycle though the available menus using the Page Up and Page Down keys. Alt+1 to Alt+0 can also be used to select among the first 10 menus in this case.

If a menu is no longer required it can be removed using the Delete button in the Resources window.

Adding and Modifying Menu Items

To add a menu item use either the Insert After or Insert Before options on the Edit menu. These options are also available on the right-click menu or via keyboard shortcuts. Menu items are created with a default caption of 'Empty' and automatically given a default identifier.

When a menu item is first created it becomes the current item. Different items can be selected via the mouse or cursor keys. To select an item in a sub-menu which is not displayed you must first select the item which leads to that sub-menu. The sub-menu will then be displayed. To select an item using its identifier name select View→Used Identifiers. This will display a list of the items in the current menu. The Select button will select the first item with the highlighted identifier.

The properties of the selected item can be changed via the Menu Item Properties dialog. This can be accessed by double clicking on an item (or select Properties on the Edit or right-click menus). Alt+Enter also brings up the same dialog. This dialog is used to enter the selected item's identifier, caption, status bar prompt (optional), type and initial state. Refer to the online help for full details of this dialog.

To assign a keyboard accelerator to the selected item select Accelerator from the Edit or right-click menus or press Tab. Full details of this dialog are included in online help. A description of the chosen accelerator will be added to the caption for the item, if not already present.

The selected item can be copied or deleted using the options available on the Edit menu. The usual keyboard shortcuts, Ctrl+C, Ctrl+X, etc. are also available for these options.

Tutorial - Creating a Menu

This brief tutorial will guide you through the basic steps of creating a menu, based on the methods described in the preceding sections:

1. Load the resource editor and select File→New, then select Resource→Add and choose 'Menu'. You will be prompted for the identifier of the new menu. For now just accept the default value. When you click OK a menu mimic, i.e. an emulation of an application program's menu which allows you to visualise how the menu will appear in your W*interacter* program, will be displayed. Initially it has one item (labelled Empty) which is highlighted. Double click on this item to display its properties.

2. The properties dialog is used to alter the settings for each individual item. First, set the text that actually appears on the menu bar. This is done via the Caption field. Delete the current contents of the field and then type &File into the field. The ampersand (&) character is used to set which character within the string is underlined. This key can then be combined with the Alt key to select the menu item via the keyboard.

3. Click on the Popup check box (so that a tick appears). This will attach a pull down menu (or child menu) to the current item. Press OK. You should notice two things. First the highlighted item will now say File. Secondly, underneath it is a child menu with a single option (labelled Empty).

4. Double click on the new child item to display the properties dialog. Type &Option 1 into the caption box. As this item will be used to input a user action we must give it an identifier. This identifier will be reported via `WMessage` in the message loop at run-time, when the item is selected. By default an identifier of ID_ITEM2 is given to this object but let’s change this to something more meaningful. Type ID_OPTION1 into the item ID field. Press OK.

5. The display will now be updated. To add another item to the child menu select Edit →Insert After or press Ctrl+A. Remember that the Edit menu can also be accessed by pressing the right mouse button. You should now have a new item at the bottom of the child menu (labelled Empty) which will be highlighted.

6. Double click on the new item. Type E&xit into the caption box. Type ID_EXIT into the item ID box. Click the OK button. Your first menu is now finished.

7. To save your resource, select File→Save. This will display the standard file selector. Enter a file name (e.g. `resource.rc`) and click OK.

You now have a resource script which can be compiled and linked with your W*interacter* program code. See the earlier *Building a WiSK Program* chapter for details on how to link this resource script with your program. Refer to the subroutine reference section for details on the routines for manipulating menus and processing selections. Other resources (e.g. dialogs) can be added to the same resource script.

6 Dialogs

Overview

A dialog is a set of associated data entry fields. Normally a dialog will be specific to a particular application and you will define its layout in your program's resource script using the resource editor. Details of how to use this program are given later in this chapter and in online help. Alternatively certain dialogs are commonly required by many different programs. W*interacter* provides access to these common dialogs via the routines in the Common Dialogs section in the subroutine reference section. The remainder of this chapter is concerned with program defined dialogs.

Every dialog and field used in a given program must have a unique identifier, as set in the resource file. A Fortran module or include file should normally be used to specify PARAMETER values for these dialog/field identifiers. The resource editor will create such a module or include file automatically. This file contains identifier definitions for all resource types. The values in this file will be updated whenever your resource script is saved by the resource editor.

Up to 400 dialog descriptions can be built into a program executable, as part of the program resource. The Dialog Manager operates by selectively loading one or more of these dialogs from the program resource (see `WDialogLoad`), possibly modifying the dialog, then displaying it (see `WDialogShow`). When no longer required, a dialog can be unloaded (see `WDialogUnload`). The same dialog can be loaded and unloaded as many times as required. Alternatively, it can be loaded just once then repeatedly displayed and hidden (see `WDialogHide`).

When more than one dialog is loaded, the Dialog Manager uses the concept of the 'current' dialog. This is simply the dialog which most of the other dialog management routines will operate on. This can be set explicitly via `WDialogSelect`, but is also set implicitly by `WDialogLoad`. Opening a combined window and dialog using `WindowOpen` or `WindowOpenChild` also selects the combined dialog as the current dialog.

The initial contents of each field can be defined in the resource file, using the resource editor. These will be the initial values each time `WDialogLoad` is called. While loaded, the contents of the dialog can be updated via the various `WDialogPutXXX` routines in the DM(2) group. See the subroutine reference section for details. The values of dialog fields can be interrogated, (both before and after user data entry) via the corresponding `WDialogGetXXX` routines in the same group. For example, the contents of a string field can be assigned via `WDialogPutString` and retrieved via `WDialogGetString`.

When a user has finished entering data in a dialog they will normally press a termination button (e.g. OK, Cancel, etc.). This will cause a modal dialog to be terminated immediately, with termination information reported via `WInfoDialog`. However, a modeless or semi-modeless dialog will remain on-screen, with the button press reported as a `PushButton` message via `WMessage`. The dialog remains on screen until explicitly removed by `WDialogHide` or `WDialogUnload`.

The dialog can also be closed or `PushButton` messages reported for other user actions. The following actions will cause this:

- Pressing return in a non-push button field will act as if the default push button was pressed. If there is no default push button in the current dialog W*interacter* will act as if a default push button with an identifier of IDOK was present. A push button can be made the default button using its Style Properties dialog in the resource editor.
- Pressing Esc, Alt+F4 or closing the dialog via its title bar controls will act as if a button with an identifier of IDCANCEL was pressed. For this reason you should normally use this identifier for the Cancel button of a dialog. This will simplify your code by ensuring you only have to check for one cancel value.
- Pressing F1 in a modeless or semi-modeless dialog will report a `PushButton` message with an identifier of IDHELP. Using this value for any actual Help button is recommended since it will simplify your code.

The IDOK, IDCANCEL and IDHELP identifier values are pre-defined in the `WINTERACTER` module.

Dialog Types

When a dialog is shown by `WDialogShow` its type must be specified. This controls how the dialog interacts with the user and your program. Three types of dialog are supported :

Modal

A modal dialog blocks data entry or option selection via other dialogs or menus belonging to the current program. The button used to close the dialog is available via `WInfoDialog`. Such dialogs are easier to manage from a programming viewpoint because display, user-entry and termination all occur as a single operation (in a call to `WDialogShow`). They also eliminate any need to process Resize or Expose messages while the dialog is displayed.

Modal dialogs are best used where access to the rest of the program is not required and the only buttons are OK and Cancel buttons to confirm or abandon data entry.

Modeless

A modeless dialog remains on screen while a program continues to run. Button presses are reported via `WMessage`, as `PushButton` messages. The dialog remains visible until explicitly removed by the calling program. Several modeless dialogs can be active simultaneously. Because the program continues to run modeless dialogs permit more sophisticated processing of the displayed dialog than is possible when using a modal dialog. For example a combination of `FieldChanged` messages and the `WDialogFieldState` routine can be used to selectively grey out or enable fields depending on the options chosen in a dialog.

Modeless dialogs should be used when simultaneous access to other dialogs or windows is required.

Semi-Modeless

A semi-modeless dialog is a useful hybrid of the previous two types. Such a dialog appears modeless to the program, but modal to the user. In other words, control returns to the program as soon as the dialog is displayed by `WDialogShow` (allowing message processing via `WMessage`, for example), but input to other dialogs or program menus is blocked. Like a modeless dialog, a semi-modeless dialog remains on-screen until explicitly removed by the calling program. Multiple semi-modeless dialogs can be stacked, allowing the use of 'Options' or 'Advanced' buttons to activate sub-dialogs.

Semi-modeless dialogs should be used when the additional facilities offered by a modeless dialog are required but access to other windows and dialogs needs to be prevented.

Under Windows, all dialogs are either 'pop-up' or 'child' dialogs (determined in the resource file, via Dialog Properties in the resource editor). A pop-up dialog can be moved to any position on the screen either inside or outside of the application window. Child dialogs are restricted to the root window. Child dialogs must be modeless. 'Sub Component' dialogs are a special type of Child dialog. While these can be loaded and displayed by the routines in the Dialog Manager chapter of the subroutine reference section they are primarily intended to be combined with a window by `WindowOpen`/`WindowOpenChild`. Under X Windows, all dialogs which are not combined with a window are 'pop-up' dialogs.

Field Types

Whichever dialog type is used, it can consist of the following field/control types :

Strings

String fields allow the user to enter character data into a dialog. Either single-line strings or multi-line fields are available. In read-only mode, string fields are also useful for the output of character data which the user can then copy to other applications via the clipboard.

Menus

Menu fields allow the user to choose from a list of options. Available types are:

Simple Combo Box : In addition to a list of options this type also provides a string field into which the user may type any value.

Drop Down Combo Box : This also consists of a string field and list of options. However the list is hidden until the user displays it using the button at the right of the string field. This type of menu can be used as an enhanced string field which allows the user to choose from a list of standard values.

Drop Down List Combo Box : This is similar to the Drop Down Combo Box except that only the listed options may be chosen. This type of menu is useful where space is limited.

List Box : This is a simple, permanently visible list. This menu type can optionally be used for the selection of multiple options. List boxes are also useful for displaying scrolling output, e.g. a list of messages where the user may wish to scroll back through earlier messages.

Check Boxes

A check box provides a convenient way for the user to indicate Yes/No choices. They consist of a label and an on/off indicator.

Radio Buttons

Radio buttons are similar in appearance to checkboxes. However they are used in groups to select between a small number of mutually exclusive options. When a radio button is selected all others in the same group are cleared. The grouping of radio buttons is determined by the 'Group' flags in the 'General Properties' dialogs in the resource editor.

Push Buttons

These are buttons, such as 'OK' and 'Cancel', which the user can use to close the dialog. In modeless and semi-modeless dialogs push buttons can also be used to access other dialogs, e.g. an 'Advanced' or 'Options' button.

Progress Bars

Progress Bars are output only fields which show an integer value as a bar. They are typically used to visually indicate the progress of a time consuming task.

In addition to these 'functional' fields a dialog can also include various 'decorations'. These are defined within the resource script, but are display-only fields:

Labels

Label fields are used to label most other types of fields. They should not be confused with string fields since they are different in appearance and do not provide any input or cut/paste facilities.

Group Boxes

Group boxes consist of a box and associated label and are used to visually group other fields. Group boxes have no functional effect on other fields. In particular, they do not affect the behaviour of radio button groups.

Pictures / Frames

These allow bitmaps and icons to be displayed in dialogs. They can also be used to draw unlabelled frames or filled rectangles.The rectangle variation is particularly suited to displaying program generated graphics in a dialog. See `IGrSelect`.

Keyboard Processing in Dialogs

When a dialog is displayed and has the input focus it will normally process all keystrokes automatically. For this reason `KeyDown` message are not reported while a dialog has the focus. A dialog will process a keystroke in one of several ways:

- It will treat the keystoke as data to be entered into the current editable field. This will not report a message to the program.
- The Tab and Shift-Tab keys will move to the next or previous fields. This can be reported via a `FieldChanged` message for a modeless or semi-modeless dialog.
- The keystoke will correspond to a dialog button and either close a modal dialog or report a `PushButton` message in a modeless or semi-modeless dialog, as documented in the introduction to this chapter. Note: There need not be an actual push button field corresponding to these actions. For example, the Escape key always generates a `PushButton` message with an IDCANCEL identifier.
- It will perform a field type dependent action, such as the space bar toggling the state of the current checkbox. Some of these actions can be reported as a `Field-Changed` message for a modeless or semi-modeless dialog. Other actions, such as displaying the drop-down list in a combo box, will not report a message.
- The keystroke is a program defined shortcut to a particular field. The exact action taken in this case depends on the field type. See later for details of how to create these shortcuts and their effects.
- The keystroke will be ignored because it has no meaning for the current field.

Keyboard shortcuts can be created for a particular field, such as an Apply button. This is most commonly done for push button, check box and radio-button fields. To implement such a shortcut prefix a letter in the field's caption with an ampersand (&). Normally the first letter of the field's caption will be chosen, except where this would cause a conflict with another field. When creating a dialog which will later be combined with a window (see `WindowOpen` and `WindowOpenChild`) you should also avoid choosing letters which will cause conflicts with the top level of the window's main menu. The prefixed letter will be underlined. Alt and the shortcut letter can then be used to access the field directly. Such shortcuts are not normally defined for the OK and Cancel buttons since keyboard equivalents of these actions already exist.

Keyboard shortcuts can also be used to move directly to an editable field, such as a string field. In this case the ampersand prefix and shortcut letter should be placed in the label or group box which labels this field. The fields should be re-ordered so that the field with the shortcut letter immediately precedes the editable field. No coding is needed to implement these shortcuts. They are handled automatically once defined using the resource editor.

Keyboard accelerators for menu items can optionally be enabled when a modeless dialog is displayed. Such accelerators are also always enabled when a dialog is combined with a window. If enabled, keystrokes which correspond to these accelerators are reported as

`MenuSelect` messages in the usual way. This facility will not normally be required with non-combined dialogs. It is most useful with permanently displayed child dialogs. Using this facility with a popup or temporarily displayed dialog would be highly unusual behaviour.

When creating menu accelerators which will be enabled when a dialog has the focus you should avoid using keystrokes which normally have a meaning in a dialog. Creating such an accelerator would disable the keystoke's usual function in a dialog. The following keystrokes should be avoided:

Table 1: Dialog Keystrokes

Keystroke	**Usual DIalog Function**
Unmodified Characters	Data entry
Tab/Shift Tab	Move between fields.
Cursor Keys	Move cursor within current enterable field Change option in current menu Move between fields in same group (if no other action defined)
Page Up/Page Down	Move cursor within current enterable field Change option in current menu
Home/End	Move cursor within current enterable field Change option in current menu
Shift+Cursor Keys Shift+Page Up/Down Shift+Home/End	Select characters in editable field Select range of options in current extended selection listbox
Ctrl+A	Select all characters in editable field
Space Bar	Toggle state of current checkbox Activate current push button Toggle state of current option in current mutiple or extended selection listbox
Enter	Activate current or default push button Insert new line into multi-line string (if enabled) Close open drop-down combo box
Esc	Cancel dialog (simulate Cancel button)
Alt+F4	Cancel dialog.Under X Windows other key sequences may perform this function, depending on the window manager being used
F4	Open/Close current drop down combo box

Table 1: Dialog Keystrokes

Keystroke	Usual DIalog Function
Alt + Letter	Program defined field shortcut, see above
Ctrl+X, Shift+Delete	Cut selection to clipboard
Ctrl+C, Ctrl+Insert	Copy selection to clipboard
Ctrl+V, Shift+Insert	Paste clipboard contents
Delete	Delete selection

Dialog Validation and FieldChanged Messages

The data entered into a dialog can be validated in several ways:

- Validation can be performed at the calling level by checking the values returned by the 'get' routines in group DM(2). See the subroutine reference section for details. If a modeless or semi-modeless dialog is used this can be done on the fly by checking for `FieldChanged` messages.
- If it is known in advance that selecting certain radio buttons or checkboxes would be invalid these fields can be disabled, using `WDialogFieldState`, to prevent such selections.

These methods can also be combined. For example if selection of a particular checkbox precludes the use of certain other fields in the same dialog then the `FieldChanged` message would be used to detect selection of the checkbox and `WDialogFieldState` used to disable the other fields.

`FieldChanged` messages fall into two distinct categories:

- Messages which report that the user has moved from one field to another. For these messages the *%value1* and *%value2* elements of the message structure will be different.
- Messages which report a change in state of a field with a well defined number of possible states, i.e. a check box, radio button or menu field. For these messages the *%value1* and *%value*2 elements will both refer to the changed field.

It is important that code to handle `FieldChanged` messages should process the correct messages for the task to be performed. In particular, be aware that some user actions actually result in two messages in quick succession. For example consider the case of the user using the mouse to toggle a checkbox which is not the current field. The first message reports the change of focus. This occurs when the user presses the mouse button. The second message reports the change of state of the checkbox. This is reported when the user releases the mouse button. To take action based on the new state of the checkbox when it changes you must ensure that you process the second of these messages. Specifically your code must include a test that *%value1* and *%value2* both report the same identifier. Processing the wrong message can cause various problems. For example in this particular case calling `WDialogGetCheckBox` after the first message will report the previous checkbox state, since it has not yet changed.

`FieldChanged` messages must be specifically enabled, using `WMessageEnable`. They are not reported by default.

Cut and Paste in Dialogs

String fields support cut and paste of their contents. These facilities are available via the usual Windows shortcuts, Ctrl+X, Ctrl+C, etc. or via the right-click shortcut menu.

Creating and Modifying Dialogs

Dialogs are created and maintained using the resource editor. See the earlier Resource Editor chapter. You should also refer to the resource editor's on-line help, available via Help→Contents.

A new dialog can be created either by starting a new resource script or by adding a dialog to an existing resource script. To start a new resource script select File→New, then select Resource File. Next, select Resource→Add and select 'Dialog' to add a new dialog. Alternatively, open an existing resource script using File→Open. If the resource contains any existing dialogs, they will be listed in the Resources window under the Dialogs branch. The dialog to edit can be selected by double clicking on the appropriate identifier in the Resources window. If the resource script contains no dialogs (for example it contains only menu definitions) or to start a new dialog, select Resource→Add as described above,

Adding a new dialog will display the Dialog Properties dialog. This is used to determine basic features of the dialog such as its identifier, size and title. This dialog also determines whether the dialog appears outside or inside the root window or is to be used as a sub-component of another dialog or window. Refer to the online help for details on the individual options available in this dialog. The position of the dialog and whether it is modal, modeless or semi-modeless is chosen at run-time by the `WDialogShow` routine. Selecting OK will create and display an empty dialog with the chosen properties.

Once created the dialog is ready to have fields added. To modify its properties after creation, select Resource→Properties. Under Windows this dialog can also be displayed by right clicking on the dialog's title bar.

When a resource script contains several dialog definitions you can choose which dialog to edit via the Resources window. Alternatively, whenever a dialog is selected as the current resource type, you can cycle though the available dialogs using the Page Up and Page Down keys. Alt+1 to Alt+0 can also be used to select among the first 10 dialogs in this case.

If a dialog is no longer required it can be removed using the Delete button in the Resources window.

Creating and Modifying Fields

To create a field you must first select the type of field to be created. To do this either use the toolbar across the top of the main window or Field→Add. Tooltips are available for each of the buttons on the toolbar which indicate the type of field created. Once the type of field to create has been selected the field is created by left-clicking in the window representing the dialog. The field is created with its top-left corner at the clicked position. Subsequent fields of the same type can be created simply by further clicks. Alternatively a different field type can be chosen using the toolbar or menu. To prevent accidental creation of fields select the pointer button on the toolbar. This enters a select only mode.

When a field is first created it becomes the current field. This is indicated by a thickened border around the field. Any previously selected field is deselected. Different fields can be selected by clicking on them using the mouse. To select a group box field or the rectangle variant of the picture/frame field you should click on or near the border. Clicks in the center of these fields do not select the field to allow other fields to be created inside them. Alternatively pressing Tab/Shift-Tab will select the next/previous field. To select a field using its identifier name select View→Used Identifiers. This will display a list of the fields used in the current dialog. The Select button will select the field with the highlighted identfier. To select multiple fields hold down the shift key while clicking a field. Alternatively a group of fields can be chosen by dragging a rectangle around them using the mouse when the Select Only option is chosen via the toolbar or menu. When multiple fields are selected the last field selected has a different colored border. This is important with some of the field alignment options. To clear any current selection click in an empty area of the dialog when in Select Only mode.

Selected field(s) can be moved to a different position by dragging using the mouse. The mouse cursor will change to a four-headed arrow when positioned to move a field. This will snap to the grid unless it has been disabled using View→Grid. Alternatively the selected field(s) can be moved by a single dialog unit using the cursor keys or by 8 dialog units using shift and the cursor keys. When only a single field is selected it can be resized by dragging its border. The mouse cursor will change to a two-headed arrow when positioned to resize a field. Again this snaps to the grid if enabled.

There are also various options to align selected fields or make them the same size available via the toolbar at the left of the main window and on the Alignment sub-menu of the Field menu. The centring options center each of the selected fields individually. The options to align field edges align the appropriate edge of the selected fields with the last field selected. The sizing options make the appropriate dimension(s) the same as for the last field selected. Refer to the online help for full details of these options.

The selected fields can be copied or deleted using the options available on the Edit menu. The usual keyboard shortcuts (Ctrl+C, Ctrl+X, etc.) are also available.

When a single field is selected there are various dialogs available to modify its properties. These are accessed via the shortcut menu displayed by the right mouse button or via Field→ Properties. The General Properties dialog can also be displayed using Alt+Enter. Under Windows it can also be accessed by double-clicking a field. Refer to the on-line help for details of the specific dialogs used for each field type. However the following general notes apply for each option:

General

This dialog is used to control commonly available features of a field such as its identifier, position, size, initial contents and whether it is enabled or disabled.

Style

This dialog is used to control options affecting the appearance of a field or to enable optional features.

Border

This dialog controls the style of border, if any, drawn around the field.

Colour

This dialog allows the colors used by the field to be changed. However in general you should use the default field colors. This will give your application an appearance which is consistent with other programs. Overuse of color can make your program look very out of place in a graphical environment.

By default the order of fields is determined by the order in which they are created. This order determines the order in which the Tab key moves between fields and is also important in the grouping of radio buttons. Grouping of radio buttons will be dealt with in detail later. To change the ordering of fields select Dialog→Re-Order Fields. Once chosen simply click the fields in the desired order. To finish re-ordering fields take any other action, for example select Dialog→Test to check the new field order. Fields should be ordered so that pressing the Tab key moves through the fields in a logical order, starting at the top-left of the dialog. OK, Cancel and other buttons should normally be at the end of the tab order.

Radio Buttons and Field Grouping

Radio buttons are a useful method of choosing between a small number of mutually exclusive options. While they are created and modified in the same way as for any other field type, they are unique in that several need to be grouped together to be useful. To do this two things must be done:

- The fields should be consecutive in the field order. If the fields which form the group are created at the same time then this will happen automatically. However when extending a radio button group later it will be necessary to re-order the fields using Dialog→Re-Order Fields. Specifically you should click each field in the group in order. This also sets the order within the group for keyboard navigation.
- The Group flags in the General Properties dialog must be set correctly. This flag is used to indicate that a field is the start of a new group. Hence this flag should be set for the first radio button in the group, cleared for all other radio buttons in the same group and set for the first field, of whatever type, after the group. Again the logic in the resource editor is such that these flags will normally be set correctly when a new group of radio buttons is created. It will normally only be necessary to change these flags when extending an existing group.

In addition to arranging radio buttons into functional groups, grouping fields also affects keyboard navigation of the dialog. Where the cursor keys have no other meaning for a field, e.g. for check boxes, they will move between fields in the same group.

Tutorial - Creating a Dialog

This brief tutorial will guide you through the basic steps of creating a dialog, based on the methods described in the preceding sections:

1. Load the resource editor and select File→New, then select Resource→Add and choose 'Dialog'. You will be prompted for the properties of the new dialog. For now just accept the defaults. They can be changed later if necessary. When you click OK an empty window will appear. This represents your new dialog. The Field Insertion toolbar will appear at the top of the main window. This is used to add new fields to your dialog.

2. Now that we have a dialog we can create some fields inside it. First we will create the simplest type of field, a label. Click on the 'Label' button then click inside the window representing your dialog. Don't worry too much about exact positioning at this stage. The field is created with the default contents 'label'. Let's change this to something more meaningful. Right click on the field to bring up a small floating menu. Select General from this menu. This will display a dialog showing the general properties of the field. For now just move to the caption field and enter something more meaningful, then click OK.

3. Having changed the caption of the label field it is likely that it is no longer big enough to display the text. To fix this, change the field's size by dragging the border of the field just as if it was a program window. Dragging within the field will move the field to a new position. Notice that the field's size and position snaps to the grid.

4. If the next field we wanted to create was also a label then we could simply click in an empty region of the dialog again. However we will now create an entry field. Select the 'String' button from the toolbar and click somewhere to the right of your label. Notice that the thick border is removed from the label and placed around the string field, indicating that this is currently selected.

5. At this point we can test the dialog by selecting Dialog→Test. This will allow you to type into your newly created string field. Press Esc or Enter when done.

6. While this dialog worked, having to use the keyboard to close it is not normal for a graphical environment. Let's create OK and Cancel buttons. Do this by selecting the 'Push Button' toolbar button and clicking twice in the dialog window where you wish the buttons to appear. Notice that the captions on the buttons are automatically set to 'OK' and 'Cancel'. They are also assigned the standard identifiers IDOK and IDCANCEL. This is done for the first two buttons in every dialog. Also note that the first button (OK) was automatically made the default push button (as indicated by its different frame style). This will cause the Enter key to act as if this button was pressed. You may now wish to test your dialog again to see the improved effect

7. To save your resource, select File→Save. This will display the standard file selector. Enter a file name (e.g. resource.rc) and click OK.

You now have a resource script which can be compiled and linked with your W*interacter* program code. See the earlier *Building a WiSK Program* chapter for details on how to link this resource script with your program. Refer to the subroutine reference section for details on the routines for manipulating dialogs in your program. Other resources (e.g. menus) can be added to the same resource script.

