Using Intel® Visual Fortran to Create and Build Windows*-Based Applications

Document Number: 324197-001US

www.intel.com

Legal Information

Contents

Legal Information..6

Chapter 1: Introduction
Overview..7
Notational Conventions...7
Related Information..8

Chapter 2: Creating Windowing Applications
Creating Windowing Applications Overview..9
Understanding Coding Requirements for Fortran Windowing Applications............9
Using Menus and Dialogs in SDI and MDI Fortran Windowing Applications.........13
Sample Fortran Windows Applications..15

Chapter 3: Creating and Using DLLs
Creating and Using Fortran DLLs Overview...16
Coding Requirements for Sharing Procedures in DLLs....................................16
Coding Requirements for Sharing Data in DLLs...18
Building Dynamic-Link Libraries...20
Building Executables that Use DLLs...21

Chapter 4: Using QuickWin
Using QuickWin Overview..23
Special Naming Convention for Certain QuickWin and Windows* Graphics Routines..24
Comparing QuickWin with Windows*-Based Applications................................25
Using Windows API Routines with QuickWin..25
Types of QuickWin Programs..25
QuickWin Programs Overview...25
Fortran Standard Graphics Applications..26
Fortran QuickWin Graphics Applications..27
The QuickWin User Interface..27
QuickWin User Interface Overview..27
Default QuickWin Menus...28
USE Statement Needed for Fortran QuickWin Applications...............................29
Creating QuickWin Windows...29
Creating QuickWin Windows Overview..29
Accessing Window Properties..30

Creating Child Windows....32
Giving a Window Focus and Setting the Active Window....33
Keeping Child Windows Open....34
Controlling Size and Position of Windows....35
Using QuickWin Graphics Library Routines....35
Using Graphics Library Routines....35
Selecting Display Options....36
Checking the Current Graphics Mode....36
Setting the Graphics Mode....36
Setting Figure Properties....37
Understanding Coordinate Systems....38
Understanding Coordinate Systems Overview....38
Text Coordinates....38
Graphics Coordinates....39
Setting Graphics Coordinates....42
Real Coordinates Sample Program....43
Advanced Graphics Using OpenGL....47
Adding Color....49
Adding Color Overview....49
Color Mixing....49
VGA Color Palette....51
Using Text Colors....52
Writing a Graphics Program....52
Writing a Graphics Program Overview....52
Activating a Graphics Mode....53
Drawing Lines on the Screen....54
Drawing a Sine Curve....55
Adding Shapes....56
Displaying Graphics Output....57
Displaying Graphics Output Overview....57
Drawing Graphics....57
Displaying Character-Based Text....59
Displaying Font-Based Characters....60
Using Fonts from the Graphics Library....61
Storing and Retrieving Images....64
Working With Screen Images....64
Transferring Images in Memory....64
Loading and Saving Images to Files....65
Editing Text and Graphics from the QuickWin Edit Menu....65
Customizing QuickWin Applications....66
Customizing QuickWin Applications Overview....66

Enhancing QuickWin Applications....66
Controlling Menus....67
Changing Status Bar and State Messages....70
Displaying Message Boxes....71
Defining an About Box....71
Using Custom Icons....71
Using a Mouse....72
QuickWin Programming Precautions....75
QuickWin Programming Precautions Overview....75
Using Blocking Procedures....75
Using Callback Routines....75
Simulating Nonblocking I/O....76

Chapter 5: Using Dialog Boxes for Application Controls

Using Dialog Boxes for Application Controls Overview....77
Using the Resource Editor to Design a Dialog Box....78
Designing a Dialog Box Overview....78
Setting Control Properties....82
Including Resources Using Multiple Resource Files....83
The Include (.FD and .H) Files....84
Writing a Dialog Application....84
Writing a Dialog Application Overview....84
Initializing and Activating the Dialog Box....85
Using Dialog Callback Routines....86
Using a Modeless Dialog Box....88
Using Fortran AppWizards to Help Add Modal Dialog Box Coding....89
Using Fortran AppWizards to Help Add Modeless Dialog Box Coding....91
Using Dialog Controls in a DLL....93
Summary of Dialog Routines....95
Understanding Dialog Controls....96
Understanding Dialog Controls Overview....96
Using Control Indexes....97
Available Indexes for Each Dialog Control....98
Specifying Control Indexes....101
Using Dialog Controls....102
Using Dialog Controls Overview....102
Using Static Text....103
Using Edit Boxes....103
Using Group Boxes....104
Using Check Boxes and Radio Buttons....104
Using Buttons....105

Using List Boxes and Combo Boxes....105
Using Scroll Bars....109
Using Pictures....109
Using Progress Bars....110
Using Spin Controls....110
Using Sliders....111
Using Tab Controls....111
Setting Return Values and Exiting....112
Using ActiveX* Controls....113
Using ActiveX* Controls Overview....113
Using the Resource Editor to Insert an ActiveX Control....113
Using the Intel® Fortran Module Wizard to Generate a Module....114
Adding Code to Your Application....114
Registering an ActiveX Control....117
Index....118

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.
Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.
The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.
Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.
Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

Centrino, Cilk, Intel, Intel Atom, Intel Core, Intel NetBurst, Itanium, MMX, Pentium, Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 1996-2011, Intel Corporation. All rights reserved.

Portions Copyright © 2001, Hewlett-Packard Development Company, L.P.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Introduction

1

Overview

This document describes how to use features specific to Windows* OS when building applications using the Intel® Visual Fortran Compiler.

Notational Conventions

This documentation uses the following conventions:

THIS TYPE	Indicates statements, data types, directives, and other language keywords. Examples of statement keywords are WRITE, INTEGER, DO, and OPEN.
this type	Indicates command-line or option arguments, new terms, or emphasized text. Most new terms are defined in the Glossary.
`This type`	Indicates a code example.
`This type`	Indicates what you type as input.
This type	Indicates menu names, menu items, button names, dialog window names, and other user-interface items.
File > Open	Menu names and menu items joined by a greater than (>) sign indicate a sequence of actions. For example, "Click **File>Open**" indicates that in the **File** menu, click **Open** to perform this action.
{value | value}	Indicates a choice of items or values. You can usually only choose one of the values in the braces.
[*item*]	Indicates items that are optional. Brackets are also used in code examples to show arrays.

item [, *item*]...	Indicates that the item preceding the ellipsis (three dots) can be repeated. In some code examples, a horizontal ellipsis means that not all of the statements are shown.
Windows* OS Windows operating system	These terms refer to all supported Windows* operating systems.
compiler option	This term refers to Windows* OS options that can be used on the compiler command line.