7 Icons, Bitmaps and Cursors

W*interacter* programs can use icons, bitmaps and cursors defined via the resource script, in several ways:

Icons

Each program has an associated icon. For an existing icon file this can be selected in the resource editor using File→Import Image. Enable 'Set as program icon' in the subsequent dialog. A 'Set as program icon' checkbox is also available in the Image Propeties dialog when adding a new icon to a resource script via the Add→Resource option.

Icons can also be displayed in dialogs in various field types (see *Using Bitmaps and Icons in Dialogs* later in this chapter). Their main advantage over bitmaps is that they allow for transparent pixels. Icons can also be selectively displayed in a dialog at run time, via `WDialogPutImage`.

Bitmaps

Bitmaps can also be displayed in dialogs using similar mechanisms to icons (e.g. via `WDialogPutImage`). Unlike icons, bitmaps do not support transparent pixels and cannot be used as a program icon. The image editor can be used to create small bitmaps.

Cursors

In addition to various pre-defined cursors, `WCursorShape` allows selection of user defined cursors. (Note : User-defined cursors are only used by the Windows version of W*interacter*, but can be created by all versions of the image editor)

Images specified in the resource script will be incorporated into your executable. There is therefore no need to distribute the `.ico`, `.bmp` or `.cur` files with your application.

Image Editor

An image editor is provided as part of the W*interacter* resource editor which allows you to interactively create and edit icons, small bitmaps and cursors. All of these image types can be incorporated into your application via the resource file.

The image editor within the resource editor can be used in one of two ways:

- Image files can be edited directly, without associating that file with a particular resource script. An existing image file can be loaded via File→Open or by specifying its name on the resource editor's command line. The latter option ensures that the resource editor supports invocation via drag-and-drop or via a file-type association (View→Options→File Types in Windows Explorer). Alternatively, a new standalone image can be created via File→New.

- Image files which are associated with a resource file can be edited by loading that resource file, then selecting the icon, bitmap or cursor to edit from the Resources window. Alternatively, a new icon, bitmap or cursor can be added to a resource from scratch using Resource→Add. Using the built-in image editor in this way still creates a separate image file, but it also ensures that the file is associated with a resource script.

In addition to this chapter you should also refer to the resource editor's on-line help, available via Help→Contents. This contains more detailed information on each of the available options.

Supported Formats

The image editor can create icons, bitmaps and cursors at any size between 8x8 and 48x48 pixels, in either 16 or 256 colors. The image format is specified when the image is first created, or by using the Resource→Properties option. Bitmaps larger than 48x48 pixels will need to be created using an external program which can save files in Windows `.bmp` format. Once created these images can be added to your resource script using the File→Import Image.

Editing 256 color images requires a display which supports more than 256 colors.

When designing an icon for use as the program icon you should select a 32x32 pixel, 16 color icon. While any of the supported sizes can be used 32x32 pixels gives the best results since this is the icon size displayed by Explorer under Windows. Other icon sizes will need to be scaled. Using a 16 color icon gives better results than 256 colors on displays with a limited number of colors.

Drawing Tools

The image editor displays a grid which represents an enlarged version of your image, plus an actual size image and a color selector. The image can be edited using various tools which are available from the toolbar at the left of the window or from the Tools menu:

Pen : Click to set single pixels or hold mouse button for freehand drawing.

Line : Click and drag to set opposite ends of the line.

Outline Rectangle : Click and drag to set opposite corners of the rectangle.

Filled Rectangle : Click and drag to set opposite corners of the rectangle.

Fill : Click to flood fill an area in the current color.

Hotspot : Click to set the selection point within a cursor.

Mirror image : Reverses the image horizontally.

Flip image : Reverses the image vertically.

Color Selection

The current drawing color can be selected in two ways. Normally colors will be selected using the Colour Selector window. To select a color use the left mouse button. Alternatively a color which has already been used in your image can be reselected using the Colour Picker tool and clicking within the main edit window. This is useful to ensure the exact shade is selected when editing a 256 color image.

In addition to the opaque colors, a transparent color can be selected when editing an icon or cursor. Transparent pixels allow the dialog or window background to show through. The Pen tool can be used to draw transparent pixels, regardless of the currently selected color, by using the right mouse button.

The palette used for an image can be changed in two ways: by right clicking a color in the Colour Selector window or by using the Edit Pixel Colour tool and clicking in the main edit window. Either of these will display a dialog to choose a new color for the corresponding palette entry. All pixels in the changed color will be updated to use the new color.

Cut and Paste

Rectangular areas of the image can be cut, copied and pasted using the Select Region tool and the options on the Edit menu. To cut or copy an area it must first be selected. To do this press and hold the left mouse button in one corner of the area to be selected. Next drag the mouse to the opposite corner and release the mouse button. To paste an area select Paste from the Edit menu. Now select the point at which to paste. To do this press and hold the left mouse button. An outline of the area to be pasted will appear. Drag this to the desired location and release the mouse button.

Cursor Hotspot

When designing a cursor it it important to set the cursor hotspot, i.e. the point in the image which represents the cursor location. This can be set using the Hotspot tool or via Resource→Properties.

Adding Images to Your Resource

Images which are used in your program are normally defined via your resource script. It is therefore usual to add image files to your resource either via Resource→Add or File→Import Image. The former option is used to create a resource file image from scratch, whereas the latter is used to import existing image files.

Selecting File→Import Image will prompt for a suitable identifier. This identifier will be used to refer to the image in your program or to link the image to a dialog field. Alternatively, if the image to be imported is an icon, this dialog allows you to select it as the program icon. An identifier is not required in this case.

Using Bitmaps and Icons in Dialogs

Both bitmaps and icons can be used in dialogs. They should be added to your resource script as described in the preceding section.

When adding or importing an image for use in a dialog, it is important to select an appropriate identifier. This is used to link the picture to a field. In general you should give the picture the same identifier as the field that will display it. However if you intend to assign the picture to the field at run time (using `WDialogPutImage`) then any identifier may be used.

Once a picture has been added to your resource script it can be displayed by Picture/Frame, Group Box, Push Button, Checkbox and Radio Button fields. The type of picture, if any, to display is set using the field’s Style Properties dialog. The same image can be used by multiple fields provided they have the same identifier or `WDialogPutImage` is used to assign the image at run-time. This results in a smaller executable than importing the same image multiple times with different identifiers.

Note : When editing a dialog, the resource editor displays a dummy bitmap/icon rather than the actual picture which will be used at run time.

To display program generated graphics in a dialog, use `IGrSelect` and a modeless or semi-modeless dialog. The most appropriate type of field to use for this purpose is the rectangle variation of the Picture/Frame field.

Using Cursors

Cursors are selected at runtime via the `WCursorShape` routine. This routine supports various pre-defined cursor shapes, using hardcoded identifiers. User defined cursors added via the resource script should therefore use identifier values greater than 100. The program cursor shape can be changed at any time.

8 Subroutine Summary

Group WM: Window Management

`WindowClear`	Clear all or part of a window
`WindowClose`	Close root window
`WindowCloseChild`	Close a child window
`WindowOpen`	Create root window with various style options
`WindowOpenChild`	Open a child window with various style options
`WindowOutStatusBar`	Output string to the status bar
`WindowSelect`	Specify number of parts in status bar
`WindowSizePos`	Change window size, position, state
`WindowStatusBarParts`	Select window which all output is sent to
`WindowTitle`	Set window title/icon
`WindowUnitsFromPixels`	Convert from pixels to W*interacter* co-ordinates
`WindowUnitsToPixels`	Convert to pixels from W*interacter* co-ordinates

Group MH: Message Handling

`WMessage`	Wait until valid message arrives
`WMessageEnable`	Enable/disable reporting of individual messages
`WMessagePeek`	Get next message or return immediately

Group MN: Menu Handling

`WMenu`	Activate/remove a menu
`WMenuFloating`	Display a floating menu at (x,y)
`WMenuGetState`	Get a menu item state (checked/greyed)
`WMenuSetState`	Set a menu item state (checked/greyed)
`WMenuSetString`	Change text for a given menu item

Group DM(1): General Dialog Management

`WDialogFieldState`	Enable/disable/show/hide a field
`WDialogHide`	Remove a dialog from the screen
`WDialogLoad`	Load a dialog from resource
`WDialogRangeProgressBar`	Set range for a progress bar
`WDialogSelect`	Select current dialog
`WDialogSetField`	Make field current
`WDialogShow`	Display a dialog
`WDialogUnload`	Free dialog resource from memory

Group DM(2): Dialog Field Assignment/Retrieval

`WDialogGetCheckBox`	Get check box value
`WDialogGetMenu`	Get menu field value
`WDialogGetRadioButton`	Get radio-button group value
`WDialogGetString`	Get string value
`WDialogPutCheckBox`	Enable/disable a check-box
`WDialogPutImage`	Change bitmap/icon displayed in a field
`WDialogPutMenu`	Change menu field contents
`WDialogPutOption`	Change menu field option number
`WDialogPutProgressBar`	Change progress bar value
`WDialogPutRadioButton`	Change radio-button group value
`WDialogPutString`	Change a string field value

Group CD: Common Dialog Management

`WMessageBox`	Display standard message box with various options
`WSelectFile`	Get a load or save filename from a common dialog

Group GG: General Graphics

`IGrArea`	Define size of graphics area
`IGrAreaClear`	Clear the current graphics screen area
`IGrGetPixel`	Get a screen pixel value
`IGrInit`	Re-initialize graphics output
`IGrSelect`	Select target drawable (window or dialog field)
`IGrUnits`	Define plotting units to be used

Group GS: Graphics Style Selection

`IGrColourModel`	Select 8-bit or 24-bit color model
`IGrColourN`	Select graphics color using a color number
`IGrFillPattern`	Define fill pattern (solid/stippled/hatched)
`IGrLineType`	Select line type (solid, dashes, etc.)
`IGrPaletteInit`	Reinitialize graphics palette
`IGrPalette`	Redefine 8-bit color palette
`IGrPlotMode`	Select plot mode (overwrite or exclusive-or)

Group GD: Graphics Drawing Primitives

`IGrCircle`	Draw/fill circle at an absolute position
`IGrLineTo`	Draw line to a new absolute position
`IGrMoveTo`	Move current plotting position to a new absolute position
`IGrPoint`	Draw a single point at new absolute position
`IGrPolygonComplex`	Draw/fill a complex (possibly intersecting) polygon

Group GT: Graphics Text

`WGrTextFont`	Select font
`WGrTextLength`	Return relative length of string
`WGrTextOrientation`	Select graphics text alignment, angle and direction
`WGrTextString`	Output a string
`WGrOFontFixed`	Load outline font (Fixed)
`WGrOFontSwiss`	Load outline font (Swiss)
`WGrVFontDuplexRoman`	Load vector FONT (Duplex Roman)
`WGrVFontStandard`	Load vector font (Standard)
`WGrVFontTriplexRoman`	Load vector font (Triplex Roman)

Group IF: Information

`InfoError`	Return error information
`InfoGraphics`	Return real graphics information
`InfoGrPalette`	Return graphics palette information
`InfoGrScreen`	Return graphics facilities information (screen)
`WInfoDialog`	Return dialog information
`WInfoFont`	Return information about current font
`WInfoScreen`	Return screen size & available colors
`WInfoWindow`	Return information about the current window

Group OS: Operating System Interface

`IOsExitProgram`	Abort program with an error message & error code
`IOsVariable`	Return the value of an environment variable

Group MI: Miscellaneous

`WCursorShape`	Select mouse cursor shape
`WFlushBuffer`	Flush X Windows i/o buffer
`WglSelect`	Select target OpenGL drawable (window or dialog field)
`WglSwapBuffers`	Swap front/back OpenGL buffers
`WindowBell`	Sound the bell
`WInitialise`	Initialize W*interacter*
`WRGB`	Convert (*r*,*g*,*b*) triplet to 24-bit color value
`WRGBsplit`	Convert 24-bit color value to (*r*,*g*,*b*) triplet

Group CH: Character Manipulation Routines

`IFillString`	Fill a character string with a given character
`IJustifyString`	Justify a string within a character variable
`ILocateChar`	Locate position of first non blank character
`ILocateString`	Locate position of first non blank sub-string
`ILowerCase`	Convert a string to lower case
`IntegerToString`	Convert an integer value to a string
`IStringToInteger`	Convert a string to an integer value
`IUpperCase`	Convert a string to upper case

Group OB: Obsolete Routines

`IActualLength`	Return actual length of string excluding trailing blanks/nulls
`IGrCharJustify`	Select graphics text justification
`IGrCharLength`	Return relative length of string allowing for prop spacing
`IGrCharOut`	Output character string at an absolute `(x,y)` position
`IGrCharSet`	Select graphics character set to use for text output
`IGrCharSize`	Select graphics text/symbol size
`IGrCharSpacing`	Select fixed or proportional spacing
`IGrGetPixelRGB`	Get a screen pixel value as an `(r,g,b)` triplet
`IGrPaletteRGB`	Redefine 8-bit color palette using an`(r,g,b)` triplet
`IGrPause`	Start a new picture
`WindowClearArea`	Clear part of a window
`WindowFont`	Set all text attributes
`WindowOutString`	Output string at XY co-ordinate
`WindowStringLength`	Return the unit length of a string
`WMenuRoot`	Activate/remove root menu

9 Window Handling

Group WM: Window Management

This group provides routines to open and manipulate windows.

Two types of windows are controlled by the routines in the group. The root (or main) window should be opened first, using `WindowOpen`. Multiple child windows can then be opened/ closed using `WindowOpenChild` and `WindowCloseChild`. When window processing is complete, the root window and all its child windows can be removed using `WindowClose`. After the initial call to `WInitialise`, multiple calls to `WindowOpen`/ `WindowClose` are allowed, provided these calls are paired.

Windows opened by the routines in this group can be written to using the graphics routines described in the *High Resolution Graphics* chapter. Both `WindowSelect` and `IGrSelect` take a window handle as an argument (zero for the root window or as returned by `WindowOpenChild` for a child window). The same handle is used by `WindowCloseChild`, allowing child windows to be opened and closed in any sequence. OpenGL graphics can be displayed in a window by specifying the window handle to `WglSelect` (in the MI group).

Each window can have its own status bar, at the bottom of the window. This is selected at window creation. It can subsequently be sub-divided via `WindowStatusBarParts` and written to using `WindowOutStatusBar`.

The title string or icon of the current window can be updated at any time via `WindowTitle`. All or part of a window can be cleared by `WindowClear`. The position and/or size of the current window can be changed by `WindowSizePos`.

Two utility routines are provided to convert between W*interacter* window co-ordinates (0-9999) and pixel equivalents, namely `WindowUnitsToPixels` and `WindowUnitsFromPixels`.

WindowClear Subroutine

Description
Clear all or part of a window

Syntax
WindowClear(*ixtopl*,*iytopl*,*ixbotr*,*iybotr*,rgb)

Arguments
INTEGER, OPTIONAL *ixtopl* = Top left corner x co-ordinate

INTEGER, OPTIONAL *iytopl* = Top left corner y co-ordinate

INTEGER, OPTIONAL *ixbotr* = Bottom right corner x co-ordinate

INTEGER, OPTIONAL *iybotr* = Bottom right corner y co-ordinate

INTEGER, OPTIONAL *rgb* = Background color (24-bit RGB value)

Effect
Clears the specified area of the current window.

The co-ordinate arguments are expressed in pixels. If any are omitted, the corresponding window edge is assumed. If all four are omitted the entire window is cleared.

The new background color is determined by *rgb*, which is a 24-bit RGB value. If it is omitted, the default window background color is used (usually white).

Example
```
  CALL WindowClear()             ! clear window using defaults
  CALL WindowClear(RGB=RGB_BLUE) ! clear window to blue
```

WindowClose Subroutine

Description
Close root window.

Syntax
WindowClose()

Effect
Closes all opened windows, including the root window, freeing all resources and memory allocations. W*interacter* internals remain initialized however, allowing `WindowOpen` to be called again without reinitialization via `WInitialise`.

Example

```
CALL WInitialise()    ! Initialize Winteracter
CALL WindowOpen()     ! Open root window
CALL WindowClose()    ! Close root window
CALL WindowOpen()     ! Re-open root window
CALL WindowClose()    ! Close root window
```

WindowCloseChild Subroutine

Description

Close a child window

Syntax

WindowCloseChild(*ihandle*)

Arguments

INTEGER *ihandle* = W*interacter* child window handle (1-20)

Effect

Closes the specified child window. The window handle must have been obtained by a call to `WindowOpenChild`. If the specified window handle is invalid, no action is taken. If the closed child window is also the current window (as set by `WindowSelect` or `WindowOpenChild`) the W*interacter* output focus returns to the root window.

If a child window is closed by the user, via the system menu, a `CloseRequest` message is sent to the program via `WMessage`/`WMessagePeek`. This message returns the handle in the *win* member of the `WIN_MESSAGE` structure. The calling program is then responsible for closing the window by calling this routine.

Example

```
CALL WindowOpenChild(ICHILD1)  ! Open child window 1
CALL WindowOpenChild(ICHILD2)  ! Open child window 2
CALL WindowOpenChild(ICHILD3)  ! Open child window 3
        !
CALL WindowCloseChild(ICHILD2) ! Close child window 2
CALL WindowCloseChild(ICHILD1) ! Close child window 1
CALL WindowCloseChild(ICHILD3) ! Close child window 3
```

Errors

`ErrWinHandle`	(1003)	Invalid window handle

WindowOpen Subroutine

Description

Open root window

Syntax

WindowOpen(*flags*,*x*,*y*,*width*,*height*,*menuid*,*toolid*,*dialogid*,*title*,*ncol256*)

Arguments

INTEGER, OPTIONAL *flags*= Title bar buttons, etc. Sum of:

SysMenuOn	(1)	= System menu on title bar
MinButton	(2)	= Minimize button
MaxButton	(4)	= Maximize button
MaxWindow	(8)	= Maximize window
StatusBar	(32)	= Status bar
FixedSizeWin	(64)	= Fixed size window
HideWindow	(128)	= Hidden window
AlwaysOnTop	(512)	= Keep window on top

INTEGER, OPTIONAL *x* = Horizontal top corner of window in pixels

INTEGER, OPTIONAL *y* = Vertical top corner of window in pixels

INTEGER, OPTIONAL *width* = Width of window in pixels

INTEGER, OPTIONAL *height* = Height of window in pixels

INTEGER, OPTIONAL *menuid* = Main horizontal menu identifier

INTEGER, OPTIONAL *toolid*(4) = Reserved

INTEGER, OPTIONAL *menuid* = Identifier of dialog to combine with window

CHARACTER, OPTIONAL *title* = Window title

INTEGER, OPTIONAL *ncol256* = Number of colors in a 256-color video mode (16/32/64/128)

Effect

Opens the root window. Each call to `WindowOpen` must be paired with a call to `WindowClose` and cannot be nested. `WindowOpen` will :

- Create a new root window of the specified style, size, etc.
- Terminate with a message box if a fatal error occurs.

- Initialize any accelerators in the resource file.
- Display a menu given a valid menu id.
- Combine the window with a dialog given a dialog id.
- Initialise graphics output.

The *flags* argument determines various window attributes :

- Presence of a system menu.
- Maximise/minimise buttons on the title bar.
- Maximised or normal window size.
- Presence of a status bar.
- Should the window be fixed in size.
- Window visibility.
- Window stack ordering relative to other windows.

To assign values to *flags*, sum the appropriate SysMenu, MinButton, MaxButton etc. parameters.

The `HideWindow` option causes the window to be created but not displayed, allowing applications to open 'floating' child windows or pop-up dialogs without a visible root window. Child dialogs and 'inside-parent' child windows are not available in this case. Hidden windows still exist, so they must be closed using `WindowClose` as normal.

Specifying `AlwaysOnTop` forces a window to remain visible above all windows which do not have this window style. This also ensures that any dialogs opened by the program remain visible in the same way, even if `HideWindow` has been specified.

If `MinButton` or `MaxButton` are specified, `SysMenuOn` must also be specified.

If *flags* is omitted, a system menu and maximise/minimise buttons are selected.

x and *y* specify the position of the window relative to the full screen. To centre the window in either direction, omit the corresponding argument.

width and *height* specify the size of the client area of the window (i.e. the usable area of the window inside the frame). If either is omitted, the appropriate dimension is set to 80% of the screen size.

Alternatively, `WInfoScreen` can be used to interrogate the total screen size to allow screen-specific position and size values to be calculated.

menuid specifies a text menu in the program resource, which should appear as the root window menu. Omit this argument if no root menu is required (e.g. for a program which relies solely on floating menus and/or dialogs).

toolid is included for compatibility with the full version of W*interacter* but is not used in W*i*SK. It can be omitted.

dialogid specifies a dialog to be combined with the window. This dialog should not have already been loaded. The window will be opened at the exact size required to hold the specified dialog, allowing for the presence of a menu and status bar. Any specified window size is ignored when combining the window with a dialog. Similarly certain elements of the *flags* argument are overriden to ensure an appropriate window style. Specifically the `FixedSizeWin` flag is always applied and the `MaxButton`, `MaxWindow` and `HideWindow` flags are ignored. Before a dialog can be combined with a window in this way its type must first be set to 'Sub Component' in the resource editor. An error code will be set if any other type of dialog is specified. If menu accelerators are defined for the main menu of this window then these will be processed before keystrokes are passed to the combined dialog, regardless of the setting of the 'Allow Accelerators' flag in the resource editor. Since the dialog occupies the entire area of the window text and graphics output to this window is unavailable.However graphics can be redirected to a field in the dialog using `IGrSelect`. Omit this argument to open a standard window for use with text and graphics output.

title specifies the window title, if supplied. This can be changed later via `WindowTitle`.

ncol256 determines the number of graphics colors used in a 256 color video mode. This should be one of 16, 32, 64 or 128 colors. This determines the number of simultaneously selectable colors via `IGrColourN` on a 256 color display. It is not used at other color depths. A default of 16 is assumed if *ncol256* is omitted or invalid.

W*interacter* graphics routines are also initialised by calling `IGrInit(' ')`. See the documentation for `IGrInit` for details of the initial state of the graphics system. Screen graphics become available to any output window, including child windows opened subsequently by `WindowOpenChild`.

The actual size of the root window and the associated *flags* value can be obtained at any time after it has been opened via `WInfoWindow`. Note : This requires that the root window should have the current output focus (see `WindowSelect`).

Note : In earlier versions of W*i*SK, an alternative calling interface based on the `WIN_STYLE` structure was used. This has been replaced by the current calling interface which provides greater flexibility. The old `WIN_STYLE` interface is now obsolete, but is still supported. Use of the newer interface is recommended however.

Portability notes

X Windows: The `MaxWindow`, `SysMenuOn`, `MinButton`, `MaxButton` and `AlwaysOnTop` flags have no effect, since these features are controlled by the window manager.

Example

```
INTEGER, PARAMETER :: ID_MENU = 30001
! Initialise Winteracter once, the open window
CALL WInitialise()
CALL WindowOpen(Y=100,HEIGHT=250,MENUID=ID_MENU, &
                TITLE='Hello World')
```

WindowOpenChild Subroutine

Description

Open child window

Syntax

WindowOpenChild(ihandle,*flags*,*x*,*y*,*width*,*height*,*menuid*,*toolid*,*dialogid*,*title*,*iparent*)

Arguments

INTEGER *ihandle* = Returned window handle (1-20)

INTEGER, OPTIONAL *flags*= Title bar buttons, etc. Sum of:

SysMenuOn	(1)	= System menu on title bar
MinButton	(2)	= Minimize button
MaxButton	(4)	= Maximize button
MaxWindow	(8)	= Maximize window
InsideParent	(16)	= Window inside parent
StatusBar	(32)	= Status bar
FixedSizeWin	(64)	= Fixed size window
HideWindow	(128)	= Hidden window
OwnedByParent	(256)	= Keep window above parent
AlwaysOnTop	(512)	= Keep window on top

INTEGER, OPTIONAL *x* = Horizontal top corner of window

INTEGER, OPTIONAL *y* = Vertical top corner of window

INTEGER, OPTIONAL *width* = Width of window

INTEGER, OPTIONAL *height* = Height of window

INTEGER, OPTIONAL *menuid* = Main horizontal menu identifier

INTEGER, OPTIONAL *toolid*(4) = Reserved

INTEGER, OPTIONAL *dialogid* = Identifier of dialog to combine with window

CHARACTER, OPTIONAL *title* = Window title

INTEGER, OPTIONAL *iparent* = Handle of parent window

Effect

Opens a child window using a similar set of arguments to `WindowOpen`. A maximum of 20 child windows can be open at one time.

A window handle is returned in *ihandle* which should be used in subsequent calls to `WindowCloseChild`, `WindowSelect`, `WMenu`, `IGrSelect` or `WglSelect`. If a window could not be created then *ihandle* will be returned as -1. The first child window to be opened receives handle 1. All subsequently opened windows have incremental values. Closing windows out of sequence will cause W*interacter* to reuse the freed handles for subsequently opened windows. (Note : *ihandle* is a W*interacter* handle, not a Windows API or Motif handle.)

Child windows can have a parent window specified via the *iparent* argument. If no parent window is specified then the root window will be used as the parent window. The parent window is used only with the `InsideParent` and `OwnedByParent` flags. Other types of child window do not have a parent window.

The *flags*, *x*, *y*, *width*, *height*, *menuid*, *toolid*, *dialogid* and *title* arguments have the same meaning as for `WindowOpen`, except that :

a) `InsideParent` is available as one of the *flags* styles. When `InsideParent` is not specified, a child window can move anywhere on the screen. Its size and position should then be specified in screen (pixel) units, as for the root window. If `InsideParent` is specified, the child window will be restricted to the window specified by *iparent*. The window size and position should then be specified in W*interacter* window units (0-9999). This style of window is not available when the parent window is combined with a dialog. (Note : The more general `InsideParent` name replaces the earlier `InsideRoot`. Both names have the same value. The latter name is still supported for backwards compatibility.)

b) *menuid* is used only when `InsideParent` is not specified.

c) If `InsideParent` is specified and any of *x*, *y*, *width* or *height* are omitted, the default values are calculated in the same manner as for `WindowOpen` but relative to the parent window rather than the full screen.

d) `OwnedByParent` is also available as one of the *flags* styles. This can be used when `InsideParent` has not been specified, to force the window to be owned by the parent window. This ensures that the child window remains above the parent window at all times and causes the child window to minimise automatically when the parent window is minimised. If this flag is not set, the child window effectively becomes a completely independent window.

e) `HideWindow` is only normally useful for special types of window which can be created with the full version of W*interacter*. If it is used with W*i*SK, the window can be revealed later using `WindowSizePos`.

f) `AlwaysOnTop` affects child windows when `InsideParent` has not been specified, but has no effect otherwise. When used on a non-`InsideParent` child window, `AlwaysOnTop` keeps the child window visible above any other applications windows. Its visibility relative to the root window then depends on whether that too has the `AlwaysOnTop` style and whether `OwnedByRoot` was specified.

When a child window is opened it receives the current output focus and all subsequent output will go to this window until `WindowSelect`, `WindowOpenChild` or `WindowCloseChild` are called. See also `IGrSelect`.

As for the root window, the actual size of a child window and the specified *flags* value can be obtained at any time after it has been opened via `WInfoWindow`, provided it has the current output focus.

If the parent window is hidden (i.e. `HideWindow` was specified when the parent window was opened), child windows must not specify `InsideParent` as one of the styles in *flags*. No window will be opened if `InsideParent` is specified when the parent window is hidden and an error code will be generated. The window handle will also be returned as -1.

Minimised `InsideParent` child windows appear in their iconised form just above the status bar (if present) in the parent window. Maximised child windows will not obscure the status bar.

Portability notes

X Windows: Windows opened with the `InsideRoot` flag are not restricted to the parent window. See also the `WindowOpen` Portability notes.

Example

```
CALL WindowOpenChild(IHAND1,FLAGS=SysMenuOn+FixedSizeWin, &
     WIDTH=400,HEIGHT=300,TITLE='Child Window')
CALL WindowOpenChild(IHAND2,FLAGS=SysMenuOn+InsideParent, &
     WIDTH=2000,HEIGHT=1500,TITLE='Child Window',IPARENT=IHAND1)
```

Errors

`ErrRootHidden`	(1016)	`InsideRoot` specified when root window is hidden

WindowOutStatusBar Subroutine

Description

Write text on the status bar

Syntax

WindowOutStatusBar(*ipart*,*string*)

Arguments

INTEGER *ipart* = Status bar sub-division number (1-255)

CHARACTER *string* = String to write

Effect

Outputs *string* to the specified sub-division of the status bar in the current window. If the window has no status bar or the window has less than *ipart* sections, this routine has no effect. The contents of the status bar are maintained automatically, so there is no need to perform Expose/Resize processing on text written here.

By default text is output left justified within the sub-division. Leading spaces are not removed. The text can also be centred or right justified. To centre the text prefix the string with a single tab character, `ACHAR(9)`. To right justify the text use two tab characters.

Portability notes

Windows : The status bar font is determined by the 'Tooltips' font setting in Display:Appearances in Control Panel.

X Windows: The status bar font is determined by the `Wint*fontList` setting in `.Xdefaults` in the current user's home directory.

Example

```
  CALL WindowOutStatusBar(1,’This is on the status bar’)
```

WindowSelect Subroutine

Description

Select window to receive the output focus

Syntax

WindowSelect(*ihandle*)

Arguments

INTEGER *ihandle* = W*interacter* window handle (1-20 or 0 for root window)

Effect

Selects the window for W*interacter*'s various window manipulation routines such as `WindowClear`, `WindowTitle`, `WindowSizePos`, etc. It also selects the target window for graphics output when the current target drawable is set to a window (rather than a dialog), as determined by `IGrSelect`. Where menus have been added to child windows, this routine also determines which menu is affected by many of the menu handling routines in the MN group.

Handle 0 specifies the root window, otherwise, *ihandle* must have been returned by an earlier `WindowOpenChild` call. If *ihandle* specifies a non-existent window, no action is taken.

Do not confuse the W*interacter* output focus with the Windows input focus (i.e. the front window). W*interacter* allows manipulation of any of its windows while receiving input from whichever window has the input focus. The two mechanisms are entirely separate.

Example

```
CALL WindowOpen(TITLE='Root window')     ! Create root window
CALL WindowOpenChild(IH1,Y=300,TITLE='Child 1')
                                         ! Open child window 1
CALL WindowOpenChild(IH2,Y=600,TITLE='Child 2')
                                         ! Open child window 2
CALL WindowSelect(0)                     ! Select root window
CALL WGrTextString(0.5,0.5,'Root Window')
CALL WindowSelect(IH2)                   ! Select child #2
CALL WGrTextString(0.5,0.5,'Child Window 2')
CALL WindowSelect(IH1)                   ! Select child #1
CALL WGrTextString(0.5,0.5,'Child Window 1')
```

Errors

`ErrWinHandle`	(1003)	Invalid window handle

WindowSizePos Subroutine

Description

Set the size, position or state of the current window

Syntax

WindowSizePos(*width*,*height*,*x*,*y*,*istate*)

Arguments

INTEGER, OPTIONAL *width* = New window width

INTEGER, OPTIONAL *height* = New window height

INTEGER, OPTIONAL *x* = New window X position

INTEGER, OPTIONAL *y* = New window Y position

INTEGER, OPTIONAL *istate* = New window state:

WinMinimised	(0) : minimized
WinNormal	(1) : normal size
WinMaximised	(2) : maximized
WinHidden	(3) : hidden

Effect

Changes the size, position and/or state of the current window. Units for window size and position are in the same units as the call to `WindowOpen` or `WindowOpenChild` used to open the window, i.e. pixels for the root window and non-`InsideParent` child windows and W*interacter* window units for `InsideParent` child windows. Similarly *width* and *height* specify the usable area within the window and *x*/*y* specify the top-left corner of the window, in the same way as for `WindowOpen` and `WindowOpenChild`. If an argument is omitted then the corresponding size or position is left unchanged. Calling this routine will cause a Resize or Expose message to be reported by `WMessage` or `WMessagePeek` if the window size is changed or moving or displaying the window causes it to require redrawing.

This routine can be used with non-resizeable windows, i.e. those with `FixedSizeWin` specified in the call to `WindowOpen` or `WindowOpenChild` used to open them. In this case the window becomes fixed at the newly specified size. i.e. specifying `FixedSizeWin` prevents the user from changing the window size, but still allows program control.

When the window state is maximized, minimized or hidden the window size and position are not updated immediately. Instead `WindowSizePos` specifies the size and position which the window will have when the user restores the window to its normal size or the window is redisplayed without specifying a position or size.

If the current window has been combined with a dialog then only the position and visibility can be changed, any specified size is ignored.

This routine should be used with care. Moving or resizing windows under program control can confuse users if they have not explicitly asked for it to be done, e.g. by choosing a Tile or Cascade option from a Window menu. Resizing a window can also be acceptable if the user has chosen an option which changes the data displayed in a window and a different window size is required to view the new data. In this case the window should only be moved if required for the new window size to fit on screen.

Portability notes

X Windows: The *istate* argument only controls visibility. `WinMinimised` and `WinMaximised` are treated as `WinNormal`, since minimised and maximised states are controlled entirely by the window manager.

Example

```
!Set usable area of floating window to 800 by 600 pixels
CALL WindowSizePos(800,600)
```

WindowStatusBarParts Subroutine

Description

Sub-divides the status bar for the current window

Syntax

WindowStatusBarParts(*nparts,iwidths,istyles*)

Arguments

INTEGER *nparts* = Number of parts to divide status bar into (1-255)

INTEGER *iwidth(*)* = Array of status bar section widths

INTEGER, OPTIONAL *istyles(*)* = Array of sub-division border widths (0-2)

Effect

Sub-divides the status bar for the current window into the specified number of parts. The text in these sub-divisions can then be updated via separate calls to `WindowOutStatusBar`.

nparts specifies how many parts the bar should be divided into. The *iwidths* array defines how wide each part should be, in W*interacter* text window units. The special value of -1 can be used to indicate that the specified part extends to the end of the window. All subsequent widths will then be ignored.

The optional border style for each sub-division can be specified using *istyles*, where 0=none, 1=sunken (default) and 2=raised.

If the current window, as set by `WindowSelect`, `WindowOpen` or `WindowOpenChild` does not have a status bar, this routine has no effect.

Example

```
CHARACTER(LEN=12) :: FILENAME = 'default.dat'
CALL WInitialise()
CALL WindowOpen(FLAGS=SysMenuOn+StatusBar, &
                TITLE='Window with status bar')
CALL WindowStatusBarParts(2,(/2000,-1/))
CALL WindowOutStatusBar(1,'File:')
CALL WindowOutStatusBar(2,FILENAME)
```

WindowTitle Subroutine

Description

Set title/icon of current window

Syntax

WindowTitle(*title*,*idicon*)

Arguments

CHARACTER *title* = Window title string

INTEGER *idicon* = Icon identifier (0=revert to program icon)

Effect

Updates the title string and/or icon of the currently selected output window. If *title* is omitted then the current title string is used and only the icon is changed. Similarly if *idicon* is omitted the current icon is retained and only the title string is changed. The change takes effect immediately.

Example

```
CALL WindowOpen(TITLE=' Original Root Window Title ')
! . . .
CALL WindowSelect(0)
CALL WindowTitle('New Root Window Title')
```

WindowUnitsFromPixels Subroutine

Description

Convert pixel co-ordinate to W*interacter* units

Syntax

WindowUnitsFromPixels(*ixpix*,*iypix*,*ixwin*,*iywin*)

Arguments

INTEGER *ixpix* = X co-ordinate in pixels

INTEGER *iypix* = Y co-ordinate in pixels

INTEGER *ixwin* = Returned X co-ordinate in W*interacter* units (0-9999)

INTEGER *iywin* = Returned Y co-ordinate in W*interacter* units (0-9999)

Effect

Converts the supplied pixel co-ordinate in the currently selected window to the equivalent W*interacter* window units, as used by `WindowOpenChild` and `WindowStatusbarParts`.

WindowUnitsToPixels Subroutine

Description

Convert W*interacter* window co-ordinates to pixels

Syntax

WindowUnitsToPixels(*ixwin*,*iywin*,*ixpix*,*iypix*)

Arguments

INTEGER *ixwin* = X co-ordinate in W*interacter* units (0-9999)

INTEGER *iywin* = Y co-ordinate in W*interacter* units (0-9999)

INTEGER *ixpix* = Returned X co-ordinate in pixels

INTEGER *iypix* = Returned Y co-ordinate in pixels

Effect

Converts the supplied W*interacter* window units in the currently selected window to the equivalent pixel values.

10 Input Handling

Group MH: Message Handling

The `WMessage` routine will be at the core of most W*interacter* programs, reporting all forms of user input. It reports the main events which a GUI based program needs to know about including menu selections, key presses, mouse clicks, and so on. These events are reported as 'messages', with associated information being returned at each event (e.g. the location of the mouse cursor when the user clicked a button). A typical W*interacter* program will operate around a `DO` loop containing a `WMessage` call and a `SELECT CASE` statement which processes each type of message.

`WMessage` is complemented by the alternative routine `WMessagePeek`. This performs exactly the same task, but does not block program execution if no messages are waiting to be processed.

Since some message types may only be needed in certain applications or in particular parts of an application, reporting of specific message types can be individually enabled or disabled via `WMessageEnable`.

WMessage Subroutine

Description

Get next message.

Syntax

WMessage(*itype*,*value*)

Arguments

INTEGER *itype* = The type of message that is returned:

Table 2: Message types

Name	no.	Message type
KeyDown	1	Key press
MenuSelect	2	Menu item selected
PushButton	3	Push Button pressed
MouseButDown	4	Mouse button down
MouseButUp	5	Mouse button up
MouseMove	6	Mouse moved
Expose	7	Window expose
Resize	8	Window resize
CloseRequest	9	Window close requested
FieldChanged	10	Changed to a new dialog field
BorderSelect	12	Window selected via border/ title-bar
MouseDoubleClick	16	Mouse button double clicked

WIN_MESSAGE *value* = Structure containing additional message information

TYPE WIN_MESSAGE

INTEGER	*win*	Window or Dialog the message came from
INTEGER	*value1*	Message-type dependent parameter #1
INTEGER	*value2*	Message-type dependent parameter #2
INTEGER	*value3*	Message-type dependent parameter #3
INTEGER	*value4*	Message-type dependent parameter #4
INTEGER	*x*	X co-ordinate in W*interacter* window units (0-9999)
INTEGER	*y*	Y co-ordinate in W*interacter* window units (0-9999)
REAL	*gvalue1*	*value1* expressed in graphics units
REAL	*gvalue2*	*value2* expressed in graphics units
REAL	*gx*	X co-ordinate in graphics units (see `IGrUnits`)
REAL	*gy*	Y co-ordinate in graphics units (see `IGrUnits`)
INTEGER	*time*	Message time in milliseconds

END TYPE WIN_MESSAGE

Effect

Returns the next message from the event queue and returns any parameters associated with that message. If no message is available, `WMessage` waits for the next event to occur. Use the altenative `WMessagePeek` to continue processing if no messages are waiting.

Only those messages enabled via `WMessageEnable` are reported. The initial reporting state for each message type is shown at the start of each message type description as '(Default : Enabled/Disabled)'.

For all message types (except 3 and 10), *value%win* identifies the window which generated the message. For `PushButton` and `FieldChanged` messages, *value%win* specifies the unique identifier of the dialog which generated the message.

value%time reports the system time when the event occurred in milliseconds See the Portability notes for further details.

All other information returned in *value* is message-type dependent:

itype = `KeyDown` (Default : Enabled)

If the user presses a key which is not processed as a keystroke in a dialog or a menu selection, the key code will be returned in *value%value1*. `KeyDown` messages will *not* be reported when a dialog or menu has the focus. The (x,y) co-ordinate of the mouse cursor when the key was pressed will also be returned. The possible key codes which can be returned are summarized in the following table:

Table 3: Key codes

Name	Code	Key	Name	Code	Key
-	1-31	Ctrl/keys (Ctrl/A=1)	-	32-255	Printable characters
KeyBackSpace	8	Backspace	KeyInsert	272	Insert
KeyTab	9	Tab	KeyDelete-Under	273	Delete under cursor Key
Return	13	Return	KeyShiftTab	274	Shift/Tab
KeyEscape	27	Escape	Keypad0	280	Keypad 0
KeyDelete	127	Delete left	Keypad1	281	Keypad 1
KeyCursorUp	258	Cursor Up	Keypad2	282	Keypad 2
KeyCursorDown	259	Cursor Down	Keypad3	283	Keypad 3
KeyCursorRight	260	Cursor Right	Keypad4	284	Keypad 4

Table 3: Key codes

Name	Code	Key	Name	Code	Key
KeyCursorLeft	261	Cursor Left	Keypad5	285	Keypad 5
KeyPageUp	262	Page Up	Keypad6	286	Keypad 6
KeyPageDown	263	Page Down	Keypad7	287	Keypad 7
KeyPageRight	264	Shift/cursor right	Keypad8	288	Keypad 8
KeyPageLeft	265	Shift/cursor left	Keypad9	289	Keypad 9
KeyUpExtreme	266	Ctrl/cursor up	KeypadMi-nus	290	Keypad -
KeyDownEx-treme	267	Ctrlcursor down	KeypadPoint	291	Keypad .
KeyRightExtreme	268	Ctrl/cursor right	KeypadPlus	292	Keypad +
KeyLeftExtreme	269	Ctrl/cursor left	KeypadDi-vide	293	Keypad /
KeyHome	270	Home	KeypadMul-tiply	294	Keypad *
KeyEnd	271	End			
KeyF1-KeyF20	301-320	F1-F20			
KeyShiftF1-KeyShiftF20	321-340	Shift/F1 - Shift/F20			
KeyCtrlF1-KeyCtrlF20	341-360	Ctrl/F1-Ctrl/F20			

Note that F10 is not available, since this is used to activate the system or application menu, in the same manner as pressing/releasing the Alt key. Other function key codes may not be available across all platforms. See the Portability notes for details.

itype = `MenuSelect` (Default : Enabled)

When a menu item is selected, its unique identifier is returned in *value%value1*. This will correspond to the identifier defined in the program's resource file.This message will also be generated when the user presses an accelerator key.

itype = `PushButton` (Default : Enabled)

When a push button is pressed in a modeless or semi-modeless dialog, the unique identifier of that button is returned in *value%value1*. This will correspond to the identifier defined in the program's resource file. *value%value2* will be set to the identifier of the currently selected field in the corresponding dialog when the button was pressed.(*Note* : The button pressed to terminate a modal dialog is available separately, via `WInfoDialog(ExitButton)`)

Certain standard push-button identifiers are pre-defined in the `WINTERACTER` module :

```
INTEGER, PARAMETER :: IDOK     = 1
INTEGER, PARAMETER :: IDCANCEL = 2
INTEGER, PARAMETER :: IDABORT  = 3
INTEGER, PARAMETER :: IDRETRY  = 4
INTEGER, PARAMETER :: IDIGNORE = 5
INTEGER, PARAMETER :: IDYES    = 6
INTEGER, PARAMETER :: IDNO     = 7
INTEGER, PARAMETER :: IDCLOSE  = 8
INTEGER, PARAMETER :: IDHELP   = 9
```

Attempting to close a dialog (via the × button, via Close on the system menu or by pressing Alt/F4) will also generate a `PushButton` message, with a button identifier value of `IDCANCEL` (2). Pressing F1 in a dialog will generate a `PushButton` message, with a button identifier value of `IDHELP` (9).

itype = `MouseButDown` (Default : Enabled)
itype = `MouseButUp` (Default : Disabled)

When a mouse button-down or button-up occurs, *value%value1* will contain the button number:

LeftButton	(1)	Left button pressed
MiddleButton	(2)	Middle button pressed (where available)
RightButton	(3)	Right button pressed

The (x,y) co-ordinate of the mouse cursor when the event occurred is returned.

value%value2 reports the state of the Ctrl/Shift keyboard modifiers and all three mouse buttons at the time the event occurred, summed as follows:

ModCtrl	(1)	Ctrl key down
ModShift	(2)	Shift key down
ModLeftButton	(4)	Left button down
ModMiddleButton	(8)	Middle button down (where available)
ModRightButton	(16)	Right button down

If any picture/frame fields in a currently visible dialog have been selected as target drawables (via `IGrSelect`), these may report mouse button messages. *value%value3* will be set to `FromWindow` (0) or `FromDialog` (1) to indicate the source of the message. In the latter case, *value%value4* reports the identifier of the field in which the button down/up occurred and *value%win* reports the dialog id.

(In the original W*i*SK release, *value%value2* returned a centisecond time stamp which is no longer supported. It was superseded by the *value%time* millisecond time stamp.)

itype = `MouseMove`   (Default : Disabled)

When a mouse moves, the new mouse (x,y) co-ordinate is returned.

This messages is only reported if specifically enabled via `WMessageEnable`. Programs which enable this message must be prepared to process large numbers of movement messages.

value%value2/3/4 report the same values as `MouseButDown`/`Up` messages.

itype = `Expose`   (Default : Enabled)

If all or part of a window becomes exposed, that window area will need to be repainted. The returned (x,y) co-ordinate identifies the top left corner of the exposed area. *value%value1* and *value%value2* define the width and height of the exposed area.

If any fields in a currently visible dialog have been selected as target drawables (via `IGrSelect`), these must be maintained by the calling program. If such a field needs redrawing, an `Expose` message will be reported. *value%value3* will be set to `FromWindow` (0) or `FromDialog` (1) to indicate the source of the message. In the latter case, *value%value4* reports the identifier of the field which needs to be redrawn and *value%win* reports the dialog id. The exact area exposed is not reported in this case. The entire field should be redrawn.

itype = `Resize`   (Default : Enabled)

When the user resizes a root or child window, *value%value1* and *value%value2* return the new window width/height in pixels.

itype = `CloseRequest`   (Default : Enabled)

If a user selects Close on the system menu (or clicks on the Close button under Windows) a `CloseRequest` message is returned. The handle of the window which originated the request is returned in *value%win*. This will be zero for the root window. It is then the responsibility of the calling program to close that window (via `WindowCloseChild` or `WindowClose`), if the close request is to be allowed. Typically, a `CloseRequest` message from the root window should result in program termination. However, the calling program may wish to ask for the user's confirmation (e.g. via `WMessageBox`) and close any data files, before terminating.

itype = `FieldChanged` (Default : Disabled)

When a user changes a field value or moves between fields in a modeless/semi-modeless dialog, *value%value1* will be set to the identifier of the previous field and *value%value2* will be set to the 'moved to' or current field. If the current field number has not changed (e.g. the user has just changed their selection in a menu field), *value%value1* will be the same as *value%value2*. field This allows the calling program to perform field-by-field validation without having to wait for a dialog to terminate via a `PushButton` event.

It should be understood that a `FieldChanged` message is only generated under the following general conditions :

1) When a field which has multiple known states (a menu, radio button or check box) changes value.

2) When the user moves between fields (e.g. using the tab/back-tab keys or by clicking on another field with the mouse).

By implication, `FieldChanged` messages do not occur in response to every keystroke in an enterable field. This allows users to edit such a field (e.g. using cursor keys, backspace, etc.) without the application attempting to perform intrusive validation on an incomplete field value.

itype = `BorderSelect` (Default : Disabled)

When the user selects a window by clicking on the title bar or border controls, a `BorderSelect` message reports the selected window in the *value%win* parameter. The mouse button number is returned in *value%value1* as for a `MouseButDown` message.

In practice, this message very rarely needs to be used, since other messages already report the originating window. Note in particular that a focus change via a mouse click in a window's client area will be reported separately as a `MouseButDown` message. The `BorderSelect` message is not reported by default since most programs need not be concerned about which window currently has the input focus. It is also non-portable. Use of `BorderSelect` is discouraged.

itype = `MouseDoubleClick`(Default : Disabled)

When the user double-clicks a mouse button this message will be reported in response to the second click. A `MouseButDown` message will already have been reported for the first click. If this message type is disabled `MouseButDown` messages are reported for both clicks. Since a double-click also generates a `MouseButDown` message you should design your user interface so that the action for a single-click is a subset of the actions for a double-click. Details reported by *value* are the same as for `MouseButDown` messages.

Portability notes

Windows:

`ITYPE=KeyDown` : Function keys 13-20 are not normally available but are defined within Windows itself so we allow for them here.

value%time is the elapsed time in milliseconds since the system start time. It is available for all Windows message types.

X Windows:

`ITYPE=KeyDown` : Keyboard handling is necessarily generalised under X Windows. However, all the listed keys have the potential to be generated under X.

`ITYPE=BorderSelect` : This message is not available under X. All border selection actions are intercepted by the window manager.

value%time is only available for keyboard and mouse messages. The returned value is measured relative to the X server start time.

Example

```
TYPE (WIN_MESSAGE) :: MESG
DO
  CALL WMessage(ITYPE,MESG)
  SELECT CASE (ITYPE)
    CASE (MenuSelect)     ! Check for 'Exit' menu option
      SELECT CASE (MESG%VALUE1)
        CASE (IDM_OPEN) ! Open option on the menu
           CALL OpenMyFile
        CASE (ID_EXIT)  ! Was Exit selected from menu
           EXIT
      END SELECT
    CASE (MouseButDown)    ! Display floating menu at mouse pos
      CALL WMenuFloating(ID_MENU_TWO, MESG%X, MESG%Y)
    CASE (Resize, Expose) ! Redraw graph
      CALL Draw_My_Graph()
    CASE (CloseRequest)   ! Close button or System menu option
      IF (MESG%WIN==0) EXIT
  END SELECT
END DO
CALL WindowClose()        ! Close root window and all children
```

WMessageEnable Subroutine

Description

Enable/disable message reporting.

Syntax

WMessageEnable(*itype*,*ionoff*)

Arguments

INTEGER *itype* = Message type, as for `WMessage`

INTEGER *ionoff* = Turn message on/off (Disabled (0) = off, Enabled (1) = on)

Effect

Enables or disables reporting of the specified message type via `WMessage` and `WMessagePeek`. See the '(Default: Enabled/Disabled)' notes in `WMessage` for details of the individual default reporting states. Note that disabling a particular message type does not prevent that event from occurring. It simply determines whether the event is reported to the calling program.

Example

```
CALL WMessageEnable(KeyDown,Disabled)  ! keystroke messages off
CALL WMessageEnable(MouseMove,Enabled) ! mouse move messages on
```

WMessagePeek Subroutine

Description

Get next message. Return if none waiting.

Syntax

WMessagePeek(*itype*,*ivalue*)

Arguments

See `WMessage`

Effect

This routine is identical to `WMessage` except that it will not 'block' if no message is waiting. If the input queue contains no messages, *itype* will be returned as `NoMessage` (-1). Use `WMessagePeek` where a program wishes to poll for messages but needs to continue processing if no events have occurred.

Example

```
TYPE (WIN_MESSAGE) :: MESG
DO Iteration = 1, 100000
  CALL WMessagePeek(ITYPE, MESG)
  IF (ITYPE/=NoMessage) THEN
    ! Process message(s) here
  END IF
  ! Next iteration of number cruncher
  CALL Number_Cruncher(Iteration)
END DO
```

Group MN: Menu Handling

Menu selections are reported via `WMessage` in the MH group. The routines in this group provide the remaining menu handling capabilities, namely :

- Activation and removal of main menus via `WMenu`.
- Activation of floating menus via `WMenuFloating`.
- Setting and retrieving the state (i.e. checked and/or greyed) of individual menu items via `WMenuSetState`/`WMenuGetState`.
- Updating the strings associated with a given menu item, via `WMenuSetString` (initial strings can be assigned in the resource file).

All the routines in this group use unique menu or menu-item identifiers as defined in the program resource file.

Routines which change or interrogate the state of menu items operate on the current window, as determined by `WindowSelect`, if the current window has a main menu. Where the current window does not have a main menu these routines operate on the root window menu.

WMenu Subroutine

Description

Activate or remove a menu structure.

Syntax

WMenu(*menuid*,*iwindow*)

Arguments

INTEGER *menuid* = Identifier of root menu to activate (0 to remove current root menu)

INTEGER, OPTIONAL *iwindow* = Window handle

Effect

Activates the specified main menu structure, which will be attached to the top of the specified window. If no window is specified the menu is attached to the top of the root window. Multiple menus are allowed in a single program as a result, though only one can be active on each window at one time. To remove the current menu, specify a zero menu identifier. The specified window must be the root window or a popup child window. Menus are not available in child windows inside their parent window.