Related Information

For more information, see the Intel® Visual Fortran Compiler documentation set, and specifically the *Intel® Visual Fortran Compiler User and Reference Guides*.

These guides include the following information:

- Key features
- Compatibility and Portability
- Compilation
- Program Structure
- Compiler Reference
- Language Reference

Creating Windowing Applications 2

Creating Windowing Applications Overview

With Intel® Fortran, you can build Fortran applications that are also fully-featured Windows*-based applications. You can create full Windows-based applications that use the familiar Windows interface, complete with tool bars, pull-down menus, dialog boxes, and other features. You can include data entry and mouse control, and interaction with programs written in other languages or commercial programs such as Microsoft* Excel*.

With full Windows-based applications programming you can:

- Deliver Fortran applications with a Windows Graphical User Interface (GUI). GUI applications typically use at least the Graphic Device Interface (GDI) and USER32 Windows API routines.
- Access all available Windows GDI calls with your Fortran applications. GDI functions use a 32-bit coordinate system, allowing coordinates in the +/-2 GB range, and performs skewing, reflection, rotation and shearing.

Only the Fortran Windows project type provides access to the full set of Windows API routines needed to create GUI applications. Windows projects are much more complex than other kinds of Fortran projects. Before attempting to use the full capabilities of Windows programming, you should be comfortable with writing C applications and should familiarize yourself with the .NET Framework Software Development Kit (SDK).

To build your application as a Fortran Windows application in the visual development environment, choose **Windowing Application** from the list of Project types when you open a new project.

When using the command line, specify the `/winapp` option to search the commonly used link libraries.

Fortran Windows applications must use the IFWIN module or subset of IFWIN.

The following Fortran Windows application topics are discussed:

See Also

- Understanding Coding Requirements for Fortran Windowing Applications
- Using Menus and Dialogs in SDI and MDI Fortran Windowing Applications

Understanding Coding Requirements for Fortran Windowing Applications

This topic covers the following:

- General Coding Requirements: WinMain Function and USE Statements
- Code Generation Options Using the Fortran Windows Application Wizard

- Single Document Interface (SDI) or Multiple Document Interface (MDI) Sample Code

General Coding Requirements: WinMain Function and USE Statements

Coding requirements for Fortran Windowing applications include (in the following order):

1. WinMain function declaration and interface

The WinMain function declaration and interface are required for Windows Graphical User Interface (GUI) applications (typically use at least the GDI and USER32 Windows API routines). An interface block for the function declaration can be provided. The following function must be defined by the user:

```
INTEGER(DWORD) function WinMain (hInstance, hPrevInstance, &
& lpszCmdLine, nCmdShow)
use IFWIN
!DEC$ ATTRIBUTES STDCALL, DECORATE, ALIAS:"WinMain"  :: WinMain
INTEGER(HANDLE), INTENT(IN) :: hInstance, hPrevInstance
INTEGER(LPVOID), INTENT(IN) :: lpszCmdLine
INTEGER(DWORD), INTENT(IN) :: nCmdShow
```

In a program that includes a WinMain function, no program unit can be identified as the main program with the `PROGRAM` statement.

2. The statement `USE IFWIN` or other appropriate `USE` statements

The **USE IFWIN** statement makes available declarations of constants, derived types and procedure interfaces for much of the Windows Application Programming Interface (API) routines, also referred to as the Windows API. The Intel® Visual Fortran Compiler also provides individual modules for the various libraries that make up the Windows API, such as `KERNEL32` and `USER32`. Any Fortran program or subprogram that references declarations from the Windows API must include a USE statement for either `IFWIN`, which collects all the available modules, or the appropriate subset modules for the Windows API libraries that are used.

If you want to limit the type of parameters and interfaces for Windows applications or if unresolved references occur when linking your Fortran Windowing application, see the *Intel® Visual Fortran Compiler User and Reference Guides* for information on calling Windows API routines.

3. Data declarations for the WinMain function arguments.

4. Application-dependent code (other `USE` statements, variable declarations, and then executable code).

For example, consider the first lines of this sample, which uses free-form source code:

```
INTEGER(DWORD) function WinMain (hInstance, hPrevInstance, &
& lpszCmdLine, nCmdShow)
use IFWIN
!DEC$ ATTRIBUTES STDCALL, DECORATE, ALIAS:"WinMain"  :: WinMain
INTEGER(HANDLE), INTENT(IN) :: hInstance, hPrevInstance
INTEGER(LPVOID), INTENT(IN) :: lpszCmdLine
INTEGER(DWORD), INTENT(IN) :: nCmdShow
 .
 .
 .
```

IFWIN.F90 includes a Fortran version (a subset) of the Windows WINDOWS.H header file.

Code Generation Options Using the Fortran Windowing Application Wizard

When you choose the Fortran Windowing Application project type, you will need to select the type of project.

The following choices are available:

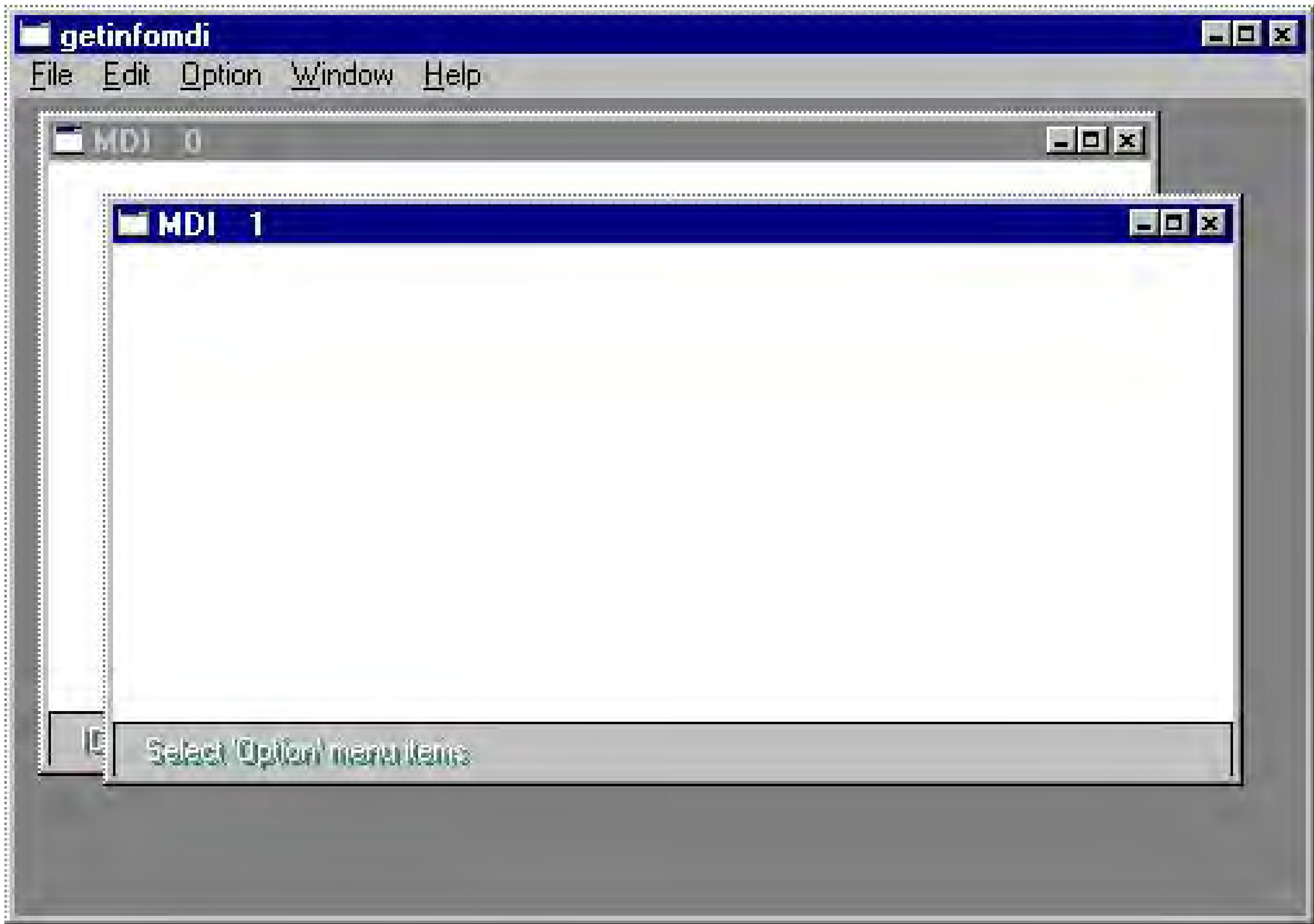
- Empty project
- Single Document Interface (SDI) sample code
- Multiple Document Interface (MDI) sample code
- Dialog-based sample code
- Single Document Interface (SDI) ActiveX* sample code
- Multiple Document Interface (MDI) ActiveX sample code
- Dialog-based ActiveX sample code

The ActiveX AppWizards will add additional template code for supporting ActiveX controls in your dialog boxes.

Single Document Interface (SDI) or Multiple Document Interface (MDI) Sample Code

Creating these types of application requires advanced programming expertise and knowledge of the Windows routines API. Such applications call certain library routines and require the statement USE IFWIN. SDI applications display a single window, whereas MDI application can display multiple windows (a main frame window with one or more child windows that appear within the frame window).

For example, select the MDI option from the Fortran AppWizard screen. After you build and run the application (without changing the source files), the following screen might appear after you create two child window by clicking New from the File menu twice:



If you selected the SDI option from the Fortran AppWizard screen and built and ran the application, you could not create child windows within the main window.

For more information:

- On using menus and dialogs from SDI and MDI Fortran Windowing applications, see Using Menus and Dialogs in SDI and MDI Fortran Windowing Applications
- About SDI and MDI Samples that use the Fortran Windowing project type, see Sample Fortran Windowing Applications.

Dialog-Based Sample Code

Dialog applications use a dialog box for the application's main window. Creating these applications requires some knowledge of the Windows routines API, but considerably less than for a SDI or MDI application. These applications call certain Intel Visual Fortran library routines and require the statement `USE IFLOGM`. Dialog-based applications usually do not have menus.

For example, select the Dialog-based applications from the Fortran AppWizard screen. After you build and run the application (without changing the source files), the following dialog box appears:

You can use dialogs in any project type. Dialogs don't require the use of a windowing application if you are using the module `IFLOGM`.

For more information:

- On using dialog boxes, see Using Dialog Boxes for Application Controls

Using Menus and Dialogs in SDI and MDI Fortran Windowing Applications

This section describes the following topics:

- Creating the Menu
- Using the Menu
- Handling Menu Messages
- Using Dialogs in an SDI or MDI Application

Creating the Menu

When you create a new SDI or MDI application, a default menu bar is created for you. The default menu bar contains many of the menu entries that are common to Windows applications. You can modify the default menu, or create a new menu, by using the Menu Editor, which is one of the Visual Studio Resource Editors. The Resource Editor is not available with the Visual Studio Shell.

To create a new menu resource (menu bar):

1. From the Insert menu, select Resource.
2. Select Menu as the resource type.

The menu bar consists of multiple menu names, where each menu name contains one or more items. You can use the Menu Editor to create submenus (select the pop-up property).

To edit an existing menu:

1. Double-click on the project .RC file.
2. Expand the Menu item in the list of resource types.
3. Double click on the menu name.

For more information about the menu resource editor, see the Visual C++ User's Guide section on Resource Editors.