Unlike `WMenuFloating`, this routine does not block. It simply updates the currently displayed main menu. Item selections are reported via `WMessage` in a `MenuSelect` message.

`WMenu` also loads and activates an accelerator table from the program resource, if such a table exists with the same identifier. An accelerator table identifies keystrokes which can be used to directly select a menu item from the keyboard, without having to navigate the menu. When a key is pressed the accelerator table for the menu on the active window is checked. If the current window does not have a menu then the accelerator table for the root window's menu is checked.

Depending on the number and length of the menu items and the current window size calling this routine may cause a change in the size of the useable window area. This will be reported via `WMessage` as a Resize message.

The state of individual menu items (greyed/checked) can be set via `WMenuSetState`. Menu item strings can be updated via `WMenuSetString`.

Example

```
CALL WMenu(0)               ! Remove menu from root window
CALL WMenu(ID_MENU1,IHAND) ! Add a menu to a child window
```

Errors

`ErrLoadMenu`	(1002)	Unable to load menu from resource

WMenuFloating Subroutine

Description

Activate a floating (vertical) menu.

Syntax

WMenuFloating(*menuid*,*ixpos*,*iypos*)

Arguments

INTEGER *menuid* = Identifier of floating menu to activate

INTEGER *ixpos* = X position of top left of menu

INTEGER *iypos* = Y position of top left of menu

Effect

Activates the specified floating menu at the specified (x,y) position. The program will block until a selection is made or the menu is cancelled. If a selection is made, it will be reported via `WMessage`.

The (x,y) co-ordinate is measured in W*interacter* window units relative to the current window as selected by `WindowSelect`.

If any of the menu items on the floating menu have the same identifiers as items in the current root menu, their state (greyed/checked) and text can be set via `WMenuSetState` and `WMenuSetString`.

Example

```
CALL WMessage(ITYPE,MESSAGE)
SELECT CASE (ITYPE)
      :
!  Use the right mouse button to display a floating menu.
     CASE (MouseButDown)
         IF (MESSAGE%VALUE1==RightButton) &
         CALL WMenuFloating(IDM_SHORTCUT,MESSAGE%X,MESSAGE%Y)
     CASE (MouseMove)
      :
END SELECT
```

Errors

`ErrLoadMenu`	(1002)	Unable to load menu from resource

WMenuGetState Function

Description

Get grayed/checked state of a menu item.

Syntax

INTEGER WMenuGetState(*menuitem*,*iprop*)

Arguments

INTEGER *menuitem* = Menu item identifier as specified in resource file

INTEGER *iprop* = Property to retrieve:

ItemEnabled (1): Is item enabled ?

Returns	:	Disabled	(0) Item is greyed out
		Enabled	(1) Item is selectable

ItemChecked (2): Is item checked ?

Returns	:	Unchecked	(0) No check mark
		Checked	(1) Check mark is present

Effect

Retrieves the state of the specified menu item property. Only one property can be interrogated at one time. An `INTEGER` binary flag is returned to indicate the state of the specified property. See also `WMenuSetState`.

Example

see `WMenuSetState`

Errors

`ErrMenuItem`	(1001)	Invalid menu item

WMenuSetState Subroutine

Description

Set grayed/checked state of a menu item.

Syntax

WMenuSetState(*menuitem*,*iprop*,*ivalue*)

Arguments

INTEGER *menuitem* = Menu item identifier as specified in resource file.

INTEGER *iprop* = Property to set:

ItemEnabled	(1)	Enable item
ItemChecked	(2)	Check item

INTEGER *ivalue* = New state for menu item property (WintOff (0): Off, WintOn (1): On)

Effect

Sets the state of the specified menu item property. The specified menu item must exist on the current root menu. Floating menu items will also be affected if they share an identifier with an item on the current root menu.

iprop = ItemEnabled

When an item is enabled, it is selectable in the normal manner. When it is disabled, it will be greyed out.

iprop = ItemChecked

A checked item has a tick mark against it.This is used to indicate that a particular program option is currently enabled. It is not possible to add checks to top level root menu items.

Example

```
IPROP = WMenuGetState(ID_OPTION,MenuChecked) ! Toggle check mark
                                             ! next to an option
IPROP = 1 - IPROP
CALL WMenuSetState(ID_OPTION,MenuChecked,IPROP)
```

Errors

`ErrMenuItem`	(1001)	Invalid menu item

WMenuSetString Subroutine

Description

Change the text of a menu item

Syntax

WMenuSetString(*menuitem*,*string*)

Arguments

INTEGER *menuitem* = Menu item identifier as specified in resource file.

CHARACTER *string* = New text for the specified menu item.

Effect

Changes the text of the specified root menu item. Floating menu item strings will also be modified automatically, if they share an identifier with an item on the current root menu. Otherwise, floating menu item strings cannot be modified.

Example

```
IF (FIRSTTIME) THEN
    CALL WMenuSetString(ID_OPTION,'New data')
ELSE
    CALL WMenuSetString(ID_OPTION,'Old data')
END IF
```

Errors

`ErrMenuItem`	(1001) Invalid menu item

11 Dialog Manager

An overview of the facilities provided by dialogs and instructions on creating them using the W*interacter* resource editor, can be found in the earlier **Dialogs** chapter.

Group DM(1): General Dialog Management

The main routines in this group are those which load a dialog from a resource and then display it on the screen (`WDialogLoad` and `WDialogShow`). Modeless and semi-modeless dialogs can be hidden while not required to be visible, by `WDialogHide`. When a dialog is no longer needed in memory it can be unloaded completely by `WDialogUnload`. Up to 100 dialogs can be loaded simultaneously.

The majority of the routines in this group and in the DM(2) group operate on the 'current' dialog. This can be set by `WDialogSelect`. It is also set by `WDialogLoad` or (in the case of combined windows/dialogs) by `WindowOpen`/`WindowOpenChild`.

Progress bar range control is provided via `WDialogRangeProgressBar` .`WDialogFieldState` determines whether a given field is active. `WDialogSetField` forces the specified field to become the current field in a modeless dialog.

WDialogFieldState Subroutine

Description

Set the state of field.

Syntax

WDialogFieldState(*ifield*,*istate*)

Arguments

INTEGER *ifield* = Field identifier

INTEGER *istate* = Field state

Disabled	(0) : display only (protected)
Enabled	(1) : enterable (unprotected)
DialogReadOnly	(2) : read only (protected)
DialogHidden	(3) : hidden (protected)

Effect

Sets the state of the specified field. A field can be enabled (i.e. data can be entered in it or selections made), disabled/read-only (i.e. an output only field) or hidden (not shown on screen). This allows the calling program to selectively enable/disable data entry in particular fields at run time.

Read-only fields are the same as disabled fields except that the cursor/focus can be moved to the field and the contents can be copied to the clipboard. This option applies to fields for which a read-only state can be defined in the resource editor. For other field types, the field is disabled instead.

Hidden fields are most useful when a nearly identical dialog is required for multiple purposes. If the state of a field can change while the dialog is displayed it is usually preferable to disable it rather than hide it.

Example

```
! Disable all bar one field in the current dialog.
DO IFIELD = 1,NFIELD
    CALL WDialogFieldState(IFIELD,0)
END DO
CALL WDialogFieldState(IDF_FIELD3,1)
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogHide Subroutine

Description

Remove current dialog from screen.

Syntax

WDialogHide()

Effect

Remove the current dialog from the screen, but keep it in memory so that its contents can still be accessed. It is only necessary to call this routine for modeless or semi-modeless dialogs. Modal dialogs are automatically removed from the screen when `WDialogShow` terminates.

If the current dialog is semi-modeless and there are no other semi-modeless dialogs active, then all program windows and modeless dialogs are re-enabled.

A dialog can be removed from memory subsequently by calling `WDialogUnload`.

Example

```
CALL WMessage(ITYPE,MESSAGE)
SELECT CASE (ITYPE)
    CASE (MenuSelect)
        :
    CASE (PushButton)
        CALL WDialogHide()  ! Remove dialog from display
                            ! when a push-button is pressed
    CASE (MouseButDown)
        :
END SELECT
```

Errors

`ErrCurDialog`	(1005)	No current dialog

WDialogLoad Subroutine

Description

Load a dialog definition from program resource.

Syntax

WDialogLoad(*idialog*)

Arguments

INTEGER *idialog* = Identifier of dialog to load, as defined in the resource script.

Effect

Loads the specified dialog from the program resource. The dialog contents are initialized to the settings specified in the resource script. To make the dialog appear on screen, call `WDialogShow`. Optionally, the contents of the dialog can be modified using the various `WDialogPutXXX` routines in the DM(2) group, before the `WDialogShow` call.

`WDialogLoad` implicitly selects *idialog* as the current dialog for use by other W*interacter* routines. A dialog can be reselected after loading other dialogs by calling `WDialogSelect`.

If the dialog specified by *idialog* is already loaded, it will simply be re-selected as the current dialog. To completely reinitialize a dialog, unload it (via `WDialogUnload`) and reload it.

Example

```
CALL WDialogLoad(IDD_ABOUT)
CALL WDialogShow(ITYPE=Modal)
CALL WDialogUnload()
```

Errors

`ErrLoadDialog`	(1007)	Unable to load dialog from resource

WDialogRangeProgressBar Subroutine

Description

Set the range of a progress bar field

Syntax

WDialogRangeProgressBar(*ifield*,*ipbmin*,*ipbmax*)

Arguments

INTEGER *idialog* = Progress bar field identifier

INTEGER *ipbmin* = Value which represents an empty progress bar (0-65535)

INTEGER *ipbmax* = Value which represents a full progress bar (0-65535)

Effect

Defines the range for a progress bar. The default range is 0-100. If the progress bar is in a modeless or semi-modeless dialog which is currently visible, the display is updated immediately. Max/min values must be in the range 0-65535. Values outside this range are ignored and generate error code 1013.

Example

```
See WDialogPutProgressBar
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrProgressRange`	(1013)	Invalid range

WDialogSetField Subroutine

Description

Move input cursor to a specific field

Syntax

WDialogSetField(*ifield*,*ipos*)

Arguments

INTEGER *ifield* = Field identifier

INTEGER, OPTIONAL *ipos* = Initial cursor position (string or combobox)

Effect

Forces *ifield* to become the current input field in the current dialog. The dialog must have been selected (by `WDialogLoad` or `WDialogSelect`) and it must be currently visible (i.e. it must be a modeless or semi-modeless dialog).

If *ifield* is a string or numeric field or a combo box menu with an enterable string then the initial cursor position can also be specified. If *ipos* is present then the cursor is placed before the specified character. If an initial cursor position is not specified then the cursor is placed at the end of the string, for single line strings and numeric fields. For multiline strings the cursor is placed at the start of the string by default.

Overuse of this routine is discouraged. However, it is an important control where on-the-fly field validation is performed.

The initially selected input field for a dialog which is not combined with a window, can be set in the call to `WDialogShow` which displays the dialog.

Example

```
! Force an important value to be entered in a modeless dialog.
CALL WMessage(ITYPE,MESSAGE)
SELECT CASE (ITYPE)
  CASE (FieldChanged)
    IF (MESSAGE%VALUE1==IDF_IMPORTANT) THEN
        CALL WDialogGetStringLength(IDF_IMPORTANT,LENGTH)
        IF (LENGTH==0) THEN
            CALL WDialogSetField(IDF_IMPORTANT)
        END IF
    END IF
END SELECT
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogSelect Subroutine

Description

Select the current dialog.

Syntax

WDialogSelect(*idialog*)

Arguments

INTEGER *idialog* = Identifier of dialog to select

Effect

Selects the current dialog. This is the dialog which all W*interacter* field manipulation routines subsequently operate on. Calling this routine does not cause the dialog to appear (see `WDialogShow`). The specified dialog must already be loaded (see `WDialogLoad`).

Example

```
CALL WDialogLoad(IDD_DIALOG1) ! Load several dialogs
CALL WDialogLoad(IDD_DIALOG2)
CALL WDialogLoad(IDD_DIALOG3)
CALL WDialogSelect(IDD_DIALOG2)
CALL WDialogShow(ITYPE=Modal) ! Will show 2nd dialog
```

Errors

`ErrSelDialog`	(1008)	Unable to select specified dialog

WDialogShow Subroutine

Description

Display the currently selected dialog.

Syntax

WDialogShow(*ixpos*,*iypos*,*ifield*,*itype*)

Arguments

INTEGER *ixpos* = X Co-ordinate of top left corner of dialog (-1 or omit to center)
(Child dialogs: W*interacter* units, Popup dialogs : Pixels)

INTEGER *iypos* = Y Co-ordinate of top left corner of dialog (-1 or omit to center)
(Child dialogs: W*interacter* units, Popup dialogs : Pixels)

INTEGER, OPTIONAL *ifield* = Identifier of initial field to edit
(0 for default: 1st field with `WS_TABSTOP` style)

INTEGER, OPTIONAL *itype* = Dialog type for popup dialogs

Modal	(1) : Modal dialog
Modeless	(2) : Modeless dialog (default)
Semi-modeless	(3) : Semi-modeless dialog

Effect

Displays the currently selected dialog, allowing the user to edit its contents. The dialog must have been loaded previously using `WDialogLoad`. If the current dialog selection has changed since the dialog was loaded, the dialog to be displayed must be reselected by calling `WDialogSelect`. If the specified dialog is already active, the position and type parameters will be ignored. Instead the dialog will simply be brought to the front and made active.

Both child and popup dialogs are supported (this is determined by a setting in the resource file). Child dialogs are restricted to the program's root window. Popup dialogs can be moved anywhere on screen. The initial position of a child dialog is determined in W*interacter* units (i.e. 0-9999), relative to the top corner of the root window. Popup dialog positions are specified in pixels relative to the top corner of the *screen*. Only popup dialogs can be displayed if the root window is hidden (i.e. if `HideParent` was specified in the call to `WindowOpen`). Attempting to show a child dialog will fail and generate an error code. To centre a dialog, either horizontally or vertically, omit the corresponding argument or set it to -1.

ifield specifies the initially highlighted data entry field. This argument has the `OPTIONAL` attribute. If it is zero or omitted, the initial field will be the first editable field in the dialog.

itype specifies the dialog type:

`Modal` : A modal dialog is one which blocks all other input to the program until the user terminates the dialog (e.g. by clicking on OK). `WDialogShow` will not return until the user has terminated the dialog at which point the dialog window is automatically removed from the screen. The exit field and terminating button identifiers are available via `WinfoDialog`.

`Modeless`: A modeless dialog does not block input. `WDialogShow` returns as soon as the dialog has been displayed. The dialog remains on screen until `WDialogHide` or `WDialogUnload` is called. Any push-button clicks in the dialog will be reported via `WMessage` as a `PushButton` message. The current field number when the button was pressed will be returned via the associated *value* argument. It is not possible to display a modeless dialog while a semi-modeless dialog is already active. If any semi-modeless dialogs are active and modeless is requested then the dialog will be displayed as semi-modeless and an error code set.

`SemiModeless` : A semi-modeless dialog is a hybrid of the other two dialog types. It appears modeless to the program but modal to the user. It blocks user input to all other dialogs and windows, but `WDialogShow` returns as soon as the dialog has been displayed. The dialog remains on screen until `WDialogHide` or `WDialogShow` is called. Any push-button clicks in the dialog will be reported via `WMessage` as a `PushButton` message. The current field number when the button was pressed will be returned via the associated *value* argument.

The *itype* argument is `OPTIONAL` and need only be specified for popup dialogs. If not specified, the default type is Modeless.

Further dialogs may be activated while a semi-modeless dialog is in use. However, these dialogs must be modal or semi-modeless. If a modeless dialog is requested, then a semi-modeless dialog will be displayed instead. Error code 1014 will be set in this case.

Portability notes

Windows : Child dialogs are always modeless, since Windows does not allow modal or semi-modeless child dialogs.

X Windows: All dialogs are 'popup' dialogs, regardless of the resource file setting. Dialogs which are specified as 'child' dialogs are still positioned according to the documented child dialog logic and are treated as modeless regardless of *itype*. This ensures consistency with the Windows implementation.

Example

```
CALL WDialogLoad(IDD_DIALOG1)    ! Display dialog1 as
CALL WDialogShow()               ! a centered modeless dialog
CALL WDialogLoad(IDD_DIALOG2)    ! Display dialog2 as a
CALL WDialogShow(ITYPE=Modal)    ! centered modal dialog
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrDialogType`	(1014)	Invalid dialog type
`ErrRootHidden`	(1016)	Child dialogs cannot be shown when root window hidden
`ErrModalPixmapLow`	(1041)	X server pixmap resources are low
`ErrModalPixmapExh`	(1042)	X server pixmap resources are exhausted

WDialogUnload Subroutine

Description

Remove dialog from screen and memory.

Syntax

WDialogUnload(*iaction*)

Arguments

INTEGER, OPTIONAL *iaction* = Not used in Starter Kit

Effect

Removes the currently selected dialog from the screen, if it is currently visible. It then removes the dialog and its associated data from memory, releasing all associated resources. The same dialog can be reloaded later using `WDialogLoad`, though this will reset all the dialog parameters to their initial values as set in the program resource.

Example

```
CALL WDialogLoad(IDD_ABOUT)
CALL WDialogShow(ITYPE=Modal)
CALL WDialogUnload()
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier

Group DM(2): Assign/Retrieve Field Contents

The routines in this group assign and retrieve the contents of individual dialog fields via a set of `WDialogPutXXX` and `WDialogGetXXX` routines. The 'Put' routines would normally be called *before* `WDialogShow`. The resulting field contents can then be retrieved from a dialog using the 'Get' equivalents *after* `WDialogShow`.

Where a field is part of an already visible modeless dialog, the various 'Put' routines update the on-screen contents of the dialog immediately.

All routines in this group affect the currently selected dialog, as set by `WDialogLoad` or `WDialogSelect`. The error flag is therefore set to `ErrCurDialog` if an attempt is made to assign or retrieve the contents of a field when no dialog is currently selected. Bear in mind that `WDialogUnload` causes the current dialog to become undefined. `WDialogSelect` must be called in this case to reselect the current dialog.

WDialogGetCheckBox Subroutine

Description

Get state of a dialog check box.

Syntax

WDialogGetCheckBox(*ifield*,*istate*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

INTEGER *istate* = Returned check box state (Unchecked (0): Clear, Checked (1): Set)

Effect

Gets the state of a check box field in the current dialog.

Example

```
LOGICAL :: UseColor
CALL WDialogGetCheckBox(IDF_COLOR,ICHECKED)
UseColor = ICHECKED==1
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogGetMenu Subroutine

Description

Get a value from a dialog menu field.

Syntax

WDialogGetMenu(*ifield*,*ioption*,*cvalue*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

INTEGER *ioption* = Number of selected option or 0 if user entered a string in a combo box

or

INTEGER *ioption(:)* = Array of binary flags indicating state of each option

CHARACTER, OPTIONAL *cvalue* = Entered string

Effect

Gets a value (or values) from a menu field in the current dialog.

For combo boxes and single-selection list boxes, *ioption* returns the currently selected item number from the pre-defined list held in this field. If this returns 0, the user entered a non-matching string in the enterable field of a combo box menu field.

For mutiple and extended selection list boxes *ioption* returns the state of each option. 1 is returned for selected options, 0 is returned for unselected options.

If *ioption* is not an array for a multiple or extended selection list box then an error code is set. The same error code will be set if an array is specified for a single selection menu.

If *cvalue* is supplied, the string corresponding to the currently selected option is returned. This is mainly useful when *ioption* returns zero, i.e. when the user has entered a string in a combo box. *cvalue* returns the user supplied string in this case. *cvalue* should not be specified for multiple or extended selection list boxes.

If the menu field contents are undefined and there is no user entered string value, *ioption* is returned as -999 and an error code is set. Multiple and extended selection list boxes simply return 0 for each element of *ioption*.

Example

```
CHARACTER (LEN=80) :: USERSTRING
CALL WDialogGetMenu(IDF_COMBO1,IOPTION,USERSTRING)
IF (IOPTION==0) THEN
    :            ! Process user option.
ELSE IF (IOPTION>0) THEN
    :            ! Process standard options
ELSE
    :            ! Combo box contents are undefined
END IF
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrFieldUndefined`	(1010)	Field value is undefined

WDialogGetRadioButton Subroutine

Description

Gets a radio-button group value

Syntax

WDialogGetRadioButton(*ifield*,*iset*)

Arguments

INTEGER *ifield* = Field identifier of a radio button as set in resource file

INTEGER *iset* = Position of currently set radio button within the group which contains button *ifield*. (-999 if none set)

Gets the position of the currently selected item in the group of radio buttons in the current dialog which contains button *ifield*. In other words, rather than returning the state of the specified individual radio button, this routine identifies which radio button is on in the group which *ifield* belongs to. Note that *iset* is a positional value. So, for example, if there are 5 radio buttons in a group, *iset* will normally be in the range 1 to 5.

If all the radio buttons are clear (because the initial button state was not defined in the resource file) *iset* is returned as -999 and an error code is set.

Example

```
CALL WDialogGetRadioButton(IDF_RADIO1,IPOSITION)
SELECT CASE (IPOSITION)
   !
END SELECT
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrFieldUndefined`	(1010)	Field value is undefined

WDialogGetString Subroutine

Description

Get a string from a dialog string field.

Syntax

WDialogGetString(*ifield*,*cvalue*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

CHARACTER *cvalue* = Returned character value (blank if undefined)

Effect

Gets a string from a field of almost any type. While this routine will normally just be used to retrieve the contents of ordinary string fields, it also provides access to the contents of just about any field which has an associated string value. This includes push buttons, check-boxes and radio-buttons.

Portability notes

Windows : If *ifield* specifies a multi-line edit control, the returned string will contain embedded carriage return/line-feed pairs (`CHAR(13)//CHAR(10)`) indicating line ends.

X Windows: If *ifield* specifies a multi-line edit control, the returned string will contain embedded line-feed characters (`CHAR(10)`) indicating line ends.

Example

```
CHARACTER (LEN=20) :: TEXT,BUT
CALL WDialogGetString(IDF_STRING,TEXT) ! Contents of string field
CALL WDialogGetString(IDF_BUTTON,BUT)  ! Caption of a push-button
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogPutCheckBox Subroutine

Description
Set the state of a dialog check box.

Syntax
WDialogPutCheckBox(*ifield*,*istate*)

Arguments
INTEGER *ifield* = Field identifier as set in resource file

INTEGER *istate* = Check box field state (Unchecked (0): Clear, Checked (1): Set)

Effect
Sets the state of a check box in the current dialog. To set the associated check box string, call `WDialogPutString`.

Example
```
   CALL WDialogPutCheckBox(IDF_CHECK,Checked)
```

Errors

`ErrCurDialog`	(1005)	No current dialog
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogPutImage Subroutine

Description
Change the bitmap/icon displayed in a field.

Syntax
WDialogPutImage(*ifield*,*imageid*,*itype*)

Arguments
INTEGER *ifield* = Field identifier as set in resource file

INTEGER *imageid* = Bitmap or icon identifier as set in resource file

INTEGER, OPTIONAL *itype* = Image type

1 : Bitmap (default)

2 : Icon

Changes the bitmap or icon which is displayed in the specified field in the current dialog. The bitmap or icon must exist in the program resource and the field specified by *ifield* must be one of the following types :

Picture/frame

Push-button

Group-box

Check-box

Radio button

If *itype* is omitted, a bitmap resource is assumed. The significant difference between a bitmap and an icon is that the latter allows for transparent pixels.

Example

```
CALL WDialogPutImage(IDF_PICTURE,ID_BITMAP)
```

Errors

`ErrCurDialog`	(1005)	No dialogs currently loaded
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrImageNum`	(1015)	Invalid bitmap/icon identifier

WDialogPutMenu Subroutine

Description

Set the contents of a dialog menu field.

Syntax

WDialogPutMenu(*ifield*,*option*,*maxopt*, *ioption*,*cvalue*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

CHARACTER *option*(:) = Array of menu options

INTEGER *maxopt* = Number of menu options

INTEGER *ioption* = Number of initially highlighted option

or

INTEGER *ioption(:)* = Array of binary flags indicating options to highlight

CHARACTER, OPTIONAL *cvalue* = User modifiable string

Effect

Sets the contents of the specified menu field in the current dialog. An array of *maxopt* option strings should be supplied. The maximum number of options which can be added to a menu field is 32767. In practice the lengths of the option strings are limited only by the width of the field.

For combo boxes and single selection list boxes *ioption* specifies the item number which should be highlighted initially. This will normally be in the range 1 to *maxopt*.

For multiple and extended selection list boxes *ioption* specifies the state of each option. Specify 1 to highlight an option and 0 to not highlight an option.

If *ioption* is not specified as an array for a multiple or extended selection list box then an error code is set. The same error code will be set if an array is specified for a single selection menu.

If *ifield* specifies a combo box with a user enterable field, *cvalue* can contain the initial user modifiable value. In this case, *ivalue* should be specified as zero. Otherwise, *cvalue* can be omitted since it has the `OPTIONAL` attribute. *cvalue* should not be specified for multiple or extended selection list boxes.

If *ifield* specifies a list box, option strings can contain tab characters (ASCII 9) to vertically align sub-strings within menu items.

Example

```
INTEGER, PARAMETER  :: NFRUIT = 4
CHARACTER (LEN=7), DIMENSION (NFRUIT)  :: FRUIT = &
          (/'Apples','Oranges','Pears','Bananas'/)
IFRUIT = 1
CALL WDialogPutMenu(IDF_FRUIT,FRUIT,NFRUIT,IFRUIT)
```

Errors

`ErrCurDialog`	(1005)	No dialogs currently loaded
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrOptionNum`	(1011)	Option number out of range

WDialogPutOption Subroutine

Description

Set the selected option in a dialog menu field.

Syntax

WDialogPutOption(*ifield*,*ioption*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

INTEGER *ioption* = Menu option number to highlight (0=none, combo boxes only)

or

INTEGER *ioption(:)* = Array of binary flags indicating options to select

Effect

Sets the currently selected option or options in a menu field in the current dialog. This is the same as the *ioption* argument to `WDialogPutMenu`. The option number can be zero if *ifield* specifies a combo box, in which case none of the predefined menu items will be highlighted. When used with a multiple or extended selection list box *ioption* should contain the same number of entries as there are options in the menu.

Example

```
CALL WDialogPutOption(IDF_FRUIT,IAPPLE)
```

Errors

`ErrCurDialog`	(1005)	No dialogs currently loaded
`ErrFieldNum`	(1006)	Invalid field identifier
`ErrOptionNum`	(1011)	Option number out of range

WDialogPutProgressBar Subroutine

Description

Set the value of a progress bar

Syntax

WDialogPutProgressBar(*ifield*,*ivalue*,*method*)

Arguments

INTEGER *ifield* = Field identifier

INTEGER *ivalue* = Absolute or relative progress bar value

INTEGER, OPTIONAL *method* = Interpretation of *ivalue*.

Absolute	(0) : Set progress bar to specified value
Relative	(1) : Change progress bar by specified value

Effect

Sets the value of a progress bar field. By default, *ivalue* is treated as an absolute value. By specifying an optional *method* argument of 1, the progress bar value can be amended by an incremental value (either positive or negative).

Out of range values are truncated to the appropriate min/max value.

Example

```
CALL WDialogLoad(ID_DIALOG1)
CALL WDialogShow(ITYPE=SemiModeless)
!
CALL WDialogRangeProgressBar(IDF_PROG1,1,MAXITER)
DO ITER = 1,MAXITER
  ! Calculation in here
  CALL WDialogPutProgressBar(IDF_PROG1,ITER,Absolute)
END DO
CALL WDialogHide()
CALL WDialogUnload()
```

Errors

`ErrCurDialog`	(1005)	No dialogs currently loaded
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogPutRadioButton Subroutine

Description

Set the state of a radio button group.

Syntax

WDialogPutRadioButton(*ifield*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

Enables the specified radio button. All other radio buttons in the group to which it belongs are automatically unselected.

Example

```
! Set Radio button number 2, clear all others in the same group
CALL WDialogPutRadioButton(IDF_RADIO2)
```

Errors

`ErrCurDialog`	(1005)	No dialogs currently loaded
`ErrFieldNum`	(1006)	Invalid field identifier

WDialogPutString Subroutine

Description

Set the value of a dialog string.

Syntax

WDialogPutString(*ifield*,*cvalue*)

Arguments

INTEGER *ifield* = Field identifier as set in resource file

CHARACTER *cvalue* = Character string to be placed in field

Effect

Sets the string of a field of almost any type. This routine can be used to set the contents of ordinary string fields or any field which has an associated string value. This includes push buttons, check boxes, radio-buttons and multi-line edit controls.

If *ifield* specifies a multi-line edit control, the supplied string should contain embedded carriage return/line-feed pairs (i.e. `CHAR(13)//CHAR(10)`) to indicate line ends.

On labels, group-boxes, push-buttons, check-boxes and radio-buttons an & prefix character will normally be available. The character after the & then acts as a short-cut when used with the Alt key. For labels and group boxes this shortcut either moves to the next entry field or acts as if the next push-button was pressed. For push-buttons, check-boxes and radio-buttons the shortcut acts as if the field was clicked. Specify two ampersands (i.e. &&) to actually display a single ampersand. On label fields, use of the & prefix can be disabled in the resource editor via the field Style dialog (enable the 'No prefix' check box). In this case ampersands are displayed without interpretation.

Portability notes

Windows : Multi-line edit controls can hold a maximum of 64k of text (maximum 1k per line). Ordinary string fields (i.e. not multi-line) are limited to a maximum of 32k of text. These limits are subject to Windows sucessfully allocating sufficient storage for the text. Windows 9x/Me will sometimes fail to allocate sufficient storage for large amounts of text, even when several megabytes of memory are available. Tab characters (`CHAR(9)`) can be embedded to vertically align text at tab stops in multi-line edit controls.

X Windows: There are no specific limits on string length.

Example

```
CALL WDialogPutString(IDF_STRING,'Some Text')
! now a push button field
CALL WDialogPutString(IDF_BUTTON,'Press Me')
```

Errors

`ErrCurDialog`	(1005)	No dialogs currently loaded
`ErrFieldNum`	(1006)	Invalid field identifier

Group CD: Common Dialogs

The file-selector and message-box ’common dialogs' are supported via the routines in this group. The dialogs displayed by these routines are modal.

WMessageBox Subroutine

Description

Display a standard Windows message box.

Syntax

WMessageBox(*ibutton*,*icon*,*idefbut*,*message*,*title*)

Arguments

INTEGER *ibutton* = The type of buttons to be displayed

Table 4: Common Dialog Buttons

Name	No.	Button(s)
OKOnly	0	OK button
OKCancel	1	OK and Cancel buttons
RetryCancel	2	Retry and Cancel buttons
YesNo	3	Yes and No buttons
YesNoCancel	4	Yes, No and Cancel buttons
RetryAbortIgnore	5	Retry/Abort/Ignore buttons

INTEGER *icon* = The type of icon to be displayed

Table 5: Common Dialog Icons

Name	No.	Icon
NoIcon	0	No icon
StopIcon	1	Stop icon
QuestionIcon	2	Question mark icon
ExclamationIcon	3	Exclamation mark icon
InformationIcon	4	Information icon

INTEGER *idefbut* = Default highlighted button:

Table 6: Common Dialog Button Numbers

Name	No.	Highlighted Button
CommonCancel	0	Cancel
CommonIgnore	0	Ignore
CommonOK	1	OK
CommonOpen	1	Open

Table 6: Common Dialog Button Numbers

Name	No.	Highlighted Button
CommonYes	1	Yes
CommonRetry	1	Retry
CommonAbort	2	Abort
CommonNo	2	No

CHARACTER *message* = Message box text

CHARACTER *title* = Message box title

Effect

Displays a standard message box consisting of a message and up to three push buttons. Since many programs require only simple confirmations from the user this routine can help reduce the number of required dialog resources.

ibutton selects the number and type of buttons the message box will contain.

icon selects the pre-defined icon that appears in the message box beside the text.

idefbut specifies which button will be highlighted when the message box is first opened. This value follows the same numbering scheme as the exit button code returned by `WInfoDialog`.

message should contain the text to be displayed in the message box. The supplied text is not automatically word wrapped. The message string must contain carriage returns (i.e. `CHAR(13)`) to break the lines at the appropriate places. The message can be blank if required.

title should contain the message box title. Again, this can be blank.

The button pressed to exit from this routine is available via `WInfoDialog(4)`. This uses the same numbering scheme as *idefbut*.

Example

```
CALL WMessageBox(YesNo, QuestionIcon, 1, &
                        'Another message box ?', 'Question')
IF (WInfoDialog(4)==1) THEN
    CALL WMessageBox(OKOnly, InformationIcon, 1,            &
              'This is how to split your text'//CHAR(13)// &
              'over several lines.','Information')
END IF
```

Errors

`ErrCommonDlg`	(1004)	Common dialog function returned an error. Actual system error code available via `InfoError(3)`
`ErrModalPixmapLow`	(1041)	X server pixmap resources are low
`ErrModalPixmapExh`	(1042)	X server pixmap resources are exhausted

WSelectFile Subroutine

Description

Choose a file using the standard file selector dialog

Syntax

WSelectFile(*filterstr*,*iflags*,*filedir*,*title*,*iftype*)

Arguments

CHARACTER *filterstr* = Filter strings

INTEGER *iflags* = Dialog settings. Sum of:

	LoadDialog	(0)	) Load or
or	SaveDialog	(1)	) Save dialog
	PromptOn	(2)	Enable prompting
	NonExPath	(4)	Allow non existent paths
	DirChange	(8)	Allow directory change
	MultiFile	(16)	Allow multiple file selection
	AppendExt	(32)	Append extension

CHARACTER *filedir* = Entry : Initial directory path + filename

Exit : Final directory path + filename

CHARACTER, OPTIONAL *title* = Dialog title

INTEGER, OPTIONAL *iftype* = File type on entry and exit

Effect

Prompts for a filename using the standard file selector. The dialog allows the user to enter a new file name or path. It is the responsibility of the caller to create these if necessary.

The dialog is displayed slightly below the top left corner of the window or dialog which currently has the input focus. If the file selector is displayed relative to the root window and the root window is hidden, you can still specify the window position to `WindowOpen` to determine where the file selector dialog will appear.

filterstr defines a list of filter pairs. This will normally consist of program type descriptions and corresponding file name match strings. They are used by the file selector to restrict the file types which are offered in the dialog. Each description is separated from its corresponding match string by a vertical bar (|). For example :

'Windows Bitmap|*.bmp|Paintbrush Image|*.pcx|'

This would add the filters *.bmp and *.pcx to the file type combo box. The first filter in the list is used initially. Note that the final "|" in the filter string is required. Multiple filters can be attached to one filter string by separating them with semi-colons, e.g.

'Fortran files|*.f90;*.for;*.f|'

Pass *filterstr* as a blank to match all files.

[Note : Alternatively, the filter strings argument can be specified as an integer resource file id, documented in early versions as *ifilterid*. This should be the identifier of a string table entry in the resource script. Use of this calling interface is supported for backwards compatibility, but its use in new software is strongly discouraged. In particular, this obsolete interface will not work with the X Windows implementation.]

iflags provides control over the exact behaviour of the file selector, by summing together the following settings :

`LoadDialog` selects a standard open-file (i.e. Load) dialog. `SaveDialog` selects a Save file dialog.

Add `PromptOn` to *iflags* to enable prompting. When prompts are enabled, the setting of bit 0 determines the actual type of warning prompts displayed. For Load dialogs the user will be prompted if they specify a file or directory that does not exist. For Save dialogs the user will be prompted if they try to save to a file that already exists.

Add `NonExPath` to *iflags*, if non-existent paths (i.e. non-existent directories) are to be allowed. Otherwise, only file names with valid directory names can be returned from the dialog.

Add `DirChange` to *iflags* to allow the current directory to be updated. Enabling this option causes the current directory to be updated dependent on which directories the user browses in the file selector. If this option is not selected, `WSelectFile` saves and restores the current directory across the call to the file selector dialog.

Add `MultiFile` to *iflags* to allow multiple file selection. The Shift or Ctrl keys can be used in combination with mouse clicks to select groups of files or multiple single files respectively. Multiple file selection is only available when `LoadDialog` is specified.

Add `AppendExt` to *iflags* to automatically add an extension to the selected filename. The extension added depends on the filter string chosen by the user. If the selected filter string contains more than one extension then the first extension is used. The rules used to determine whether the user has actually specified an extension, or has simply entered a filename which includes a full-stop/period, are slightly complex. Supposing a filename such as `myfile.xyz` is entered, a file extension will still be added to this if :

a) `myfile.xyz` does not exist in the selected directory

and

b) `xyz` is not one of the file extensions in the filter string table

Add `MustExist` to *iflags* to allow only the names of existing files to be selected in an open file dialog. This flag has no effect when `SaveDialog` is specified. If this flag is specified then `PromptOn` and `NonExPath` will have no effect, the user will alway be prompted if they enter a non-existent path or filename.

On entry *filedir* will contain the directory path and/or filename to use as the initial default. Specify a trailing directory separator (\ under Windows or / under Linux) if *filedir* identifies a directory only. If *filedir* is blank the current directory path is used with no default file name. On exit the final path and file name will be returned in *filedir*, if a single file was selected. This will be left unchanged if the user pressed the Cancel button or closed the dialog window.

If multiple file selection is enabled and multiple files are chosen, *filedir* will contain the directory name (without a trailing directory separator character) and a list of file names. The directory and file names will be separated by nulls (i.e. `CHAR(0)`). The calling program is responsible for extracting the individual file names.

title specifies the dialog window title, e.g. 'Load a Data File'. If this is blank or omitted, a default title of 'Select File' is used.

The optional *iftype* argument allows you to determine which of the supplied filter strings appears in the 'Files of type' field initially. This is an index value, so a value of 2 would select the second file type in the filter string list (e.g. "Paintbrush Image" in our earlier example). If the *iftype* argument is specified, it will also then return the index of the file type which the user selected on exit. If *iftype* is omitted, the first file type in the filter string list is used and no result is returned.

The exit button/result from this routine will be available via `WInfoDialog(4)`.

Portability notes

Microsoft Windows: Under NT 4.0, multiple file selection requires one of Microsoft's service packs to be installed. A base NT 4.0 system with no service packs installed appears to be limited to a maximum return string length of 241 characters. Service Pack 3 definitely fixes this problem. SP1 or 2 may also be suitable but this is untested.

X Windows: The supplied filter string is only used it it specifies a single file type, otherwise all files are shown. The `MultiFile` and `AppendExt` flags are not currently supported.

Example

```
CHARACTER (LEN=255)  FILENAME
           :
FILENAME = 'C:\PICTURES'         !  Default file path
IFLAGS = LoadDialog + PromptOn   !  Select load dialog
CALL WSelectFile( &
       'Fortran 90|*.f90|Fortran 77|*.for|', &
       IFLAGS, FILENAME, 'Select source file')
IF (WInfoDialog(4)==CommonOK) CALL process_file(FILENAME)
```

Errors

`ErrCommonDlg`	(1004)	Common dialog function returned an error. Actual system error code available via `InfoError(3)`
`ErrModalPixmapLow`	(1041)	X server pixmap resources are low
`ErrModalPixmapExh`	(1042)	X server pixmap resources are exhausted

12 High Resolution Graphics

The routines described in this chapter are divided into 4 groups:

GG	General Graphics	(Drawable selection, pixel interrogation, units)
GS	Graphics Style selection	(color, line type, fill style)
GD	Graphics Drawing and Movement	(Line/fill primitives)
GT	Graphics Text	(Text primitives)

In addition, graphics related information functions are provided in the IF group.

Starting and Finishing

W*interacter*'s graphics are automatically initialized by `WindowOpen`. Graphics are always directed to the currently selected root or child window as set by `WindowOpen`, `WindowOpenChild` or `WindowSelect`. Graphics remain available until `WindowClose` is called.

Target Drawable

W*i*SK can generate graphics output to either a window or a dialog field, referred to as the current 'drawable'. The initial target drawable is the current window, as set by `WindowOpen`, `WindowOpenChild` or `WindowSelect`. This can be changed by calling `IGrSelect`.

Co-ordinate System

The W*interacter* graphics co-ordinate system is user-definable and is not related to the physical resolution of the output device, making graphics based programs device independent. The area of the output drawable to be used for the graphics display can also be defined. The `IGrArea` and `IGrUnits` routines control the main graphics area and co-ordinate system respectively. Calls to routines in other groups, such as `IGrLineTo`, `WGrTextString`, etc. all use `(x,y)` co-ordinates defined in terms of the range set by the call to `IGrUnits`. So if `IGrUnits` sets the min and max `X` values as `0-1000` and the min and max `Y` values as `0-500`, all co-ordinate values should be in these ranges too. Attempts to draw outside this area will be clipped at the edge of the graphics area. By default `(0,0)` is the bottom left corner of the graphics area. Initially, the main graphics area is set to the full drawable.

Lahey Graphics Emulation

Legacy code which was written for the Lahey Video Graphics Library can be rebuilt with W*interacter* using the emulation code in `lvgl.f90`. Refer to the on-line help file (`WiSK.HLP` or `wisk.htm`) for details.

OpenGL Graphics

The OpenGL graphics interface is accessible in W*interacter* based programs, enabled via `WglSelect` in the MI group. A selection of demonstration programs are supplied in W*i*SK's `OpenGL` sub-directory. See the OpenGL pages under "Graphics Interfaces" in the on-line help file (`WiSK.HLP` or `wisk.htm`) for further information.

Group GG: General Graphics

This group provides various graphics routines which don't naturally belong in the graphics other subroutine groups. These include certain house keeping routines which are fundamental to using W*interacter* graphics routines, namely `IGrArea` and `IGrUnits` which define the size of the graphics area and the co-ordinates within that area.

The target drawable is determined by `IGrSelect`. This can be either a window or a dialog field.

W*interacter*'s graphics can be re-initialised via `IGrInit`.

The graphics area can be cleared by `IGrAreaClear`.

Pixel colors in the current drawable can be interrogated using `IGrGetPixel`.

IGrArea Subroutine

Description

Define size of graphics area.

Syntax

IGrArea(*xleft*,*ylower*,*xright*,*yupper*)

Arguments

REAL *xleft* = Left limit of main graphics area (`0.0 <=` *xleft* `< 0`)

REAL *ylower*= Lower limit of main graphics area(`0.0 <=` *ylower* `<1.0`)

REAL *xright* = Right limit of main graphics area (`0.0 <` *xright* `<= 1.0`)

REAL *yupper*= Upper limit of main graphics area(`0.0 <` *yupper* `<= 1.0`)

Effect

Defines the area of the current drawable to be used by all following graphics commands. The full window is defined as being 1 unit high and 1 unit wide, so you should describe your area in values in the range `0.0 - 1.0` as shown above. When you call `IGrUnits`, that then defines the co-ordinate system to be used within the area defined by `IGrArea`. The default values for both the `IGrArea` and `IGrUnits` ranges, are `0.0` to `1.0` occupying the whole of the graphics screen. The current graphics area dimensions can be interrogated via the `Info-Graphics` function.

`IGrArea` is particularly useful when you wish to rescale a graphics image without changing your co-ordinate system or any other parameters. In the example below, a full screen display is reduced to a quarter size, at the top right of the screen, by a single call to `IGrArea`.

It is important to appreciate that if you set a graphics area in which the sides are no longer equal (e.g., `0.5` high and `1.0` wide) then regular shapes will be distorted accordingly. For example, circles become elliptical, squares become rectangular and so on.

Example

```
CALL IGrArea(0.0,0.0,1.0,1.0)
CALL MYGRAF()
CALL IGrArea(0.5,0.5,1.0,1.0)
CALL MYGRAF()
```

Errors

`ErrBadArea`	`(44)`	Invalid `X` and/or `Y` range. Range reset to `0-1`.

IGrAreaClear Subroutine

Description

Clear the current graphics screen area.

Syntax

IGrAreaClear()

Effect

Clears the current main graphics area as defined by `IGrArea`, to the current background color. Part of the graphics window can therefore be cleared without affecting the rest of the window.

When the graphics area is defined to be `0.0-1.0` in both `x` and `y` directions, the whole window is cleared.

Example

```
   CALL IGrArea(0.05,0.05,0.4,0.4)
   CALL IGrAreaClear()
   CALL MYGRAF()
```

IGrGetPixel Function

Description
Read a screen pixel color value

Syntax
INTEGER IGrGetPixel(*xpos*,*ypos*)

Arguments
REAL *xpos* = X co-ordinate

REAL *ypos* = Y co-ordinate

Effect
Returns the color of the specified co-ordinate in the current drawable, as a 24-bit color value. Individual color components can be extracted using `WRGBsplit`. The (x,y) co-ordinate should be expressed in user units as set via `IGrUnits`. If the specified co-ordinate lies outside the graphics area, -1 is returned.

IGrInit Subroutine

Description
Re-initialize graphics output.

Syntax
IGrInit(*type*,*nx*,*ny*,*nc*)

Arguments
CHARACTER *type* = Type of output. Leave blank in Starter Kit implementation.

INTEGER, OPTIONAL *nx* = *INTERACTER* compatibility argument

INTEGER, OPTIONAL *ny* = *INTERACTER* compatibility argument

INTEGER, OPTIONAL *nc* = *INTERACTER* compatibility argument

Effect

Re-initializes graphics output. This routine is called by `WindowOpen`, so it should not normally be necessary to call it again unless the whole of the graphics system needs to be reset to its default state.

W*interacter*'s internal graphics are reinitialized to the following defaults:

- Calls `IGrArea` with parameters (0.0,0.0,1.0,1.0)
- Calls `IGrUnits` with parameters (0.0,0.0,1.0,1.0)
- Sets current plotting position to (0.0,0.0)
- Sets fill pattern to none
- Selects the 8-bit color model
- Sets plotting color to 223 (black)
- Sets secondary color for mixed-color fills to 0 (white)
- Sets line type to solid
- Selects driver-specific Courier font, with all style attributes disabled
- Sets character size to width=.01333333 and height=.04.
- Resets the target drawable to the current window

The *nx*, *ny* and *nc* arguments are provided for compatibility with *INTERACTER*. They are not used in W*interacter* and have the `OPTIONAL` atrribute. They can be safely omitted in W*interacter*-specific code.The type argument should be specified as a blank in W*interacter* Starter Kit programs (this argument has meaning in the full version of W*interacter*).

IGrSelect Subroutine

Description

Select the target drawable for graphics output.

Syntax

IGrSelect(*itarget*,*ident*)

Arguments

INTEGER *itarget* = Target drawing surface:

INTEGER, OPTIONAL *ident* = Handle or identifier of target drawable:

Effect

Selects the target drawing surface ("drawable") for graphics output. The type and size of the currently selected drawable can be interrogated via `WInfoDrawable`. Output can be routed to a window or a dialog field:

itarget = `DrawWin` : By default, graphics output is routed to the current window. This option routes graphics output to the specified window. *ident* must specify a valid window handle, as returned by `WindowOpenChild`, or zero for the root window. If the window handle is omitted or an invalid window is specified the currently selected window becomes the target drawable. The root window is the target drawable at initialisation. Calling `IGrSelect` with itarget set to 1 calls `WindowSelect` internally. While output to a window is selected, `WindowSelect` can be called directly to update the target window for graphics output (`WindowSelect` has no effect on the target graphics drawable while output to a dialog is selected).

itarget = `DrawField` : To draw into a dialog field, specify the field identifier in *ident*. This must identify a field in the current dialog. In theory, this can be any type of field, but typically this feature is best used with label or picture/frame fields. If *ident* is zero or omitted, the whole of the current dialog becomes available for drawing (but see the Portability notes). Specifying an invalid field identifier is treated as an error and the output target returns to the current window. It should be noted that dialog fields drawn in this way must be maintained by the calling program. Normal dialog fields are repainted automatically, but it is the caller's responsibility to repaint 'user drawn' fields. The `Expose` message reported by `WMessage` allows for this possibility.

Portability notes

Windows: Drawing to the whole dialog (*itarget*=3 and *ident* omitted or zero) will cause graphics to overwrite any fields in the dialog window, by default. If the 'Clip Fields' option is selected (see the Dialog Properties dialog in the resource editor), graphics will not overwite field contents, effectively drawing to the 'background' of the dialog. While the latter behaviour is preferable, enabling 'Clip Fields' causes the background of any group boxes to become transparent. This in turn causes rear windows to become visible through the dialog. We therefore don't recommend whole-dialog drawing (and hence the use of Clip Fields) with dialogs which contain group boxes.

X Windows: When drawing to the whole dialog (*itarget*=3 and *ident* omitted or zero), graphics are always clipped by the current dialog fields, regardless of field type. The 'Clip Fields' option in the resource editor has no effect.

Example

```
CALL IGrSelect(DrawWin) ! Draw graph in root window
CALL MYGRAF()
CALL IGrSelect(DrawField,IDF_PIC) ! Draw same graph in a dialog
CALL MYGRAF()
```

Errors

`ErrBadTarget`	`(1019)`	Invalid window handle or field identifier

IGrUnits Subroutine

Description

Define plotting units to be used.

Syntax

IGrUnits(*xleft*,*ylower*,*xright*,*yupper*)

Arguments

REAL *xleft* = Lower `X` co-ordinate limit

REAL *ylower*= Lower `Y` co-ordinate limit

REAL *xright* = Upper `X` co-ordinate limit

REAL *yupper*= Upper `Y` co-ordinate limit

`IGrUnits` defines the plotting units (the 'user co-ordinate system') to be used when drawing in the main graphics area defined by `IGrArea`. The initial ranges are `0.0` to `1.0` on both axes. The example below shows how to plot values in the range `500-1000` on the `x` axis and values of `300-600` on the `y` axis. The current plotting units can be interrogated via the `InfoGraphics` function.

Selecting an invalid `X` or `Y` range, sets the limits for that axis to `0-1`.

Example

```
CALL IGrUnits(500.0,300.0,1000.0,600.0)
CALL IGrCircle(750.,450.,50.)
```

Errors

`ErrBadUnits (16)` Lower `X` or `Y` value is greater than or equal to upper `X` or `Y`

Group GS: Graphics Style Selection

This group controls the appearance of output from other graphics routines.

Probably the most commonly used routines in this group will be those which control color. The current graphics color is selected using `IGrColourN`. The number of available colors is display dependent, so W*interacter* uses a common 8-bit color numbering scheme on all devices, by default. This provides 256 nominal colors (a near equivalent is selected where fewer than 256 colors are available). Alternatively, some video modes support 24-bit color mode where colors are specified directly as an RGB value. `IGrColourN` can use either an 8-bit or a 24-bit color model, selectable via `IGrColourModel`.

In the 8-bit color model, color 0 is treated as the background color and all others as foreground. However, color 0 can still be selected as the current color for graphics operations, though it will only be visible if the operation takes place on top of some non-background color.

The 8-bit color palette (the relationship between the 256 color numbers in the 8-bit model and the actual colors displayed) can be redefined using `IGrPalette`. By default the background color is white. The palette can be reinitialised by calling `IGrPaletteInit`.

In addition to color control, line type, plot mode and fill style are selectable via `IGrLineType`, `IGrPlotMode` and `IGrFillPattern`.

IGrColourModel Subroutine

Description

Select 8-bit or 24-bit color model

Syntax

IGrColourModel(*nbits*)

Arguments

INTEGER *nbits* = Color model : 8 or 24 bits.

Effect

Selects the current color model used by `IGrColourN`. *nbits* should either be 8, for an 8-bit/256-color palette based color model, or 24 for a 24-bit RGB based model.

By default, an 8-bit model is used, where colors are specified as values in the range 0-255. These represent index values in a nominal 256 color palette. On devices which support fewer than 256 simultaneous colors, W*interacter* automatically uses a subset of this palette. Each of the 256 colors in the 8-bit palette can be redefined using a 24-bit RGB value via `IGrPalette`, but no more than 256 colors are available at one time.

When the 24-bit color model is selected, colors are specified to `IGrColourN` as an RGB value of the form *red*+256**green*+256*256**blue*. This eliminates the indirection enforced by the use of a palette and allows for a theoretical maximum of 16 million simultaneous colors. This color specification model does not necessarily require a device which supports full 24-bit color (e.g. it can be used successfully on a 16-bit color display). Where a target display does not support 24-bit/RGB color specification, W*interacter* identifies the nearest matching color in its own nominal 8-bit palette. It then uses the corresponding 8-bit color number as though that had been specified directly to `IGrColourN`. This allows the 24-bit color model to be used on all displays with no significant loss of generality.

The ability of the current screen driver to take advantage of 24-bit color specification is reported by `InfoGrScreen(42)`.

While it is generally advisable to select a single color model throughout an application, it is feasible to switch between color models as needed. Internally, W*interacter* stores the 'current' color both as an 8-bit color index and as a 24-bit RGB value, so changes in the current color model should be transparent to the underlying screen driver.

See `IGrColourN` for a further explanation of the implications of calling this routine.

Example