Using the Menu

To use a menu bar in your Fortran application, you must load the menu resource and use it when creating the main window of the application. Code to do this is created automatically by the Fortran Windowing AppWizard. The code that loads the menu resource is:

```
ghMenu = LoadMenu(hInstance, LOC(lpszMenuName))
```

The returned menu handle is then used in the call to CreatWindowEx:

```
ghwndMain = CreateWindowEx( 0, lpszClassName,         &
                              lpszAppName,              &
                              INT(WS_OVERLAPPEDWINDOW),  &
                              CW_USEDEFAULT,            &
                              0,                        &
                              CW_USEDEFAULT,            &
                              0,                        &
                              NULL,                     &
                              ghMenu,                   &
                              hInstance,                &
                              NULL                      &
                            )
```

Handling Menu Messages

Windows sends a WM_COMMAND message to the main window when the user selects an item from the menu. The wParam parameter to the WM_COMMAND message contains:

- The low-order word specifies the identifier of the menu item
- The high-order word specifies either 0 if the message is from a menu item, or 1 if the message is the result of an accelerator key. It is usually not important to distinguish between these two cases, but you must be careful to compare against only the low-order word as in the example below.

For example, the following code from the main window procedure generated by the Fortran Windowing AppWizard handles the WM_COMMAND messages from the File menu Exit item and the Help menu About item:

```
! WM_COMMAND: user command
  case (WM_COMMAND)
   select case ( IAND(wParam, 16#ffff ) )

      case (IDM_EXIT)
          ret = SendMessage( hWnd, WM_CLOSE, 0, 0 )
           MainWndProc = 0
           return
         case (IDM_ABOUT)
           lpszName = "AboutDlg"C
           ret = DialogBoxParam(ghInstance,LOC(lpszName),hWnd,&
             LOC(AboutDlgProc), 0)
```

```
          MainWndProc = 0
          return
  ...
```

For advanced techniques with using menus, refer to the online Platform SDK section on User Interface Services.

Using Dialogs in an SDI or MDI Application

A Fortran Windowing SDI or MDI application that uses dialogs has the choice of using:

- Intel Visual Fortran Dialog routines
- native Windows APIs for creating dialog boxes

For any particular dialog box, you should use either the Intel Visual Fortran Dialog routines or the native Windows dialog box APIs. For example, if you create a dialog box using Windows APIs, you cannot use the Intel Visual Fortran dialog routines to work with that dialog box.

You should note, for example, that the code generated by the Fortran Windows AppWizard uses the native Windows APIs to display the About dialog box.

For more information:

- On using the Intel Visual Fortran Dialog routines, see Using Dialog Boxes for Application Controls
- On using the Windows APIs, see the online Platform SDK section on User Interface Services

Sample Fortran Windows Applications

The Intel Visual Fortran Samples folder contains many Fortran Windowing applications that demonstrate Windows* functionality or a particular Windows function. See the product release notes for the location of the Samples folder.

If you are unfamiliar with full Windows applications, start by looking at:

- Sample SDI and MDI Fortran Windows Samples in the `WIN32` folder, such as Generic, Platform, or Angle.
- Sample dialog Fortran Windows Samples in the `DIALOG` folder, such as TEMP and WHIZZY. For more information about coding requirements for dialog boxes and using the Dialog Resource editor, see Using Dialog Boxes for Application Controls.

Creating and Using DLLs

3

Creating and Using Fortran DLLs Overview

A dynamic-link library is a collection of source and object code and is similar in many ways to a static library. The differences between the two libraries are:

- The DLL is associated with a main project during execution, not during linking. Unlike a static library where routines are included in the base executable image during linking, the routines in a DLL are loaded when an application that references that DLL is loaded (run time).

A dynamic-link library (DLL) contains one or more subprogram procedures (functions or subroutines) that are compiled, linked, and stored separately from the applications using them. Because the functions or subroutines are separate from the applications using them, they can be shared or replaced easily.

Other advantages of DLLs include:

- You can change the functions in a DLL without recompiling or relinking the applications that use them, as long as the functions' arguments and return types do not change.

  This allows you to upgrade your applications easily. For example, a display driver DLL can be modified to support a display that was not available when your application was created.
- When general functions are placed in DLLs, the applications that share the DLLs can have smaller executables.
- Multiple applications can access the same DLL. This reduces the overall amount of memory needed in the system, which results in fewer memory swaps to disk and improves performance.
- Common blocks or module data placed in a DLL can be shared across multiple processes.

To build a DLL in the integrated development environment, specify the Fortran Dynamic-Link Library project type. On the command line, specify the `/dll` option.

You cannot make a QuickWin application into a DLL (see Using QuickWin) and QuickWin applications cannot be used with Fortran run-time routines in a DLL.

See Also

- Coding Requirements for Sharing Procedures in DLLs
- Coding Requirements for Sharing Data in DLLs
- Building Dynamic-Link Libraries
- Building Executables that Use DLLs

Coding Requirements for Sharing Procedures in DLLs

A dynamic-link library (DLL) contains one or more subprograms that are compiled, linked and stored separately from the applications using them.

Coding requirements include using the `cDEC$ ATTRIBUTES` compiler directive DLLIMPORT and DLLEXPORT options. Variables and routines declared in the main program and in the DLL are not visible to each other unless you use DLLIMPORT and DLLEXPORT.

This section discusses aspects of sharing subprogram procedures (functions and subroutines) in a Fortran DLL.

To export and import each DLL subprogram:

1. Within your Fortran DLL, export each subprogram that will be used outside the DLL. Add `!DEC$ ATTRIBUTES DLLEXPORT` to declare that a function, subroutine, or data is being exported outside the DLL. For example:

```
SUBROUTINE ARRAYTEST(arr)
!DEC$ ATTRIBUTES DLLEXPORT :: ARRAYTEST
REAL arr(3, 7)
INTEGER i, j
DO i = 1, 3
DO j = 1, 7
arr (i, j) = 11.0 * i + j
END DO
END DO
END SUBROUTINE
```

2. Within your Fortran application, import each DLL subprogram. Add `!DEC$ ATTRIBUTES DLLIMPORT` to declare that a function, subroutine, or data is being imported from outside the current image. For example:

```
INTERFACE
SUBROUTINE ARRAYTEST (rarray)
!DEC$ ATTRIBUTES DLLIMPORT :: ARRAYTEST
REAL rarray(3, 7)
END SUBROUTINE ARRAYTEST
END INTERFACE
CALL ARRAYTEST (rarray)
```

   Or, not using an INTERFACE block:

```
PROGRAM TESTA
!DEC$ ATTRIBUTES DLLIMPORT:: ARRAYTEST
REAL rarray (3,7)
CALL ARRAYTEST(rarray)
END PROGRAM TESTA
```

   The DLLEXPORT and DLLIMPORT options (for the cDEC$ ATTRIBUTES directive) tell the linker that a procedure, variable or COMMON block is to be visible in a DLL, or that it can be found in a DLL.

   The DLLEXPORT property declares that functions or data are being exported to other images or DLLs, usually eliminating the need for a Linker module definition (.DEF) file to export symbols for the functions or subroutines declared with DLLEXPORT. When you declare a function, subroutine, or data with the DLLEXPORT property, it must be defined in the same module of the same program.

   A program that uses symbols defined in another image (such as a DLL) must import them. The DLL user needs to link with the import LIB file from the other image and use the DLLIMPORT property inside the application that imports the symbol. The DLLIMPORT option is used in a declaration, not a definition, because you do not define the symbol you are importing.

3. Build the DLL and then build the main program, as described in Building Dynamic-Link Libraries.

Fortran and C applications can call Fortran and C DLLs provided the calling conventions are consistent.

See Also

- Building Dynamic-Link Libraries
- Coding Requirements for Sharing Data in DLLs
- Exporting and Importing Data Objects in Modules

Coding Requirements for Sharing Data in DLLs

A dynamic-link library (DLL) is an executable file that can be used as a place to share data across processes.

Coding requirements include using the `cDEC$ ATTRIBUTES` compiler directive DLLIMPORT and DLLEXPORT options. Variables and routines declared in the program and in the DLL are not visible to each another unless you use DLLIMPORT and DLLEXPORT.

When sharing data among multiple threads or processes, do the following:

- Declare the order, size, and data types of shared data consistently in the DLL and in all procedures importing the DLL exported data.
- If more than one thread or process can write to the common block simultaneously, use the appropriate features of the Windows* operating system to control access to the shared data. Such features include critical sections (for single process, multiple thread synchronization) and mutex objects (for multi-process synchronization).

This section discusses:

- Exporting and Importing Common Block Data
- Exporting and Importing Data Objects in Modules

Exporting and Importing Common Block Data

Data and code in a dynamic-link library is loaded into the same address space as the data and code of the program that calls it. However, variables and routines declared in the program and in the DLL are not visible to one another unless you use the *c***DEC$ ATTRIBUTES** compiler directive DLLIMPORT and DLLEXPORT options. These directive options enable the compiler and linker to map to the correct portions of the address space so that the data and routines can be shared, allowing use of common block data across multiple images.

You can use DLLEXPORT to declare that a common block in a DLL is being exported to a program or another DLL. Similarly, you can use DLLIMPORT within a calling routine to tell the compiler that a common block is being imported from the DLL that defines it.

To export and import common block data:

1. Create a common block in the subprogram that will be built into a Fortran DLL. Export that common block with `!DEC$ ATTRIBUTES DLLEXPORT`, followed by the `COMMON` statement, associated data declarations, and any procedure declarations to be exported. For example:

```
!DEC$ ATTRIBUTES DLLEXPORT :: /X/
COMMON /X/ C, B, A
REAL C, B, A
END
...
```

If the Fortran DLL procedure contains only a common block declaration, you can use the **BLOCK DATA** statement:

```
BLOCK DATA T
!DEC$ ATTRIBUTES DLLEXPORT :: /X/
COMMON /X/ C, B, A
REAL C, B, A
END
```

The Fortran procedure to be linked into a DLL can contain a procedure, such as the following:

```
  SUBROUTINE SETA(I)
!DEC$ ATTRIBUTES DLLEXPORT :: SETA, /X/
COMMON /X/ C, B, A
REAL C, B, A
INTEGER I
A = A + 1.
I = I + 1
WRITE (6,*) 'In SETA subroutine, values of A and I:' , A, I
RETURN
END SUBROUTINE
```

2. Refer to the common block in the main program with `!DEC$ ATTRIBUTES DLLIMPORT`, followed by the local data declarations and any procedure declarations defined in the exported DLL. For example:

```
 PROGRAM COMMONX
!DEC$ ATTRIBUTES DLLIMPORT:: SETA,  /X/
COMMON /X/ C, B, A
REAL C, B, A, Q
EQUIVALENCE (A,Q)
A = 0.
I = 0
WRITE (6,*) 'In Main program before calling SETA...'
WRITE (6,*) 'values of A and I:' , A, I
CALL SETA(I)
WRITE (6,*) 'In Main program after calling SETA...'
WRITE (6,*) 'values of A and I:' , Q, I     A
 = A + 1.
I = I + 1
WRITE (6,*) 'In Main program after incrementing values'
END PROGRAM COMMONX
```

3. Build the DLL and then build the main program, as described in Building Dynamic-Link Libraries.

Exporting and Importing Data Objects in Modules

You can give data objects in a module the DLLEXPORT property, in which case the object is exported from a DLL.

When a module is used in other program units, through the `USE` statement, any objects in the module with the DLLEXPORT property are treated in the program using the module as if they were declared with the DLLIMPORT property. So, a main program that uses a module contained in a DLL has the correct import attributes for all objects exported from the DLL.

You can also give some objects in a module the DLLIMPORT property. Only procedure declarations in `INTERFACE` blocks and objects declared `EXTERNAL` or with *c*`DEC$ ATTRIBUTES` EXTERN can have the DLLIMPORT property. In this case, the objects are imported by any program unit using the module.

If you use a module that is part of a DLL and you use an object from that module that does not have the DLLEXPORT or DLLIMPORT property, the results are undefined.

For more information:

- On building a DLL, see Building Dynamic-Link Libraries.

Building Dynamic-Link Libraries

When you first create a DLL, create a new project, and select Fortran Dynamic-Link Library as the project type.

To debug a DLL, you must use a main program that calls the library routines (or references the data). From the Project Property Pages dialog box, choose the Debugging category. A dialog box is available for you to specify the executable for a debug session.