```
! select bright red in two different ways
      CALL IGrColourModel(8)
      CALL IGrColourN(31)
      CALL IGrColourModel(24)
      CALL IGrColourN(255)
```

IGrColourN Subroutine

Description

Select graphics color using a color number.

Syntax

IGrColourN(*ncolor*)

Arguments

INTEGER *ncolor* = color number:

8-bit color model	: 0-255
24-bit color model	: Red + Green*256 + Blue*256*256

Effect

Selects the graphics color for lines, points, text and fills, using a single color number. The meaning and valid range of *ncolor* depends on the color model selected by `IGrColourModel`:

8-bit Color model

By default, W*interacter* uses an 8-bit color numbering scheme based on a device independent palette of 256 colors. The same color numbers are used regardless of the actual number of colors available on the output device. W*interacter* performs an internal mapping between its device independent color scheme and the actual color numbers used by the current hardware. The 256 color numbers are organised into 16 groups, each consisting of 16 shades of a given color. On devices which support less than 256 colors, W*interacter* sub-divides the palette according to the number of available colors. For example, where only 16 colors are available, values of 16-31 all give bright red. Each color in the palette can be redefined via `IGrPalette`, giving a maximum of 256 simultaneously different colors selected from a theoretical palette of 16 million.

24-bit Color model

When the alternative 24-bit color model is selected, *ncolor* specifies, the exact combination of red, green and blue to be used, according to the formula shown earlier. Each of the red, green and blue components should be specified as a value in the range 0-255. This gives a theoretical maximum of 16 million simultaneously different colors. On devices where 24-bit color selection is not supported, W*interacter* identifies the nearest equivalent color in its 8-bit palette and treats the resulting color number as though it had been selected using the 8-bit color model.

8-bit versus 24-bit ?

So which of the above color models should a program use ? The 8-bit color model was devised for use with *INTERACTER* in the late 1980's when 24-bit color hardware was rare and/or expensive. Applications developed using *INTERACTER* or earlier releases of W*interacter* will therefore exclusively use the 8-bit model, so this is the default for compatibility reasons. However, modern hardware offers cheap access to 24-bit color (or 16-bit color which W*interacter* treats as logically equivalent). Use of the 24-bit model is therefore recommended in new development. This is particularly true, given that W*interacter* will automatically determine the equivalent 8-bit palette value to use if 24-bit color is not available.

More about the 8-bit Color Model

While the 24-bit color model may be preferable in new software, a lot of code will already exist which uses the default 8-bit model. The following notes therefore apply specifically to the 8-bit color model:

- Color zero is treated specially by W*interacter* as the background color. This can still be selected as the current graphics color, enabling you to draw or fill in one fore-ground color and then plot on top of that using the current background color.
- The default palette associated with the W*interacter* 8-bit color model is as follows (values are (r,g,b) triplets where maximum intensity = 255) :

Table 7: 256-Color Numbering Scheme Default Palette

Actual Color	Color #	256-Color Palette
White	0-15	(255,255,255) -> (195,195,195)
Light red	16-31	(195, 0, 0) -> (255, 0, 0)
Dark red	32-47	(131, 0, 0) -> (191, 0, 0)
Light yellow	48-63	(195,195, 0) -> (255,255, 0)
Dark yellow	64-79	(131,131, 0) -> (191,191, 0)
Light green	80-95	(0,195, 0) -> (0,255, 0)
Dark green	96-111	(0,131, 0) -> (0,191, 0)
Light cyan	112-127	(0,195,195) -> (0,255,255)
Dark cyan	128-143	(0,131,131) -> (0,191,191)
Light blue	144-159	(0, 0,195) -> (0, 0,255)
Dark blue	160-175	(0, 0,131) -> (0, 0,191)
Light magenta	176-191	(195, 0,195) -> (255, 0,255)
Dark magenta	192-207	(131, 0,131) -> (191, 0,191)
Black	208-223	(60, 60, 60) -> (0, 0, 0)
Dark gray	224-239	(124,124,124) -> (64, 64, 64)
Light gray	240-255	(191,191,191) -> (131,131,131)

In 16 or 8 color output, a subset of the above palette is used:

Table 8: 16 or 8 Color Palette

Actual Color	Color #	16-color palette	8-color palette
White	0-15	(255,255,255)	(255,255,255)
Light red	16-31	(255, 0, 0)	(255, 0, 0)
Dark red	32-47	(191, 0, 0)	(255, 0, 0)
Light yellow	48-63	(255,255, 0)	(255,255, 0)
Dark yellow	64-79	(191,191, 0)	(255,255, 0)
Light green	80-95	(0,255, 0)	(0,255, 0)
Dark green	96-111	(0,191, 0)	(0,255, 0)
Light cyan	112-127	(0,255,255)	(0,255,255)
Dark cyan	128-143	(0,191,191)	(0,255,255)
Light blue	144-159	(0, 0,255)	(0, 0,255)
Dark blue	160-175	(0, 0,191)	(0, 0,255)
Light magenta	176-191	(255, 0,255)	(255, 0,255)
Dark magenta	192-207	(191, 0,191)	(255, 0,255)
Black	208-223	(0, 0, 0)	(0, 0, 0)
Dark gray	224-239	(64, 64, 64)	(0, 0, 0)
Light gray	240-255	(191,191,191)	(255,255,255)

- On a 256 color screen, a 16-color palette is used as shown in the table. However, a larger palette of 32, 64 or 128 colors can optionally be used in such a video mode, as specified via the optional *ncol256* argument of `WindowOpen`. When a 32/16/128 color palette is selected, the default 256-color palette is sub-divided accordingly (e.g. in a 32-color palette, colors 0-7 are the same whereas in a 64 color palette only colors 0-3 are the same).
- Whatever 8-bit color number is used, `IGrColourN` has no effect on the actual color which is associated with that number. It simply sets the logical color number to be used by any following graphics operations. To redefine the association of displayed colors with logical colors you should use `IGrPalette`.
- Requesting a color number outside the range 0-255, in the 8-bit model, is ignored and an error code is set.

- The number of colors available in the 8-bit color model can be checked using `InfoGrScreen(30)`. The actual number of screen colors may be different and can be obtained via `WInfoScreen(3)`.

When the 24-bit color model is selected, but the current display does not allow 24-bit color selection (e.g. a 256 color screen), W*interacter* reverts internally to using the 8-bit color model and all of the above rules apply even though the 8-bit model was not explicitly requested. `InfoGrScreen(42)`reports the ability of the current screen driver to take advantage of 24-bit color specification.

Two consecutive calls to `IGrColourN` will select the colors to be used by mixed-color area fills (see `IGrFillPattern`) or opaque text (see `WGrTextFont`). The two most recently requested colors are available via `InfoGrScreen(34/35)`. The default graphics color at initialisation is black.

See the `col256` and `col24bit` demo programs.

The number of colors supported by W*interacter*'s 8-bit color model is related to the number of colors provided by the Windows video driver or X server, as follows:

Table 9: Windows colors

Video Driver or X server Colors	Size of 8-bit palette used by W*interacter*
2	2
16	8
256	16/32/64/128
32k/65k/16m	256

When the 24-bit color model is requested, W*interacter* will use the supplied RGB values directly on 15/16/24/32 bit color displays and will revert to its 8-bit model internally on 2/16/256 color displays.

Example

```
DO ICOL = 31,255,32
  CALL IGrColourN(ICOL)
  CALL IGrMoveTo(0.0,0.0)
  CALL IGrLineTo(0.5,REAL(ICOL))
END DO
```

Errors

`ErrBadcolor`	(42)	Unknown color number. Current color unchanged

IGrFillPattern Subroutine

Description

Define fill pattern (solid/mixed-colors/hatched).

Syntax

IGrFillPattern(*istyle*,*idense*,*iangle*)

Arguments

INTEGER *istyle* = Fill style:

Table 10: Fill styles

Name	No.	Information
`CrossHatchNoOut`	-2	Cross-hatched fill with no outline
`HatchedNoOut`	-1	Hatched fill with no outline
`Outline`	0	No fill, ouline only (default)
`Hatched`	1	Hatched fills
`CrossHatch`	2	Cross-hatched fills
`MixedColour`	3	Mixed-colors (stippled)
`Solid`	4	Solid fills

INTEGER, OPTIONAL *idense* = Hatched fill density:

Table 11: Hatched Fill Density

Name	No.	Information
`Sparse`	1	Sparse
`Medium`	2	Medium (default)
`Dense1`	3	Dense
`Dense2`	4	Very dense
`Dense3`	5	Very very dense

INTEGER, OPTIONAL *iangle* = Hatched line angle:

Table 12: Hatched Line Angle

Name	No.	Information
`FillHoriz`	3	Horizontal lines (default)
`FillVertic`	4	Vertical lines

`IGrFillPattern` defines the fill pattern (if any) to be used by `IGrCircle` and `IGrPolygonComplex`. The basic choice is between no fills and hatched, solid or mixed-color fills.

The default fill style is zero which gives outlines only. In this case, the density and angle parameters are ignored.

idense and *iangle* have the `OPTIONAL` attribute. They can be omitted when not required. If they are omitted when a value is expected, (e.g. for hatched fills) the indicated defaults are assumed.

Hatched fills draw lines at intervals across the area to be filled. If a hatched fill is selected, the density and angle parameters define the precise style of the fill. A dense fill uses roughly twice as many lines to fill the area as a sparse fill. The angle parameter controls the direction of the fill lines. Type `1` hatched fills draw lines in one direction only, according to the selected *iangle* value. Type `2` (cross-hatched) fills draw lines in both directions. Hatched fills are normally drawn with an outline. Specify a negative *istyle* value to suppress this outline.

Type `4` (solid) fills use a pure color, as most recently defined by a call to `IGrColourN`.

Type `3` (mixed) fills are similar to solid ones, except that the two colors as defined by the last two calls to `IGrColourN` are mixed. Hence if two successive calls to `IGrColourN` specify Yellow then Red, a type `3` fill will mix these colors. This will either use a stippled fill (where alternate pixels are plotted in each color) or a (r,g,b) value will be selected which is exactly half way between the two specified colors.This can give the appearance of many more shades than some devices actually support. Selecting the same color twice in succession gives a solid fill. On monochrome displays, the foreground/background colors are automatically mixed in stippled fills, regardless of the last two colors specified, unless those colors were identical in which case a solid fill is selected.

When solid/stippled fills are requested, angle and density are ignored.

If an invalid style, density or angle is specified, then the indicated defaults are used.

Example

```
CALL IGrColourN(48)              ! select first mixed-fill color
CALL IGrColourN(144)             ! select 2nd mixed-fill color
CALL IGrFillPattern(MixedColour) ! density and angle omitted
CALL IGrCircle(150.,500.,30.)
```

IGrLineType Subroutine

Description
Select line type (solid, dots, dashes or dot/dash).

Syntax
IGrLineType(*ltype*)

Arguments
INTEGER, OPTIONAL *ltype* = Line type:

Table 13: Line Types

Name	No.	Information
`SolidLine`	`0`	Solid (default)
`Dotted`	`1`	Dots
`Dashed`	`2`	Dashes
`DotDash`	`3`	Dot/dash
`DotDotDash`	`4`	Dot/dot/dash
`LongShort`	`5`	Long/short dashes *
`ShortDash`	`6`	Short dashes *

Effect
Selects the line type for subsequent drawing operations. The currently requested line type is available via `InfoGrScreen(36)`. If *ltype* is omitted, solid lines are selected.

Windows only supports 5 line styles. The line type which is supposed to be dotted is more like short dashes on most displays.

Portability notes
Windows: Only 5 line types are available. Line types 2 and 3 duplicate types 6 and 5 respectively.

Example

```
CALL IGrLineType(1)
!  draw a grid of dotted lines
DO I = 1, 9
  CALL IGrMoveTo(0.0, 0.1*REAL(I))
  CALL IGrLineToRel(1.0,0.0)
  CALL IGrMoveTo(0.1*REAL(I),0.0)
  CALL IGrLineToRel(0.0,1.0)
END DO
```

IGrPaletteInit Subroutine

Description

Reinitialize graphics color palette.

Syntax

IGrPaletteInit()

Effect

Reinitializes the W*interacter* graphics palette to the default settings. See `IGrColourN`.

IGrPalette Subroutine

Description

Redefine 8-bit color palette

Syntax

IGrPalette(*ncolor*,*rgb*,*ipost*)

Arguments

INTEGER *ncolor* = 8-bit color number (same numbering scheme as `IGrColourN`)

INTEGER *rgb* = 24-bit RGB color value

INTEGER, OPTIONAL *ipost* = Postpone palette realisation on 256 color screen
(0 or omitted=no 1=yes)

Effect

Controls the 8-bit graphics color palette, using the Red/Green/Blue (RGB) color scheme. An actual 24-bit color value is assigned to a specified 8-bit color number. Redefinition of the screen palette only affects subsequent plotting. The background color can be changed by calling `IGrPalette` with an *ncolor* value of 0.

ncolor specifies the color which would be selected by supplying the same value to `IGrColourN` using the 8-bit color model. Hence *ncolor* should lie in the range 0 to 255 and will be converted to an appropriate actual color number for the current screen mode, using the same rules as `IGrColourN`. When the 24-bit color model is selected, this routine is limited to setting the RGB value in the 8-bit palette which will still be used internally on color limited devices. See `IGrColourN` for a description of the colors available in the default palette. The current 8-bit palette values can be interrogated via `InfoGrPalette`.

rgb specifies the required physical color, in the usual 24-bit color range. It can be constructed using the `WRGB` function. Where an output device supports fewer colors, the nearest approximation to the requested color is selected.

When the optional *ipost* argument is specified as a non-zero value it causes 'realisation' of the screen palette to be postponed. When this argument is omitted or is set to zero, palette realisation is performed immediately. Enabling this option can have significant benefits on a 256 color display when setting multiple palette values. On a 256 color display, W*interacter* uses its own private palette for screen graphics colors. The size of this palette is determined by `WindowOpen`. It contains 16 entries by default, but can hold up to 128 different colors. Updating a single color in this palette can be a relatively "expensive" operation and can result in palette cycling effects if other applications or the desktop background also uses many colors. By setting *ipost*/=0, W*interacter*'s internal palette is updated but the expensive "realise" operation is not performed. When updating N palette values, a performance benefit can thus be obtained by setting *ipost* to be non-zero for the first N-1 calls, and zero (or omitted) for the final call. Hence only one realise-palette operation will be performed on a 256 color display instead of N such operations. This is both faster and neater. The `col256` demo illustrates this technique in the grey scale display option.

If *ipost*/=0 on a 256 color display, then the associated color number must not be used for drawing until a subsequent call specifies *ipost*=0 (or omits *ipost*) to force the palette to be realised. *ipost* has no effect on anything other than 256 color displays.

Color redefinition is normally only effective when the current Windows video driver or X server operates in a screen mode with 256 colors or more. Colors are not normally redefinable on a 16-color display, in which case the 'nearest' available color is used when plotting subsequently in color *ncolor*. On a display which provides more than 256 colors, use of the alternative 24-bit color model is recommended, since this allows RGB values to be specified directly to `IGrColourN`, rather than indirectly via `IGrPalette`.

Example

```
CALL IGrPalette(200,WRGB(255,200,200)) ! Pale pink
```

IGrPlotMode Subroutine

Description

Set the plotting mode

Syntax

IGrPlotMode(*mode*)

Arguments

CHARACTER(LEN=*), OPTIONAL *mode* = Plotting mode (N:normal overwrite, E:EOR (exclusive-or))

Effect

Selects the plotting mode for lines, points, software text and fills. In normal over-write mode, the line/point/text/fill simply replaces what was already in the drawable. In EOR mode, the color is exclusive or'ed with that already on the screen. The main use of this is that lines, text and fills can be drawn in EOR mode then erased again, still using EOR mode, without disturbing what was previously in the drawable.

Only the first character of the supplied argument is used to determine the required plot mode. If *mode* is blank or omitted, normal plotting is selected. The plot mode argument can be in upper or lower case.

The currently requested plot mode is available via `InfoGrScreen(37)`.

Portability notes

Windows: Due to a Windows GDI limitation, plot mode selection does not affect TrueType fonts, which are always drawn in normal (over-write) mode. Software text must be used if plot mode control is required.

Example

```
CALL IGrPlotMode('EOR')
CALL IGrPoint(X,Y)    ! now you see it
     :
CALL IGrPoint(X,Y)    ! now you don't
CALL IGrPlotMode(' ') ! ... and back to normal
```

Group GD: Graphics Drawing/Movement

The routines in this group provide the main W*interacter* graphics drawing primitives. An important concept here is the 'current plotting position'. This can be set explicitly using `IGrMoveTo`, but is automatically updated by other drawing and graphics text routines in the GD and GT groups.

`IGrCircle` and `IGrPolygonComplex` draw shapes in various styles which are determined by the `IGrFillPattern` routine in the GS group. By default they simply draw an outline of the appropriate shape, but they can also perform hatched, mixed-color or solid fills.

Simple straight line drawing can be performed using `IGrLineTo`. Single points can be plotted using `IGrPoint`.

IGrCircle Subroutine

Description

Draw/fill circle at an absolute position.

Syntax

IGrCircle(*xpos*,*ypos*,*radius*)

Arguments

REAL *xpos* = `X` co-ordinate of circle center

REAL *ypos* = `Y` co-ordinate of circle center

REAL *radius* = Radius of circle in current plotting units

Effect

Draws a circle of a given radius centered at the specified absolute plotting position, in the current graphics color and plotting mode as selected by `IGrColourN` and `IGrPlotMode`. The circle will be filled, if required, using the fill pattern selected by `IGrFillPattern`. The current plotting position becomes `(`*xpos*`,`*ypos*`)`. Aspect ratio is preserved regardless of window shape. The radius is expressed in terms of the X co-ordinate system.

Example

```
   CALL IGrUnits(50.,100.,500.,300.)
   CALL IGrFillPattern(2,2,3)
   CALL IGrCircle(100.,200.,20.)
```

Errors

`ErrBadRadius`	`(20)`	*radius* <= zero. Nothing will be drawn.

IGrLineTo Subroutine

Description

Draw line to a new absolute position.

Syntax

IGrLineTo(*xpos*,*ypos*)

Arguments

REAL *xpos* = `X` co-ordinate to draw to

REAL *ypos* = `Y` co-ordinate to draw to

Effect

Draws a line from the current plotting position (as set by a previous call to `IGrMoveTo` or `IGrLineTo` itself) to the new absolute plotting position specified by `(`*xpos*`,`*ypos*`)`. On exit, `(`*xpos*`,`*ypos*`)`becomes the current plotting position.

Example

```
CALL IGrUnits(0.0,0.0,1000.0,500.0)
CALL IGrMoveTo(200.0,100.0)
CALL IGrLineTo(800.0,100.0)
```

IGrMoveTo Subroutine

Description

Move current plotting position to a new absolute position.

Syntax

IGrMoveTo(*xpos*,*ypos*)

Arguments

REAL *xpos* = `X` co-ordinate

REAL *ypos* = `Y` co-ordinate

Effect

Moves the current plotting position to the absolute position `(`*xpos*`,`*ypos*`)` without any visible effect.

Example

```
CALL IGrUnits(100.,0.,300.,400.)
CALL IGrMoveTo(150.,200.)
CALL IGrLineTo(200.,300.)
```

IGrPoint Subroutine

Description

Draw a single point at new absolute position.

Syntax

IGrPoint(*xpos*,*ypos*)

Arguments

REAL *xpos* = `X` co-ordinate

REAL *ypos* = `Y` co-ordinate

Effect

Sets the current plotting position to the absolute position `(`*xpos*`,`*ypos*`)` and plots a point at that position.

Example

```
CALL IGrUnits(100.,0.,300.,400.)
CALL IGrPoint(150.,200.)
```

IGrPolygonComplex Subroutine

Description

Draw/fill a complex (possibly intersecting) polygon.

Syntax

IGrPolygonComplex(*x*,*y*,*nvert*)

Arguments

REAL *x*(:)	= Array of `X` co-ordinates
REAL *y*(:)	= Array of `Y` co-ordinates

INTEGER *nvert* = Number of vertices in supplied *x*/*y* arrays (<=5000)

Draws an irregular polygon defined by the specified absolute plotting positions, with possibly intersecting borders. The polygon is drawn in the current graphics color and plot mode as selected by `IGrColourN` and `IGrPlotMode`. The polygon will be filled, if required, using the pattern set by `IGrFillPattern`. The polygon will be closed, i.e., the last point will be joined to the first.

IGrPolygonComplex uses an API primitive for solid and mixed-color fills. A generic scan-line search method is used for hatch fills. In very rare situations, a polygon may be too complex for IGrPolygonComplex's generic algorithm, in which case the routine will exit with error code 49. This is only likely to occur in very extreme cases.

If no fill is specified, IGrPolygonComplex simply draws a poly-line, i.e. it joins the points specified in *x* and *y* regardless of whether the borders cross. Whether filled or not, the current plotting position becomes (*x*(1),*y*(1)).

Example

```
REAL, DIMENSION (4095) :: X, Y
READ(20,*) N
N = MIN(N,4095)
DO I = 1,N
  READ(20,*) X(I),Y(I)
END DO
CALL IGrPolygonComplex(X,Y,N)
```

Errors

ErrFillComplex	(49)	Fill too complex. Unable to fill polygon.

Group GT: Graphics Text

Graphics text strings are written using WGrTextString. Font family, style, size and spacing are all selectable via WGrTextFont. This provides access to both driver-specific fonts and software based vector/outline fonts. Data for the latter are loaded via the various WGrVFont* and WGrOFont* routines. In general, we recommend use of driver-specific fonts. Software fonts are provided for portability reasons and are sometimes used internally to substitute for unavailable fonts in non-Windows GDI output.

The alignment, rotation angle and direction of graphics text is controlled by WGrTextOrientation. The relative length of a graphics text string can be measured via the WGrTextLength function.

Backward Compatibility Note : the routines in the GT group replace the earlier IGrChar* routines of the GC group. In particular, the use of external software 'character set' files became obsolete in the new calling interface, though the previous font selection mechanisms are still supported for backwards compatibility.

WGrOFont* Subroutines

Description

Load outline software font data

Syntax

WGrOFontFixed()
WGrOFontSwiss()

Effect

Each of these routines contains all of the font shape data for the corresponding outline software font. Calling one of these routines loads the data for that font into W*interacter*'s software font data area. This font can then be selected by calling `WGrTextFont` with a font family type of `FFSoftware`. The current software font can be changed at any time without needing to call `WGrTextFont` again, unless style, size or spacing are to be changed (in which case `WGrTextFont` should be called with the appropriate arguments assigned).
Note: Unlike the earlier `IGrCharSet` routine which they replace, they do not actually select the named font for text output. Rather they load the specified font, ready to be selected by `WGrTextFont`.

Example

```
   CALL WGrOFontSwiss()
   CALL WGrTextFont(FFSoftware,FSBold)
   CALL WGrTextString(0.5,0.5,'Bold Swiss outline font')
```

WGrTextFont Subroutine

Description

Set graphics text alignment, rotation and direction

Syntax

WGrTextFont(*ifamily,istyle,width,height,name,ispace)*

Arguments

INTEGER, OPTIONAL *ifamily* = Font family

FFUser	(100)	: User defined GDI font
FFCourier	(101)	: Courier
FFHelvetica	(102)	: Helvetica/Arial
FFSoftware	(1)	: Current software font
FFDriver	(2)	: Current driver-specific font
0 or omitted		: Leave current selection unchanged

INTEGER, OPTIONAL *istyle* = Font style. Sum of:

FSBold	(1) : Bold
FSItalic	(2) : Italic
FSUnderline	(4) : Underline
FSOpaque	(8) : Opaque background

REAL, OPTIONAL *width* = Average character width as a proportion of the graphics area
(default = 0.01333333)

REAL, OPTIONAL *height* = Character cell height as a proportion of the graphics area
(default = 0.04)

CHARACTER, OPTIONAL *name* = User defined GDI font name, if *ifamily*=FFUser

INTEGER, OPTIONAL *ispace* = Spacing
(0=font-specific, 1=fixed, 2=software-proportional)

Effect

Selects the font to be used in graphics text output.

ifamily=101-102 selects a driver-specific font of the specified family. As their name suggests, driver-specific fonts are those which are available for use by a particular W*interacter* graphics output driver. The most common examples are TrueType (Windows GDI) or Adobe (PostScript) fonts. If a particular output driver does not support such a font, an equivalent software outline font will be substituted automatically. These software outline fonts are built into the library as standard.

ifamily=100 selects a user-specified GDI font, as named in the *name* argument. This can be any Windows font name and will be used in all subsequent Windows GDI output to screen, memory bitmap, metafile or printer. If this option is specified when generating non-GDI output, a Courier or Helvetica style font will be selected, depending on the currently selected spacing. If *ifamily* specifies a user defined font, but no name is supplied an error code is set and a Courier or Helvetica font will be selected, as for non-GDI output.

ifamily=1/2 allows switching between the current software and device-specific fonts:

- `FFSoftware` : Selects the current software font as determined by the most recent call to one of the `WGrOFont*` or `WGrVFont*` routines. The 'Standard' vector font is loaded by default.
- `FFDriver` : Selects the current driver-specific font as most recently selected by *ifamily*=100-104. This allows such a font to be reselected after using a software font, without the overhead of recreating the driver-specific font

If *ifamily* is omitted or zero, the current font family selection will remain unchanged (useful when only the font size needs to be changed). The initial default font family is driver-specific Courier (101).

istyle selects the required font style. This can be changed without specifying any other arguments, if required. Specifying zero will disable all. Omitting this argument leaves the current style selection unchanged. When opaque text is enabled, the background of any graphics text is filled using the last but one graphics color selected by `IGrColourN`.

Some minor qualifications apply to style selection:

- The character generator used to draw software vector fonts uses double-width lines when bold is enabled.
- Underlining and opaque backgrounds work well with all GDI fonts, all software fonts and non-GDI Courier style fonts. However, the underline and opaque background extent is only guaranteed to be correct for non-GDI driver-specific proportional fonts when software-based proportional spacing is selected.

width and *height* specify the required character cell size, expressed as a proportion of the graphics area size. Hence, the physical font size will change if the graphics area size changes. The initial default width and height are 0.013333 and 0.04 (chosen for backwards compatibility reasons). If *width* or *height* are omitted, the most recently specified size is used.

name allows any available Windows font to be specified by name, for use in GDI output, provided *ifamily*=100. It will be ignored, if present and *ifamily*/=100.

ispace allows specific font spacing to be enforced, dependent on which font family was requested:

- *ispace*=0 : Selects the 'native' spacing associated with a given font. This is the default if omitted and *ifamily* is present. This means monospaced text for the Courier font or the Standard software vector font. Driver-specific proportional spacing is used when *ifamily*=100 or 102. Software-based proportional spacing is selected when any software font other than Standard is loaded.
- *ispace*=1 : Selects fixed spacing if *ifamily*=1, 100 or 101. Native spacing is used otherwise.
- *ispace*=2 : Selects software based proportional spacing, if *ifamily*=2, 100 or 102. Native spacing is used otherwise. 'Software' spacing uses a built-in font-independent character width table which can be updated by `WGrTextWidth`. Due to the generic nature of this width table, the quality of spacing may suffer, but this option has certain specific benefits:
  - `WGrTextLength` is guaranteed to give accurate results.
  - The underlining extent of non-GDI proportional device-specific fonts is correct.
  - The current graphics position is updated more accurately, giving better results if the (*xpos*,*ypos*) arguments are omitted when writing strings.
- Spacing remains unchanged if *ifamily* and *ispace* are both omitted (e.g. if font size only is specified).

Most fonts provide 8-bit ISO Latin-1 character sets (i.e. character codes 32-126 and 161-255).

Portability notes

Windows: The Courier New and Arial TrueType fonts are used for families 101/102. Both provide 8-bit Latin-1 character sets (except Symbol) and are fully rescalable.

X Windows: The Courier and Helvetica families are both supported under X Windows. While rescaleable X fonts are used, these do not guarantee to give exactly the font size requested. When a font of the required size (or a near equivalent) is not available, an outline software font (and hence software spacing) is substituted. This also occurs under all conditions for rotated (i.e. non-horizontal) text output. The supported X fonts are 8-bit ISO Latin-1 fonts. Changing character size under X Windows, may require a new font file to be loaded. Frequent character size changes should therefore be avoided where possible.

Example