To build the DLL from the Microsoft integrated development environment (IDE):

1. A Fortran DLL project is created like any other project, but you must specify Dynamic-Link Library as the project type.
2. Add files to your Fortran DLL project. Include the DLL Fortran source that exports procedures or data as a file in your project.
3. If your DLL exports data, consistently specify the project settings options in the Fortran Data compiler option category for both the DLL and any image that references the DLL's exported data. In the Fortran Data compiler option category, specify the appropriate values for Common Element Alignment (common block data) and Structure Member Alignment (structures in a module). This sets the `/align` option, which specifies whether padding is needed to ensure that exported data items are naturally aligned.

   For example, in the case of a common block containing four-byte variables, you might:

   - Open the appropriate solution and select the project in the Solution View.
   - From the **Project>Properties**, select the Fortran category.
   - Select Data.
   - In the Common Element Alignment box, select 4 Bytes.

4. If you need to specify linker options, use the Linker category of the Project Property Pages dialog box.
5. Build your Fortran DLL project.

   The IDE automatically selects the correct linker instructions for loading the proper run-time library routines (located in a DLL themselves). Your DLL is created as a multithread-enabled library. An import library (.LIB) is created for use when you link images that reference the DLL.

To build the DLL from the command line:

1. If you build a DLL from the command line or use a makefile, you must specify the /dll option. For example, if the Fortran DLL source code is in the file `f90arr.f90`, use the following command line:

   ```
   ifort /dll f90arr.f90
   ```

   This command creates:

   - A DLL named `f90arr.dll`.
   - An import library, `f90arr.lib`, that you must link with applications that call your DLL.

   If you also specify `/exe:`*`file`* or `/link /out:`*`file`*, the file name you specify is used for a .DLL rather than an .EXE file (the default file extension becomes *`projectname`*.DLL instead of *`projectname`*.EXE)

   The `/dll` option selects, as the default, the DLL run-time libraries to support multithreaded operation.

2. If your DLL will export data, the procedures must be compiled and linked consistently. Consistently use the same `/align` option for the DLL export procedure and the application that references (imports) it. The goal is to specify padding to ensure that exported data items are naturally aligned, including common block data items and structure element alignment (structures in a module).

3. If you need to specify linker options, place them after the `/link` option on the ifort command line.
4. Build the application. For example, if your DLL exports a common block containing four-byte variables, you might use the following command line (specify the `/dll` option):

```
ifort /align:commons /dll dllfile.for
```

The `/dll` option automatically selects the correct linker instructions for loading the proper run-time library routines (located in a DLL themselves). Your DLL is created as a multithread-enabled library.

The DLL Build Output

When a DLL is built, two library files are typically created:

- An import library (.LIB), which the linker uses to associate a main program with the DLL.
- The .DLL file containing the library's executable code.

Both files have the same basename as the library project by default.

For a build from the command line, your library routines are contained in the file *projectname*.DLL located in the default directory for your project, unless you specified another name and location. Your import library file is *projectname*.LIB, located in the default directory for your project.

For a build from the Microsoft* Visual Studio* integrated development environment, both the library routines and the import library file are located in the output directory of the project configuration.

NOTE. If the DLL contains no exported routines or data, the import library is not created.

Checking the DLL Symbol Export Table

To make sure that everything that you want to be visible shows up in the export table, look at the export information of an existing DLL *file* by using QuickView in the Windows Explorer **File** menu or the following DUMPBIN command:

```
DUMPBIN /exports file.dll
```

Building Executables that Use DLLs

When you build the executable that imports the procedures or data defined in the DLL, you must link using the import library, check certain project settings or command-line options, and then build the executable.

To use the DLL from another image:

1. Add the import .LIB file with its path and library name to the other image.

   In the integrated development environment, add the .LIB import library file to your project. In the Project menu, select Add Existing Item... . If the importing project and the DLL are in the same solution, you can add the DLL project as a dependency of the importing project instead.

   On the command line, specify the .LIB file on the command line.

   The import .LIB file contains information that your program needs to work with the DLL.

2. If your DLL exports data, consistently use the same property page options in the Fortran Data category `/align` option as was used to create the DLL. In the Fortran Data category, specify the appropriate values for Common Element Alignment (common block data) and Structure Member Alignment (structures in a module). This sets the `/align` option, which specifies whether padding is needed to ensure that imported data items are naturally aligned.
3. In the Project Property Pages dialog box, make sure the type of libraries specified is consistent with that specified for the Fortran DLL.
4. If you need to specify linker options:
   - In the IDE, specify linker options in the Linker category.
   - On the ifort command line, place linker options after the `/link` option.
5. Copy the DLL into your path.

   For an application to access your DLL, it must be located in a directory included in the PATH system environment variable or in the same directory as the executable. If you have more than one program accessing your DLL, you can keep it in a convenient directory identified in the environment variable PATH. If you have several DLLs, you can place them all in the same directory to avoid adding numerous directories to the path specification.

   You should log out and back in after modifying the system path.
6. Build the image that references the DLL.

   When using the visual development environment:

   - Like building other projects in the integrated development environment, use the **Build** menu items to create the executable.

   When using the command line:

   - Specify the import library at the end of the command line.
   - If your DLL exports data that will be used by the application being built, specify the same `/align` options that were used to build the DLL.
   - If you are building a main application, omit the `/dll` option.
   - When building a Fortran DLL that references another DLL, specify the `/dll` option.

   For example, to build the main application from the command line that references 4-byte items in a common block defined in dllfile.dll:

```
ifort /align:commons mainapp.f90 dllfile.lib
```

Using QuickWin 4

Using QuickWin Overview

This section introduces the major categories of QuickWin library routines. It gives an overview of QuickWin features and their use in creating and displaying graphics, and customizing your QuickWin applications with custom menus and mouse routines. Drawing Graphics and Using Fonts from the Graphics Library cover graphics and fonts in more detail.

The Intel® Fortran QuickWin library helps you give traditional console-oriented Fortran programs a Windows look and feel, with a scrollable window and a menu bar. Though the full capability of Windows is not available through QuickWin, QuickWin is simpler to learn and to use. QuickWin applications support pixel-based graphics, real-coordinate graphics, text windows, character fonts, user-defined menus, mouse events, and editing (select/copy/paste) of text, graphics, or both.

You can use the QuickWin library to do the following:

- Compile console programs into simple applications for Windows.
- Minimize and maximize QuickWin applications like any Windows-based application.
- Call graphics routines.
- Load and save bitmaps.
- Select, copy and paste text, graphics, or a mix of both.
- Detect and respond to mouse clicks.
- Display graphics output.
- Alter the default application menus or add programmable menus.
- Create custom icons.
- Open multiple child windows.