```
CALL WGrTextFont(FFHelvetica)
CALL WGrTextString(0.5,0.8,'Helvetica/Arial')
CALL WGrTextFont(FFCourier,FSBold+FSItalic)
CALL WGrTextString(0.5,0.7,'Courier Bold/Italic')
CALL WGrTextFont(FFUser,NAME='Comic Sans MS')
CALL WGrTextString(0.5,0.6,'User Defined')
CALL WGrTextFont(FFHelvetica,WIDTH=0.025,HEIGHT=0.08)
CALL WGrTextString(0.5,0.5,'Enlarged Arial')
CALL WGrVFontTriplexRoman()
CALL WGrTextFont(FFSoftware,WIDTH=0.013,HEIGHT=0.04)
CALL WGrTextString(0.5,0.4,'Triplex Roman (vector)')
CALL WGrTextFont(FFDriver)
CALL WGrTextString(0.5,0.3,'Back to Arial')
```

WGrTextLength Function

Description

Measure the length of a graphics text string.

Syntax

REAL WGrTextLength(*string*,*method*)

Arguments

CHARACTER *string* = String or character to measure

INTEGER, OPTIONAL *method* = How to measure the string:

0 or omitted : Use ’best’ available method

1 : Force use of software character widths table

Effect

When proportional spacing is enabled, this function returns the relative length of the specified string, assuming the average character width is 1.0. When fixed spacing is enabled, `WGrTextLength(STRING)` always returns `REAL(LEN(STRING))`. Hence the result of `WGrTextLength` multiplied by `InfoGraphics(3)` always returns the width of the string in user-units, regardless of which type of spacing is enabled.

Internally, this function will use one of two methods to measure a string when a proportionally spaced font is currently selected. When a driver-specific font and 'native' spacing is selected, `WGrTextLength` will attempt to use the underlying API to measure the string. When a software font or software spacing is in use, a generic internal widths table is used. Which is most appropriate of these two methods varies depending on several factors, so `WGrTextLength` will choose the 'best' when *method*=0 or is omitted. Specify *method*=1 to force the software widths table to be used.

Example

```
CALL WGrVFontSwiss()
CALL WGrTextFont(FFSoftware)
! draw a box around a string
WIDTH  = WGrTextLength(STRING)*InfoGraphics(3)
HEIGHT = InfoGraphics(4)
CALL WGrTextOrientation(AlignLeft)
CALL WGrTextString(X,Y+HEIGHT/2.0,STRING)
CALL IGrFillPattern(Outline)
CALL IGrMoveTo(X,Y)
CALL IGrLineTo(X+WIDTH,Y)
CALL IGrLineTo(X+WIDTH,Y+HEIGHT)
CALL IGrLineTo(X,Y+HEIGHT)
CALL IGrLineTo(X,Y)
```

WGrTextOrientation Subroutine

Description

Set graphics text alignment, rotation and direction

Syntax

WGrTextOrientation(*ialign*,*angle*,*idir*,*nangle*)

Arguments

INTEGER, OPTIONAL *ialign* = Alignment of graphics text strings

AlignLeft	(0) : Left
AlignCentre	(1) : Centre
AlignRight	(2) : Right

REAL, OPTIONAL *angle* = Graphics text rotation angle
(degrees counter clockwise from horizontal)

INTEGER, OPTIONAL *idir* = Graphic text direction

DirHoriz	(0) : Horizontal
DirVertic	(1) : Vertical

INTEGER, OPTIONAL *nalign* = Reserved. Not used in WiSK.

Effect

Sets the alignment, angle of rotation and direction of graphics text. If any argument is omitted, that setting remains unchanged. By default text is centred, unrotated and horizontal.

ialign determines how graphics text strings/numbers are aligned relative to a plotting position Left aligned text is printed starting from a given position, right aligned text finishes at that position and centred text appears either side of it. This is true of both horizontal and vertical text and is independent of the current angle of rotation. In all cases 'left' and 'right' refer to which end of the string is actually at the specified plotting position.

angle defines the angle at which graphics text strings/numbers are to be written. The angle is measured in degrees counter clockwise from the horizontal, which is treated as zero. Hence, an angle of 90 degrees would give sideways text which runs vertically from the bottom of the graphics area toward the top.

idir specifies graphics text direction relative to the angle of rotation specified by *angle*. i.e. Vertical text is printed one character above the next, perpendicular to the base line defined by the current rotation angle.

Example

```
CALL WGrTextOrientation(AlignLeft)
CALL WGrTextString(100.,100.'This starts at (100,100)')
CALL WGrTextOrientation(AlignRight,0.0,DirVertic)
CALL WGrTextString(100.,150.,'This finishes at (100,150)')
CALL WGrTextOrientation(AlignCentre,180.0,DirHoriz)
CALL WGrTextString(200.0,200.0,'Upside down text !')
CALL WGrTextOrientation(ANGLE=90.0)
CALL WGrTextString(200.0,250.0,'Bottom to top')
```

WGrTextString Subroutine

Description

Output character string at an absolute `(x,y)` position.

Syntax

WGrTextString(*xpos*,*ypos*,*string*)

Arguments

REAL *xpos* = `X` co-ordinate

REAL *ypos* = `Y` co-ordinate

CHARACTER *string* = String to write

Effect

Outputs *string* at (*xpos*,*ypos*). The plotting mode and color are as previously defined by `IGrPlotMode` and `IGrColourN`. Font style, size and spacing are determined by `WGrTextFont`. A monospaced Courier style font is used by default.

The position of the text relative to (*xpos*,*ypos*) is determined by `WGrTextOrientation`, as are the direction and angle of rotation of the string. By default, text is centre aligned, unrotated and horizontal.

When text is output horizontally, *ypos* specifies a position half-way up a character. When vertical output has been selected, *xpos* specifies a position halfway across a character. In left/right justification mode the other co-ordinate specifies an extreme end of the string. These rules apply regardless of the angle of rotation.

On exit the current plotting position is updated to a point within the next character cell after the string which has been written. The exact position within the cell will depend on the orientation selected by `WGrTextOrientation`. (For the purposes of calculating this plotting position, the final character cell after the output string is assumed to be a fixed width cell, regardless of whether fixed or proportional spacing is currently selected.)

If either of *xpos* or *ypos* are omitted, the corresponding current plotting position is used. This allows text to be placed immediately after a previous string when using left alignment.

Text which would extend beyond the limits of the graphics area (as defined by `IGrArea`) is clipped at the edge of that area. Text which would be completely outside the graphics area is not printed.

Graphics text is normally transparent, i.e. it does not obliterate any underlying graphics. Specifying `FSOpaque` in the font style to `WGrTextFont` will cause graphics text to be written with an opaque background in the color specified by the last but one call to `IGrColourN`.

Normal 'over-write' plot mode is recommended for vector-based software fonts. If text needs to be plotted in exclusive-or mode (to allow it to be 'unplotted' later), use an outline software font.

Text written by this routine can contain both 7-bit and 8-bit characters, as defined in the ISO Latin-1 standard (i.e. character codes in the range 32-126 and 161-255). However, while very widely supported, Latin-1 8-bit characters are not universally available on all devices or in all fonts. See `WGrTextFont`.

Example

```
CALL WGrTextString(100.0,200.0,'This is centred at (100,200)')
CALL WGrTextOrientation(AlignLeft)
CALL WGrTextString(300.0,200.0,'This starts at (300,200)')
```

WGrVFont* Subroutines

Description

Load vector software font data

Syntax

WGrVFontDuplexRoman(*igl*)
WGrVFontStandard(*igl*)
WGrVFontTriplexRoman(*igl*)

Arguments

INTEGER, OPTIONAL *igl* = Reserved. Omit in W*i*SK programs.

Effect

Each of these routines contains all of the font shape data for the corresponding vector software font. Logically, these are exactly equivalent to the various `WGrOFont*` routines except that the associated fonts are line based rather than filled outlines. They should be called in combination with `WGrTextFont`.

The *igl* argument is reserved for use with the full version of W*interacter* and can be omitted.

Example

```
CALL WGrVFontStandard()
CALL WGrTextFont(FFSoftware,FSBold)
CALL WGrTextString(0.5,0.5,'Bold Standard vector font')
```

13 General Functions

Group IF: Information

To help you find out exactly what facilities are available on the current system or to simply interrogate the state of W*interacter* variables, a number of functions and subroutines are provided to return information to the calling program. In all cases one call to an IF group routine returns one item of information.

The routines in this group fall into two categories:

- *INTERACTER* compatible Info functions
- W*interacter* specific WInfo routines

Logically there is no particular difference between these two sets of routines. Their names simply differ to reflect the above separation.

In all cases, specifying an invalid information item number to an IF group routine returns an undefined result.

InfoError Function

Description

Return error information.

Syntax

INTEGER InfoError(*item*)

Arguments

INTEGER *item* = Number of information item required:

Table 14: Error Information *item*s

Name	No.	Information
`LastError`	1	Last error set by W*interacter* (`0` if no errors since last call to `InfoError(1)`)
`IOErrorCode`	2	I/O code for last type `1` or `2` error (file open or read/write error), otherwise undefined
`OsErrorCode`	3	Operating system error code

Effect

Returns information about the last error to be detected. If no errors have occurred since start-up or since the last call to `InfoError`, the last error is returned as zero. If several errors have occurred since the last call to `InfoError`, only the most recent error is returned.

A call to `InfoError` also resets the corresponding error flag (depending on the value of *item*) to zero. This feature can be used to clear the error flags, when your program is uncertain of what errors may already have occurred.

If a type `1` or type `2` error occurs (error on file/device open, read or write), a call with an *item* value of `2` returns the associated Fortran OPEN/READ/WRITE statement `IOSTAT` value or system I/O routine error code. This `IOSTAT` value will still be available if further I/O is performed or other non I/O errors occur. i.e., The error code for an I/O error remains available until it is cleared by interrogating `InfoError(2)` or until another I/O error occurs.

item 3 returns the error code set by internal operating system interface routines. Typically this value will only be set when *item* 1 returns a value of 13 (Operating system command error). This will be an internal operating system error code.As for the `IOSTAT` value described above, this return code remains available until it is cleared by interrogating `InfoError(3)` or until another o.s. error occurs.

Example

```
CHARACTER(LEN=6) :: STR
IERROR = InfoError(1) ! clear error flags first
ISTAT  = InfoError(2)
CALL IGrCharSet(’badname.chr’) ! obsolete, used for illustration
IERROR = InfoError(1)
IF (IERROR==1.OR.IERROR==2) THEN
    CALL WMessageBox(OKOnly,StopIcon,1,'Error on load', &
                     'File Error')
    ISTAT = InfoError(2)
END IF
```

InfoGraphics Function

Description

Return real graphics mode information.

Syntax

REAL InfoGraphics(*item*)

Arguments

INTEGER *item* = Number of information item required:

Table 15: Graphics Mode Information *item*s

Name	No.	Information
`GraphicsX`	1	Current `X` plotting position (in user units as set in `IGrUnits`)
`GraphicsY`	2	Current `Y` plotting position (in user units as set in `IGrUnits`)
`GraphicsChWidth`	3	Current character width (in user units as set in `IGrUnits`)
`GraphicsChHeight`	4	Current character height (in user units as set in `IGrUnits`)
`GraphicsAreaMinX`	7	Left limit of main graphics area
`GraphicsAreaMinY`	8	Lower limit of main graphics area
`GraphicsAreaMaxX`	9	Right limit of main graphics area
`GraphicsAreaMaxY`	10	Upper limit of main graphics area
`GraphicsUnitMinX`	11	Lower `X` co-ordinate limit
`GraphicsUnitMinY`	12	Lower `Y` co-ordinate limit
`GraphicsUnitMaxX`	13	Upper `X` co-ordinate limit
`GraphicsUnitMaxY`	14	Upper `Y` co-ordinate limit

Effect

Returns certain REAL graphics mode parameters. These are generally dynamic values which change depending on calls to other graphics routines. See also `InfoGrScreen` which returns INTEGER data, mainly describing the capabilities of the current display.

The current plotting position, as set by `IGrMoveTo` and other routines, is accessible using *item* values `1` and `2`.

The current graphics text character size (*item*'s `3` and `4`) is derived from the size set using `WGrTextFont` and converted to user plotting units. This can be useful in calculating the extent of a graphics string to be output by `WGrTextString`.

item's `7` to `10` return the graphics area dimensions as most recently specified to `IGrArea`. Similarly, *item*'s `11` to `14` return the user definable graphics area co-ordinates as most recently specified to `IGrUnits`.

Example

```
! write a blue string on a white background
CALL WGrTextFont(FFCourier)
WIDTH  = FLOAT(LEN(STRING))*InfoGraphics(3)
HEIGHT = InfoGraphics(4)
CALL IGrMoveTo(X,Y)
CALL IGrColourN(223)
CALL IGrFillPattern(Solid)
CALL RECTANGLE(WIDTH,HEIGHT)
CALL WGrTextOrientation(AlignLeft)
CALL IGrColourN(159)
CALL WGrTextSTring(X,Y+HEIGHT/2.0,STRING)
```

InfoGrPalette Function

Description

Return 8-bit color palette information.

Syntax

INTEGER InfoGrPalette(*item*)

Arguments

INTEGER *item* = 8-bit color number

Effect

Returns the RGB value associated with color number *item* in W*interacter*'s device independent 8-bit graphics color palette, as used by `IGrColourN` when the 8-bit color model is selected. These are the same color values which can be updated by `IGrPalette`. This information is available regardless of which color model (8-bit or 24-bit) is currently selected.

The returned RGB value is encoded in the usual 24-bit format, which can be separated into individual color components using `WRGBsplit`.

InfoGrScreen Function

Description

Return graphics facilities information (screen).

Syntax

INTEGER InfoGrScreen(*item*)

Arguments

INTEGER *item* = Number of information item required:

Table 16: Graphics Screen Information *item*s

Name	No.	Information
`ColNumAvailable`	30	Number of colors availablee
`AspectRatio`	32	Aspect ratio as a percentage
`PrevColReq`	34	Last but one requested graphics color
`ColorReq`	35	Most recently requested color
`LineTypeReq`	36	Most recently requested line type
`PlotModeReq`	37	Most recently requested plot mode PlotNormal (0) = Normal/overwrite PlotEor (3) = Exclusive-or
`Col24Bits`	42	24-bit color specification supported (0=no 1=yes)

Effect

Returns information about the graphics facilities available on the current display in the current mode. The value returned is an INTEGER.

item number 30 return the number of selectable colors (see `IGrColourN` for details of how this is determined).

item 32 returns the aspect ratio of the current drawable as a percentage. For example, in a window with an aspect ratio of 1.4, a value of 140 would be returned. This aspect ratio is used internally by W*interacter* when drawing circles, but can also be useful in applications which require a shape to be rotated without distortion.

item's 34 and 35 return the last two color numbers which have been requested. The color returned by *item* 34 is only used in mixed-color fills.

item's 36 and 37 returns the last line type and plot mode requested via `IGrLineType` and `IGrPlotMode`.

item 42 reports support for 24-bit color specification. This will report 1 on a display with more than 256 colors, or 0 otherwise. In the former case, use of the 24-bit color model is available and recommended (see `IGrColourModel`). If zero is returned, the 24-bit color model can still be used, but the 8-bit color palette will be used internally.

WInfoDialog Function

Description

Return dialog information.

Syntax

INTEGER WInfoDialog(*item*)

Arguments

INTEGER *item* = Number of information item required.

Table 17: Dialog Information *items*

Name	No.	Information
ExitButton	1	Identifier of button used to terminate program-supplied modal dialog
ExitField	2	Current field when modal dialog terminated
CurrentDialog	3	Identifier of current dialog, 0 = none
ExitButtonCommon	4	Button used to terminate common dialog (0-2)
DialogXPos	6	Current dialog X position
DialogYPos	7	Current dialog Y position
DialogWidth	8	Current dialog width
DialogHeight	9	Current dialog height
DialogType	10	Current dialog type DialogPopup (1) : Popup dialog DialogChild (2) : Child dialog DialogCombined (4) : Combined with a window

Effect

Returns dialog information. *item*'s 1-4 are available after a modal or common dialog function call has terminated. *item*'s 6-9 are available for currently visible modeless or semi-modeless dialogs.

item 1 returns the identifier of the push button used to exit a program supplied modal dialog. If an error occurred in the dialog, -1 is returned. Commonly used identifiers (e.g. `IDOK` and `IDCANCEL` which are typically attached to the OK and Cancel buttons) are defined in the `WINTERACTER` module. See the `PushButton` message under `WMessage` for a list. The actual button codes will depend on the definition of the particular dialog.

item 2 returns the identifier of the last active field before a dialog terminated. This can be useful if a Help button is pressed, to provide context sensitive help. This return value is not available for common dialogs.

item 3 returns the current dialog identifier, as set by `WDialogLoad` or `WDialogSelect`. Note, this is not necessarily the same as the dialog that has the current input focus.

item 4 returns a code which indicates which button was used to terminate a common dialog in the CD group:

Table 18: Common Dialog Termination Codes

Name	No.	Button
CommonCancel	0	Cancel
CommonIgnore	0	Ignore
CommonOK	1	OK
CommonOpen	1	Open
CommonYes	1	Yes
CommonRetry	1	Retry
CommonAbort	2	Abort
CommonNo	2	No

item's 6 and 7 return the position of the current dialog as selected by `WDialogLoad` or `WDialogSelect`. The returned values are in pixels relative to the top left corner of the screen for a popup dialog. For a child dialog, the returned values are in W*interacter* window units, relative to the top left corner of the root window.

item's 8 and 9 return the width and height of the current dialog, in the same units as *item*'s 6 and 7.

item 10 determines if the current dialog is a popup or child dialog or is combined with a window.

Example

```
CALL WDialogLoad(ID_DIALOG1)
CALL WDialogShow(ITYPE=Modal)
IF (WinfoDialog(ExitButton)==IDOK) THEN
    CALL WDialogGetString (IDC_STRING1,CVALUE)
END IF
CALL WDialogUnload()
```

WInfoDrawable Function

Description

Return drawable information.

Syntax

INTEGER WInfoDrawable(*item*)

Arguments

INTEGER *item* = Number of information item required

Table 19: Drawable Information *items*

Name	No.	Information
DrawableType	1	Type (1=window 3=dialog field)
DrawableID	2	Handle/identifier
DrawableWidth	3	Width in pixels
DrawableHeight	4	Height in pixels

Effect

Returns information about the current target graphics drawable, as selected by `IGrSelect`.

item's 1 and 2 return the drawable’s type and handle/identifier as set by `IGrSelect`.

item's 3 and 4 return the drawable’s pixel dimensions.

WInfoScreen Function

Description

Return screen information.

Syntax

INTEGER WInfoScreen(*item*)

Arguments

INTEGER *item* = Number of information item required

Table 20: Screen Information *items*

Name	No.	Information
ScreenWidth	1	Screen Width
ScreenHeight	2	Screen Height
ScreenColours	3	Number of screen colors

Effect

Returns information about the current screen. This information is available immediately after `WInitialise` has been called, allowing the results of this function to be used to determine the required root window size.

item's 1 and 2 return the screen resolution, in pixels, for the video mode which was selected when `WInitialise` was called. If a dynamic video mode changer (e.g. QuickRes) is used subsequently, the new screen dimensions will not be updated.

item 3 returns the total number of screen colors available in the current screen mode when `WInitialise` was called. Note this is different to the number of colors used by W*interacter* graphics. See `IGrColourN`.

Example

```
CALL WInitialise()
ISCRWID = WInfoScreen(1) ! Get screen width
ISCRHGT = WInfoScreen(2) ! Get screen height
CALL WindowOpen(WIDTH=ISCRWID=2,HEIGHT=ISCRHGT/3) ! Open window
```

WInfoWindow Function

Description

Return window information.

Syntax

INTEGER WInfoWindow(*item*)

Arguments

INTEGER *item* = Number of information item required.

Table 21: Window Information *items*

Name	No.	Information
WindowWidth	1	Current window width
WindowHeight	2	Current window height
OpenFlags	3	Window style flags
WindowHandle	4	Current window handle
WindowXPos	5	Current window X position
WindowYPos	6	Current window Y position
ClientXPos	7	Current window client-area X position
ClientYPos	8	Current window client-area y position
WindowState	9	Current window state : WinMinimised (0) : minimised WinNormal (1) : normal size WinMaximised (2) : maximised
WindowType	10	Window type: WinStandard (0) : standard window WinDialog (3) : combined window/dialog

Effect

Returns window related information.

item's 1 and 2 return the dimensions of the current root or child window as selected by `WindowSelect`, `WindowOpen` or `WindowOpenChild`. The returned values are in pixels.

item 3 returns the *flags* value specified in the original `WindowOpen` or `WindowOpenChild` call.

item 4 returns the W*interacter* handle of the current window. This will be zero for the root window or 1-20 for a child window.

item's 5 and 6 return the position of the current root or child window as selected by `WindowSelect`, `WindowOpen` or `WindowOpenChild`. The returned values are in pixels relative to the top left corner of the screen for a root or floating child window. For an 'inside parent' child window, the returned values are in W*interacter* window units, relative to the top left corner of the parent window.

item's 7 and 8 return the position of the top left corner of the 'client' area of the current window (i.e. the drawable area within the frame). This is always expressed in pixels relative to the corner of the screen.

item 9 returns the minimised/normal/maximised state of the currently selected window.

item 10 identifies the type of the current window. A basic window which has not been combined with a dialog is reported as a 'standard' window (0). Such a window will be selectable as a target graphics drawable.

Portability notes

X Windows: Item 9 is not implemented, since this information is not available.

Example

```
IWINWID    = WInfoWindow(1) ! Get parent window width
IWINHGT    = WInfoWindow(2) ! Get parent window height
! Set child to half width/height  of parent
CALL WindowOpenChild(IHANDLE,WIDTH=IWINWID/2,HEIGHT=IWINHGT/2)
```

Group OS: Operating System Interface

The routines in this group provide access to environment variables (`IOsVariable`) and allow controlled program termination with a message (`IOsExitProgram`).

IOsExitProgram Subroutine

Description

Abort program with an error message and error code.

Syntax

IOsExitProgram(*errmes*,*iexcod*)

Arguments

CHARACTER *errmes* = Error message to display to the user.

(if blank, error message display is suppressed)

INTEGER *iexcod* = Exit code to return to operating system

Effect

Aborts the current program, with the message string 'Abnormal exit (code nn)' followed by the supplied error message. This routine is designed to be used when an unexpected fatal error is encountered. If *errmes* is blank, the program terminates without a message.

In general it is recommended that exit codes greater than 20 are used. Codes from 1 to 20 are reserved for use by W*interacter*.

If you wish to leave a program immediately without issuing either an error message or a non-zero exit code, simply supply a blank error message and an *iexcod* value of zero.

Portability notes

Windows: The 'Abnormal exit' message is displayed in a standard message box. The supplied exit code is then returned to Windows via the API ExitProcess function.

Linux: The 'Abnormal exit' message is written to standard output. The supplied exit code is returned to the shell via the C library exit function. The special shell variables $? (Bourne shell) or $status (C shell) will contain the program exit code.

Example

```
LOGICAL :: EXISTS
INQUIRE(FILE=’mydata.dat’,EXIST=EXISTS)
IF (.NOT.EXISTS) CALL IOsExitProgram('Data file not found’,21)
```

IOsVariable Subroutine

Description

Return the value of an operating system environment variable.

Syntax

IOsVariable(*vname*,*value*)

Arguments

CHARACTER *vname* = Name of variable to interrogate

CHARACTER *value* = Returned value (blank if not found)

Effect

Returns the value of the specified environment variable. If the specified variable name has not been initialized or has no value, *value* will be returned blank.

Portability notes

Windows: `IOsVariable` returns environment variables as assigned using the SET command (or Control Panel, under Windows NT). The operating system converts all variable names to upper case, so the supplied variable name *vname* must also be in upper case. When defining environment variables using SET, avoid trailing spaces between the variable name and the '=' since these will be treated as part of the variable name. `IOsVariable` strips trailing blanks from the supplied variable name.

An error code will be set if the return buffer *value* is too small. *value* is returned blank in this case.

Linux: If the C shell (csh) is in use it is important to distinguish between environment variables and operating system variables. `IOsVariable` returns environment variables, which are assigned in the C shell using the `setenv` command. If the ’bash’ shell is used, environment variables can be assigned using commands of the form :

```
export VARNAME=string
```

Errors

`ErrBufferSize`	`(1023)`*value* is too small

Example

```
CHARACTER (LEN=80) :: FILNAM
CALL IOsVariable('DEFDATA',FILNAM)
IF (IActualLength(FILNAM)>0) THEN
  OPEN(20,FILE=FILNAM,STATUS='OLD')
ELSE
  OPEN(20,FILE='default.dat',STATUS='OLD')
END IF
```

Group MI: Miscellaneous

This group is for routines which have no obvious home elsewhere.

The most important routine in this group is `WInitialise`. It must be called in every W*interacter* program before opening a window.

OpenGL graphics can be enabled via `WglSelect`. The associated `WglSwapBuffers` routine exchanges the front/back buffers when double buffered OpenGL output is enabled.

`WCursorShape` allows the mouse cursor to be selected from various pre-defined shapes or a user defined cursor can be specified.

Color values can be converted to/from 24-bit RGB integer color values using `WRGB` and `WRGBsplit`.

`WindowBell` is provided to ring the bell and to control whether other routines ring the bell.

WCursorShape Subroutine

Description

Select shape of mouse cursor

Syntax

WCursorShape(*ishape*)

Arguments

INTEGER *ishape* = Mouse cursor shape

Table 22:

Name	no.	Cursor Type
CurArrow	0	Standard arrow
CurHourGlass	1	Hourglass
CurSmallHour	2	Standard arrow/small hourglass
CurCrossHair	3	Crosshair
CurIBeam	4	Text I-beam
CurCircle	5	Slashed circle
CurFourPoint	6	Four pointed arrow (N/S/E/W)
CurDoubleNS	7	Double pointer (N to S)
CurDoubleEW	8	Double pointer (E to W)
CurDoubleNESW	9	Double pointer (NE to SW)
CurDoubleNWSE	10	Double pointer (NW to SE)
CurVertical	11	Vertical arrow
	101+	User defined cursor

Effect

Selects the mouse cursor shape. The mouse cursor is updated immediately provided it is within one of the application’s windows or in a user-drawn picture/frame dialog field (i.e. a field which has been selected for graphics output via `IGrSelect`).

ishape can either specify a pre-defined cursor or the identifier of a user defined cursor. The latter should be defined in the program's resource script. User-defined cursors can be created using the resource editor.

Identifiers of user defined cursors must be greater than 11. We recommend values greater than 100, to allow for the possible introduction of additional pre-defined cursor types in future.

If a specified user defined cursor does not exist, the standard arrow cursor

Portability notes

X Windows: User defined cursor (101+) are not currently supported under X.

Example

```
CALL WCursorShape(CurHourGlass)
! . . . Do some data processing
CALL WCursorShape(CurArrow)   ! Restore cursor shape
```

WFlushBuffer Subroutine

Description

Flush X Windows screen output buffer.

Syntax

WFlushBuffer()

Effect

Flushes the screen output buffer to synchronise the application and the display under X Windows. It has no effect under Windows. It will rarely be necessary to call this routine, but it may be required before performing a time consuming operation.

Example

```
CALL WGrTextString(.5,.5,'Hang on while I crunch some numbers!')
CALL WFlushBuffer()
CALL CRUNCH()
```

WglSelect Subroutine

Description

Enable/disable OpenGL graphics

Syntax

WglSelect(*itarget*,*ident*,*iflags*)

Arguments

INTEGER *itarget* = Target drawing surface for OpenGL graphics

Disabled	(0) : None
DrawWin	(1) : Window
DrawField	(3) : Dialog field

INTEGER, OPTIONAL *ident* = Window handle or field identifier

INTEGER, OPTIONAL *iflags* = Flags controlling type of OpenGL support. Sum of:

wglColourIndex	(1) : Colour index model (default=RGBA)
wglDoubleBuffer	(2) : Double buffering (default=single buffering)

Effect

Selects the target drawing surface for OpenGL graphics. Specifying a zero target drawable disables OpenGL graphics. Only one target drawable may be selected for OpenGL graphics output at any one time. If OpenGL graphics are already enabled on another drawable, OpenGL output to that drawable is automatically disabled.

`WglSelect` is logically similar to `IGrSelect`. It allows OpenGL output to be routed to the following types of target drawable:

itarget = 1 : Enables OpenGL graphics output to the specified window. *ident* must specify a valid window handle, as returned by `WindowOpenChild`, or zero for the root window. If the window handle is omitted, the currently selected window becomes the target OpenGL drawable.

itarget = 3 : To draw OpenGL graphics into a dialog field, of any type, specify the field identifier in *ident*. This must identify a field in the current dialog. If no identifier is specified or *ident* is zero, the whole of the current dialog becomes available for OpenGL graphics. It should be noted that dialog fields drawn in this way must be maintained by the calling program. Normal dialog fields will be repainted automatically, but it is the callers responsibility to repaint 'user drawn' fields. The `Expose` message reported by `WMessage` allows for this possibility. Dialog fields selected for OpenGL output need to be visible at the point when output occurs.

Drawing to the whole dialog (i.e. *ident* omitted or zero) introduces one complication under Windows. By default this will cause OpenGL graphics to overwrite any fields in the dialog window. If the 'Clip Fields' option is selected (see the Dialog Properties dialog in the resource editor), graphics will not overwite field contents, effectively drawing to the 'background' of the dialog. However, when 'Clip Fields' is enabled, the background of any group boxes become transparent causing rear windows to become visible through the dialog. We therefore do not recommend whole-dialog drawing (and hence the use of Clip Fields) with dialogs which contain group boxes. Under X Windows programs always behave as though the 'Clip Fields' option were enabled.

itarget = 0 : This option disables OpenGL output. The other arguments can be omitted in this case. Be sure to call this option when OpenGL output is complete.

Specifying an invalid identifier/handle in *ident*, when *itarget*=1 or 3, will cause error code 1019 to be set and OpenGL graphics will be disabled.

If the initialisation of OpenGL graphics fails for some reason other than a bad *ident* value, error code 1035 is set and a platform-specific error code will be available via `InfoError(3)`.

The initialisation of OpenGL graphics can be modified via the *iflags* argument. This is the sum of the following settings:

`wglColourIndex` : By default the RGBA color model is used. Setting this flag enables the alternative Color Index model.

`WglDoubleBuffer` : By default single buffered output is used, which is appropriate for static graphics displays. If animation is required, selecting this option will enable double buffering.

A selection of OpenGL demos are provided in W*i*SK's `OpenGL` directory. For more information about OpenGL graphics refer to "Graphics Interfaces : OpenGL" section in the W*i*SK on-line help file.

Example

```
USE WINTERACTER
USE OPENGL
      !
CALL WglSelect(DrawWindow)  ! Select current window for OpenGL
CALL draw_my_OpenGL_image() ! Draw OpenGL image
CALL WglSelect(Disabled)    ! End OpenGL output
```

Errors

`ErrBadTarget`	`(1019)`	Unknown target window handle

`ErrOpenGLInit (1035)`	OpenGL initialization failed

WglSwapBuffers Subroutine

Description
Swap front/back OpenGL buffers

Syntax
WglSwapBuffers()

Effect
Swaps the front/back buffers in double buffered OpenGL graphics output. This routine will normally only be used in animated OpenGL graphics to update the next frame of the animation.

Example
```
USE WINTERACTER
USE OPENGL
     !
CALL WglSelect(DrawWindow,0,wglDoubleBuffer)
     !
CALL glPopMatrix()
IF (doubleBuffer) THEN
    CALL WglSwapBuffers()
ELSE
    CALL glFlush()
END IF
```

WindowBell Subroutine

Description
Ring/enable/disable the bell.

Syntax
WindowBell(*onoff*)

Arguments
CHARACTER *onoff*	= 'ON'or ’OFF’ to enable/disable the bell
	Any other value to ring bell if currently enabled

Effect

By default the bell is enabled so a call to `WindowBell` with a blank argument would ring the bell. However, in some environments the bell can become irritating if used frequently. To stop `WindowBell` producing any sound, the on/off option is provided. This simply controls the action taken by `WindowBell` when an argument other than 'ON' or 'OFF' is supplied.

Example

```
LOGICAL :: ENABLE_BELL
IF (ENABLE_BELL) THEN
    CALL WindowBell('ON')
ELSE
    CALL WindowBell('OFF')
END IF
! now check state of bell
CALL WindowBell(' ')
```

WInitialise Subroutine

Description

Initialize W*interacter*.

Syntax

WInitialise(*initfile*)

Arguments

CHARACTER, OPTIONAL *initfile* = Initialization file name (not used in Starter Kit)

Effect

`WInitialise` must be called to initialize the library before calling any other W*interacter* screen i/o routines.

initfile can be omitted when linking with the Starter Kit version of W*interacter*.

`WInitialise` identifies the current screen dimensions and number of available screen colors. This information then becomes available via `WInfoScreen`. This information may prove useful when selecting the initial window size/position in the subsequent call to `WindowOpen`. Bear in mind that `WInitialise` performs no screen output and does not open a window. That is the task of `WindowOpen` and the other W*interacter* routines.

This routine should only called once per program run. Subsequent calls to `WInitialise` will therefore be ignored and no values will be altered.

Example

```
PROGRAM
! Variables, modules, etc. here
CALL WInitialise() ! Initialize
   :
! Winteracter program code
   :
STOP
END PROGRAM
```

WRGB Function

Description

Convert (r,g,b) triplet into a 24-bit integer color value.

Syntax

INTEGER WRGB(*ir*,*ig*,*ib*)

Arguments

INTEGER *ir* = Red component (0-255)

INTEGER *ig* = Green component (0-255)

INTEGER *ib* = Blue component (0-255)

Effect

Packs the supplied red, green and blue values into a 24-bit RGB color value, as used by most W*interacter* color handling routines.

The `WINTERACTER` module pre-defines names for eight primary color values which can be used in place of `WRGB`, namely `RGB_BLACK`, `RGB_RED`, `RGB_GREEN`, `RGB_YELLOW`, `RGB_BLUE`, `RGB_MAGENTA`, `RGB_CYAN` and `RGB_WHITE`.

Example

```
IRGB = WRGB(200,255,200) ! pale green
CALL IGrColourModel(24)
CALL IGrColourN(IRGB)
!
IRGB = WRGB(255,0,0)     ! these statements have
IRGB = RGB_RED           ! an identical effect
```

WRGBsplit Subroutine

Description

Split a 24-bit integer color value into an (r,g,b) triplet

Syntax

WRGBsplit(*rgb*,*ir*,*ig*,*ib*)

Arguments

INTEGER *rgb* = 24-bit color value

INTEGER, OPTIONAL *ir* = Returned red component (0-255)

INTEGER, OPTIONAL *ig* = Returned green component (0-255)

INTEGER, OPTIONAL *ib* = Returned blue component (0-255)

Effect

Splits the supplied 24-bit RGB color value into its component red, green and blue values. Each of the individual color component arguments is optional, so it is only necessary to retrieve those which are required.

Example

```
! get green component from a screen pixel
IRGB = IGrGetPixel(X,Y)
IF (IRGB /= -1) CALL WRGBsplit(IRGB,IG=IGREEN)
```

Group CH: Character Manipulation

The routines in this group are not strictly user interface functions. However, since any UI code involves considerable manipulation of textual information, they provide useful basic facilities such as string to numeric conversion, sub-string location, case conversion and so on.

IFillString Subroutine

Description

Fill a character string with a given character.

Syntax

IFillString(*string*,*chr*)

Arguments

CHARACTER *string* = String to be filled

CHARACTER *chr* = Character to fill string with (note: only first character of *chr* is used)

Effect

Fills the whole of *string* with the first character of *chr*.

Example

```
CHARACTER (LEN=80) :: STARS
CALL IFillString(STARS,'*')
```

IJustifyString Subroutine

Description

Justify a string within a character variable.

Syntax

IJustifyString(*string*,*lcr*)

Arguments

CHARACTER *string* = Variable containing string to justify (also receives returned string)

CHARACTER *lcr*	= Justification required:
	= '`L`' : Left justify (upper or lower case)
	= '`C`' : center justify (default)) (upper or lower case)
	= '`R`' : Right justify (upper or lower case)

Effect

Justifies a string within the character variable which holds it.

Note that in the sense used here, a "string" is defined as all characters from the first non-blank character to the last non-blank character within the character variable *string*. Since `IJustifyString` justifies the string within the supplied variable itself, *string* must be passed as a variable rather than as a literal string. If *string* is blank, `IJustifyString` takes no action.

Example

```
CHARACTER (LEN=14) :: TITLE
TITLE = ' Test Results  '
CALL IJustifyString(TITLE,'L')
! variable TITLE will now contain: 'Test Results  '
CALL IJustifyString(TITLE,'C')
! variable TITLE will now contain: ' Test Results '
CALL IJustifyString(TITLE,'R')
! variable TITLE will now contain: '  Test Results'
```

ILocateChar Function

Description

Locate position of first non blank character.

Syntax

INTEGER ILocateChar(*string*)

Arguments

CHARACTER *string* = String to search

Effect

Locates and returns the position (an INTEGER) of the first non-blank/non-null character within *string*. If the string contains only blanks and nulls, zero is returned.

Example

```
CHARACTER (LEN=20) :: FILNAM
CALL WDialogGetString(ID_FILE,FILNAM)
IPOS1 = ILocateChar(FILNAM)
```

ILocateString Subroutine

Description

Locate position of first non blank sub-string.

Syntax

ILocateString(*string*,*istart*,*iend*)

Arguments

CHARACTER *string* = String to search

INTEGER *istart* = Start position of first non-blank string

INTEGER *iend* = End position of first non-blank string

Locates the first sub-string within *string*, returning the start and end positions in *istart* and *iend*. If *string* is blank *istart* and *iend* are returned as zero. This routine is similar to the function `ILocateChar` except here the start and end positions are returned, rather than just the start position.

Example

```
CHARACTER (LEN=80) :: STRING
READ(LFN,'(A80)') STRING
CALL ILocateString(STRING,ISTART,IEND)
IF (ISTART>0) &
  CALL WindowOutString(100,300, &
            'First substring is '//STRING(ISTART:IEND))
```

ILowerCase Subroutine

Description
Convert a string to lower case.

Syntax
ILowerCase(*string*)

Arguments
CHARACTER *string* = String to be converted to lower case

Effect
Converts any upper case characters in *string* to lower case.

Example

```
CHARACTER (LEN=10) :: STRING
STRING = 'ABCDE12345'
CALL ILowerCase(STRING)
! string should now be abcde12345
```

IntegerToString Subroutine

Description
Convert an integer value to a string.

Syntax

IntegerToString(*ivalue*,*string*,*frmat*)

Arguments

INTEGER *ivalue* = Value to convert to a string

CHARACTER *string* = Character variable to receive numeric

CHARACTER *frmat* = Character string defining format to use
(a bracketed expression as in a Fortran FORMAT)

Effect

Converts an INTEGER value into a string using the specified Fortran format. If an error occurs, (e.g., *ivalue* is too large) *string* is filled with asterisks. `IntegerToString` is the reverse of `IStringToInteger`.

Example

```
CHARACTER (LEN=5) :: CHR
I = 100
CALL IntegerToString(I,CHR,'(I5)')
CALL WindowOutString(IX,IY,CHR)
```

Errors

`ErrNumToStr`	`(18)`	Numeric-to-string conversion error.

IStringToInteger Subroutine

Description

Convert a string to an integer value.

Syntax

IStringToInteger(*string*,*ivalue*)

Arguments

CHARACTER *string* = String containing number to be converted.

INTEGER *ivalue* = Value to be returned

Effect

Converts the first substring of *string* into an integer value. The numeric in *string* must be a valid INTEGER, optionally including a leading `+/-` sign. If an error occurs during conversion *ivalue* is returned as zero and the error flag is set. `IStringToInteger` is the reverse of `IntegerToString`.

Example

```
CHARACTER (LEN=80) :: LINE
CHARACTER (LEN=10) :: VALSTR
CALL WDialogGetString(IFIELD,LINE)
CALL IStringToInteger(LINE,IVALUE)
IF (InfoError(1)>0) THEN
  CALL WindowOutString(IX,IY,'Wrong !!')
ELSE
  CALL IntegerToString(IVALUE,VALSTR,'(I10)')
  CALL WindowOutString(IX,IY,'Value = '//VALSTR)
END IF
```

Errors

`ErrLargeNum`	(4)	Number too large (exceeds 4-byte INTEGER limits)
`ErrNoSubstring`	(10)	No substring found (*string* is blank)
`ErrBadChar`	(12)	Invalid character detected (i.e. not `0123456789` or leading `+/-`)

IUpperCase Subroutine

Description

Convert a string to upper case.

Syntax

IUpperCase(*string*)

Arguments

CHARACTER *string* = String to be converted to upper case

Effect

Converts any lower case characters in *string* to upper case.

Example

```
CHARACTER (LEN=10) :: STRING
STRING = 'abcde12345'
CALL IUpperCase(STRING)
! string should now be ABCDE12345
```

Group OB: Obsolete Routines

This group collects together a handful of routines which are now obsolete, but retained for backwards compatibility. In the main these are routines which were provided for *INTERACTER* compatibility or routines for which a better calling interface now exists. We recommend that these routines should not be used in new code and that existing usage be eliminated when convenient.

IActualLength Function

Description

Return actual length of string excluding trailing blanks or nulls.

Syntax

INTEGER IActualLength(*string*)

Arguments

CHARACTER *string* = String to search

Effect

Returns the actual length of the character string held in *string*, excluding any trailing spaces or nulls. If the string is completely blank, (i.e. only contains spaces and/or nulls) zero is returned. `IActualLength` offers an alternative to the Fortran 90 `LEN_TRIM` intrinsic which treats nulls as significant characters. It is mainly included for the sake of *INTERACTER* compatibility. Use of `LEN_TRIM` is normally recommended.

IGrCharJustify Subroutine

Description

Select graphics text justification.

Syntax

IGrCharJustify(*justif*)

Arguments

CHARACTER *justif*	= Justification mode for graphics text output:
	= `C`: centered (default)
	= `L`: Left justified
	= `R`: Right justified

Effect

Sets the justification to be used when outputting graphics text via `IGrCharOut`. This routine has been superseded by `WGrTextOrientation`.

IGrCharLength Function

Description

Measure the length of a graphics text string.

Syntax

REAL IGrCharLength(*string*)

Arguments

CHARACTER *string* = String or character to measure

Effect

When proportional spacing is enabled, this function returns the relative length of the specified string. This routine has been superseded by `WGrTextLength`.

IGrCharOut Subroutine

Description

Output character string at an absolute `(x,y)` position.

Syntax

IGrCharOut(*xpos*,*ypos*,*string*)

Arguments

REAL *xpos* = `X` co-ordinate

REAL *ypos* = `Y` co-ordinate

CHARACTER *string* = String to write

Effect

Outputs *string* at the graphics co-ordinate `(`*xpos*`,`*ypos*`)`. This routine has been superseded by `WGrTextString`.

IGrCharSet Subroutine

Description

Select graphics character set to use for text output.

Syntax

IGrCharSet(*filnam*)

Arguments

CHARACTER *filnam*	= Filename or string describing character set to use
	= '`H`' or '`h`': select hardware-dependent text (TrueType fonts)
	= `' '` : load/select default software character set
	= `'`*filename*`'`: load software character set from '*filename*'

Effect

Selects the character set to be used by future calls to `IGrCharOut` to output graphics text. This routine has been superseded by `WGrTextFont` and the various `WGrOFont`/ `WGrVFont` routines.

IGrCharSize Subroutine

Description

Select graphics text size.

Syntax

IGrCharSize(*xsize*,*ysize*)

Arguments

REAL *xsize* = Character width (`1.0` = base character width, equivalent to 75 per line)

REAL *ysize* = Character height (`1.0` = base character height, equivalent to 25 per column)

Effect

Sets the size of characters printed by `IGrCharOut`. Width/height values of `1.0` give standard size text, corresponding to 75 columns by 25 rows. This character size is independent of window size, ensuring a consistent character size, regardless of the resolution of the display. This routine has been superseded by the width and height arguments to `WGrTextFont`.

IGrCharSpacing Subroutine

Description
Select fixed or proportional spacing for graphics text.

Syntax
IGrCharSpacing(*fixprop*)

Arguments

CHARACTER *fixprop*	= Required character spacing:
	= F: Fixed (default) (can be upper or lower case)
	= P: Proportional (can be upper or lower case)

Effect
Selects fixed or proportional character spacing. This routine has been superseded by `WGrTextFont`.

IGrGetPixelRGB Subroutine

Description
Read a screen pixel color value as an (r,g,b) triplet

Syntax
IGrGetPixelRGB(*xpos*,*ypos*,*ired*,*igreen*,*iblue*)

Arguments
REAL *xpos* = X co-ordinate

REAL *ypos* = Y co-ordinate

INTEGER *ired* = Red component of specified point (`0-255`)

INTEGER *igreen*= Green component of specified point (`0-255`)

INTEGER *iblue* = Blue component of specified point (`0-255`)

Effect
Returns the color of the specified co-ordinate in the current drawable, as an (*r*,*g*,*b*) triplet. The `(x,y)` co-ordinate should be expressed in the normal user units as set via `IGrUnits`.If the specified co-ordinate lies outside the graphics area the RGB value is returned as (-1,-1,-1).

This routine has been superseded by the `IGrGetPixel` function.

IGrPaletteRGB Subroutine

Description

Redefine 8-bit color palette using an (r,g,b) triplet

Syntax

IGrPaletteRGB(*ncolor*,*ired*,*igreen*,*iblue*,*ipost*)

Arguments

INTEGER *ncolor* = Logical color number to which an actual color is to be assigned (same numbering scheme as `IGrColourN`)

INTEGER *ired* = Amount of red to assign to displayed color (`0-255`)

INTEGER *igreen* = Amount of green to assign to displayed color (`0-255`)

INTEGER *iblue* = Amount of blue to assign to displayed color (`0-255`)

INTEGER *ipost* = Postpone palette realisation on 256 color screen (0=no 1=yes)

Effect

Controls the 8-bit graphics color palette, using an (r,g,b) color triplet value. This routine has been superseded by `IGrPalette` which uses a 24-bit color value instead.

IGrPause Subroutine

Description

End of picture.

Syntax

IGrPause(*action*)

Arguments

CHARACTER *action* = String describing required action (default is clear window)
= '`P`': Preserve contents of graphics window

Effect

Sounds the bell and optionally clears the graphics window. This routine is included for *INTERACTER*-compatibility. There is little benefit in using it in W*interacter* applications.

WindowClearArea Subroutine

Description

Clear part of a window

Syntax

WindowClearArea(*ixtopl*,*iytopl*,*ixbotr*,*iybotr*)

Arguments

INTEGER *ixtopl* = Top left corner x co-ordinate

INTEGER *iytopl* = Top left corner y co-ordinate

INTEGER *ixbotr* = Bottom right corner x co-ordinate

INTEGER *iybotr* = Bottom right corner y co-ordinate

Effect

Clears the specified area of the current window to the currently selected background color. `WindowClear` now incorporates the functionality of this routine.

WindowOutString Subroutine

Description

Write text to a window

Syntax

WindowOutString(*ix*,*iy*,*string*)

Arguments

INTEGER *ix* = Horizontal start position (0-9999)

INTEGER *iy* = Vertical start position (0-9999)

CHARACTER *string* = String to write

Effect

Outputs *string* to the currently selected window, starting at the specified window co-ordinate, in the font selected by `WindowFont`. Text which extends beyond the right edge of the window will be truncated.

WindowStringLength Function

Description
Measure a string in window units

Syntax
INTEGER WindowStringLength(*string*)

Arguments
CHARACTER *string* = String to measure

Effect
Returns the length of the supplied string using the current font characteristics selected by `WindowFont`, in terms of W*interacter* units for the current window.

WindowFont Subroutine

Description
Select font and font attributes

Syntax
WindowFont(*font*)

Arguments
WIN_FONT *font* = Structure describing font characteristics

```
TYPE WIN_FONT
    INTEGER ifontnum  = Font number (0-6)
    INTEGER iwidth    = Average font width
    INTEGER iheight   = Font height
    INTEGER ibold     = Bold (0=no 1=yes)
    INTEGER italic    = Italics (0=no 1=yes)
    INTEGER iunder    = Underlined (0=no 1=yes)
    INTEGER ifcol     = Foregound color (-1 or 0-16)
    INTEGER ibcol     = Background color (-1 or 0-16)
 END TYPE WIN_FONT
```

Table 23: Font Numbers

Symbolic Name	Number	Windows Font	X Windows Font
SystemProp	0	System Proportional	Lucida *
SystemFixed	1	System Fixed	6x13
TimesNewRoman	2	Times New Roman	Times Roman

Effect

Sets the characteristics of the font to be used in future calls to `WindowOutString`.

This routine has been superseded by `WGrTextFont`.

WInfoFont Function

Description

Return font information.

Syntax

INTEGER WInfoFont(*item*)

Arguments
INTEGER *item* = Number of information item required.

Table 24: Font Information *items*

Name	No.	Information
FontXPos	1	Output cursor X position
FontYPos	2	Output cursor Y position
FontBold	3	Bold (0=off 1=on)
FontItalic	4	Italic (0=off 1=on)
FontUnderline	5	Underline (0=off 1=on)
FontForeCol	6	Foreground color index
FontBackCol	7	Background color index
FontStyleNum	8	Currently selected font number
FontWidth	9	Current font width
FontHeight	10	Current font height

Effect
Returns information about the obsolete `WindowFont` and `WindowOutString` routines.

WMenuRoot Subroutine

Description
Activate or remove a root menu structure.

Syntax
WMenuRoot(*menuid*)

Arguments
INTEGER *menuid* = Identifier of root menu to activate (0 to remove current root menu)

Effect
Activates the specified root menu structure, which will be attached to the top of the root window. The functionality of this routine has been supseded by the more general `WMenu`.

Index