In Intel Visual Fortran, graphics programs must be either Fortran QuickWin, Fortran Standard Graphics, Fortran Windows, or use OpenGL routines. Fortran Standard Graphics Applications are a subset of QuickWin that support only one window.

You can choose the Fortran QuickWin or Standard Graphics application type from the list of available project types when you create a new project in the visual development environment. Or you can use the `/libs:qwin` compiler option for Fortran QuickWin or the /libs:qwins compiler option for Fortran Standard Graphics.

Note that Fortran QuickWin and Standard Graphics applications cannot be DLLs, and QuickWin and Standard Graphics cannot be linked with run-time routines that are in DLLs. This means that the `/libs:qwin` option and the `/libs:dll` with `/threads` options cannot be used together.

You can access the QuickWin routines library from Intel Fortran as well as other languages that support the Fortran calling conventions.

A program using the QuickWin routines must explicitly access the QuickWin graphics library routines with the statement **USE IFQWIN** (see USE Statement Needed for Fortran QuickWin Applications).

Special Naming Convention for Certain QuickWin and Windows* Graphics Routines

Most QuickWin routines have a QQ appended to their names to differentiate them from equivalent Windows operating system routines. However, a small group of QuickWin graphics routines have the same name as the Windows routines, causing a potential naming conflict if your program unit includes both USE IFLIBS (which includes QuickWin routine interface definitions) and USE IFWIN (which includes Windows API routine interface definitions).

The QuickWin routines perform the same functions as the SDK routines but take a unit number, or use the unit in focus at the time of call, instead of taking a device context (DC) as one of their arguments.

To handle this situation, a special MSFWIN$ prefix is used for the Windows routines. These prefixed names must be used even if you only specify **USE IFWIN**.

For example, Rectangle is a QuickWin routine, not a Windows SDK routine, and you must use the name MSFWIN$Rectangle to refer to the SDK routine:

QuickWin Routine	Windows API Routine
ARC	MSFWIN$Arc
ELLIPSE	MSFWIN$Ellipse
FLOODFILL	MSFWIN$FloodFill
GETBKCOLOR	MSFWIN$GetBkColor
GETPIXEL	MSFWIN$GetPixel
GETTEXTCOLOR	MSFWIN$GetTextColor
LINETO	MSFWIN$LineTo
PIE	MSFWIN$Pie
POLYGON	MSFWIN$Polygon
RECTANGLE	MSFWIN$Rectangle
SETBKCOLOR	MSFWIN$SetBkColor
SETPIXEL	MSFWIN$SetPixel
SETTEXTCOLOR	MSFWIN$SetTextColor

Comparing QuickWin with Windows*-Based Applications

One decision you must make when designing a program is how it will be used. If the person using your program must interact with it, the method of interaction can be important. Anytime the user must supply data, that data must be validated or it could cause errors. One way to minimize data errors is to allow the user to select a value from a list. For example, if the data is one of several known values, the user can select the desired value instead of typing it in.

When you design programs to be interactive, you use a different structure than if you design them to be run in unattended batches. Interactive applications behave more like state machines than numerical algorithms, because they perform the actions you request when you request them. You may also find that once you can change what your program is doing while it runs, you will be more likely to experiment with it.

The QuickWin library lets you build simple Windows applications. Because QuickWin is a wrapper around a subset of the Windows API, there are limitations to what you can do, but it can fulfill the requirement of most users. If you need additional capabilities, you can call the Windows API directly rather than using QuickWin to build your program. You can also build a graphic user interface in either Microsoft* Visual C++* or Visual Basic* that calls your Fortran code.

QuickWin applications do not provide the total capability of Windows*-based applications. Although you can call many Windows APIs (Application Programming Interface) from QuickWin and console programs, many other Windows APIs (such as GDI functions) should be called only from a full Windows-based application. You need to use Windows-based applications, not QuickWin, if any of the following applies:

- Your application has an OLE* (Object Linking and Embedding) container.
- You want direct access to GDI (Graphical Data Interface) functions.
- You want to add your own customized Help information to QuickWin Help.
- You want to create something other than a standard SDI (Single Document Interface) or MDI (Multiple Document Interface) application. (For example, if you want your application to have a dialog such as Windows Calculator in the client area.)
- You want to use a modeless dialog box rather than a modal dialog box.

Using Windows API Routines with QuickWin

You can convert the unit numbers of QuickWin windows to Windows handles with the `GETHWNDQQ` QuickWin function. You should not use Windows GDI to draw on QuickWin windows because QuickWin keeps a window buffer and the altered window would be destroyed on redraw. You can use Windows OS subclassing to intercept graphics messages bound for QuickWin before QuickWin receives them.

Types of QuickWin Programs

QuickWin Programs Overview

You can create a Fortran Standard Graphics application or a Fortran QuickWin application, depending on the project type you choose. Standard Graphics (QuickWin single document) applications support only one window and do not support programmable menus. Fortran QuickWin applications support multiple windows and user-defined menus. Any Fortran program, whether it contains graphics or not, can be compiled as a QuickWin application. You can use the Microsoft integrated development environment (IDE) to create, debug, and execute Fortran Standard Graphics programs and Fortran QuickWin programs.

To build a Fortran QuickWin application in the IDE, select **QuickWin Application** from the list of available project types displayed when you create a new project. You can choose to create an Empty QuickWin project (multiple-windows) or a Fortran Standard Graphics application.

To build a Fortran QuickWin application from the command line, use the `/libs:qwin` option. For example:

```
ifort /libs:qwin qw_app.f90
```

To build a Fortran Standard Graphics application from the command line, use the `/libs:qwins` option. For example:

```
ifort /libs:qwins stdg_app.f90
```

See Also

- Fortran Standard Graphics Application
- Fortran QuickWin Application

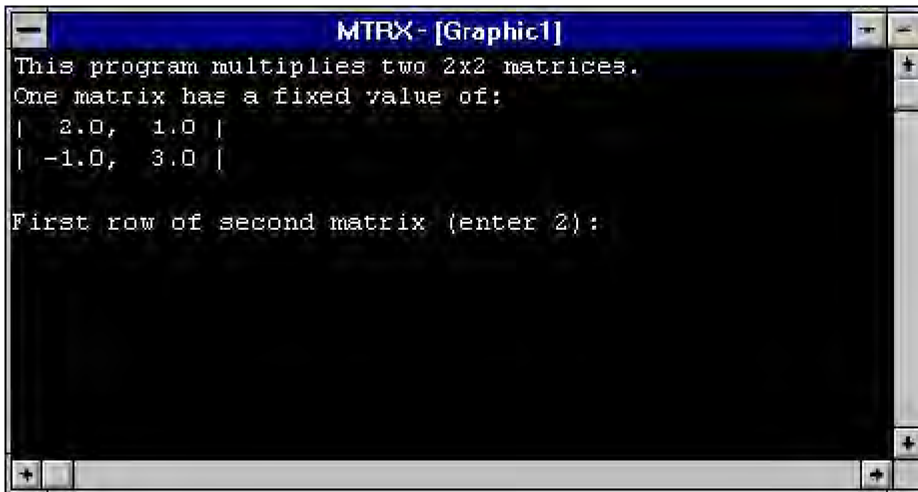
Fortran Standard Graphics Applications

A Fortran standard graphics application has a single maximized application window covering the entire screen area, whose appearance resembles a MS-DOS* screen without scrolls bars or menus. The `Esc` key can be used to exit a program that does otherwise terminate. When the `Esc` key is pressed, the frame window appears with a border, title bar, scroll bars, and a menu item in the upper-left corner that allows you to close the application.

Programmable menus and multiple child windows cannot be created in this mode.

The following figure shows a typical Fortran Standard Graphics application, which resembles an MS-DOS application running in a window.

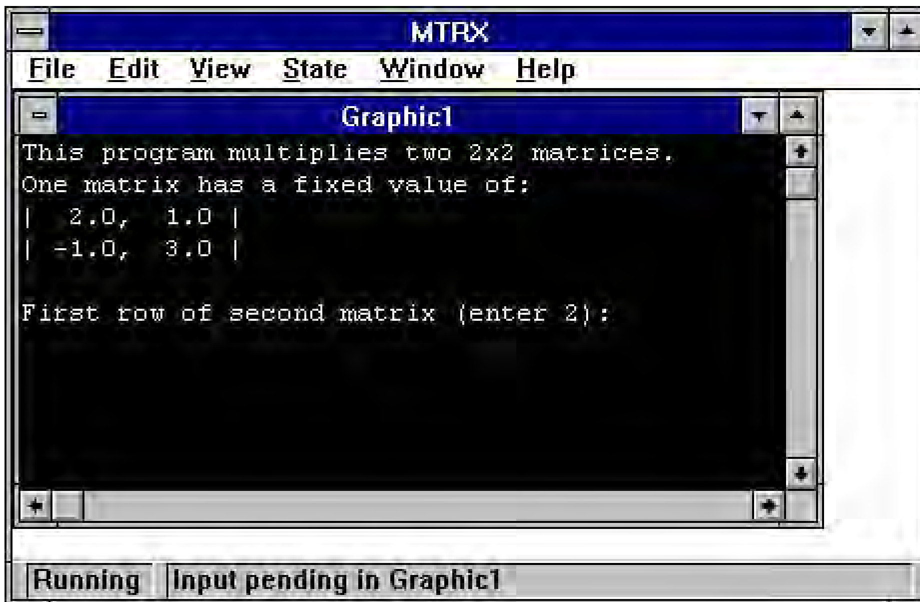
MTRX.F90 Compiled as a Fortran Standard Graphics Application



Fortran QuickWin Graphics Applications

The following shows a typical Fortran QuickWin application. The frame window has a border, title bar, scroll bars, and default menu bar. You can modify, add, or delete the default menu items, respond to mouse events, and create multiple child windows within the frame window using QuickWin enhanced features. Routines to create enhanced features are listed in Enhancing QuickWin Applications. Using these routines to customize your QuickWin application is described in Customizing QuickWin Applications.

MTRX.F90 Compiled as a QuickWin Application



The QuickWin User Interface

QuickWin User Interface Overview

All QuickWin applications create an application or frame window; child windows are optional. Fortran Standard Graphics applications and Fortran QuickWin applications have these general characteristics:

- Window contents can be copied as bitmaps or text to the Clipboard for printing or pasting to other applications. In Fortran QuickWin applications, any portion of the window can be selected and copied.
- Vertical and horizontal scroll bars appear automatically, if needed.
- The base name of the application's .EXE file appears in the window's title bar.
- Closing the application window terminates the program.

In addition, the Fortran QuickWin application has a status bar and menu bar. The status bar at the bottom of the window reports the current status of the window program (for example, running or input pending).

Default QuickWin Menus shows the default QuickWin menus.

Default QuickWin Menus

The default MDI (Multiple Document Interface) menu bar has the following menus:

- File
- Edit
- View
- State
- Window

File Menu

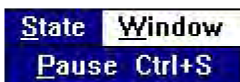
Edit Menu

For instructions on using the Edit options within QuickWin see Editing Text and Graphics from the QuickWin Edit Menu.

View Menu

The resulting graphics might appear somewhat distorted whenever the logical graphics screen is enlarged or reduced with the Size to Fit and Full Screen commands. While in Full Screen or Size To Fit mode, cursors are not scaled.

State Menu



Window Menu

For instructions on replacing the About default information within the Help menu with your own text message, see Defining an About Box.

For instructions on how to create custom QuickWin menus, see Customizing QuickWin Applications.

USE Statement Needed for Fortran QuickWin Applications

A program using the Fortran QuickWin or Standard Graphics features must explicitly access the QuickWin graphics library routines with the statement `USE IFQWIN`.

Any program using the QuickWin features must include the statement `USE IFQWIN` to access the QuickWin graphics library. The IFQWIN.MOD module file contains subroutine and function declarations in `INTERFACE` statements, derived-type declarations, and symbolic constant declarations for each QuickWin routine.

`USE` statements must immediately follow the `PROGRAM`, `SUBROUTINE` or `FUNCTION` statement, and precede any `IMPLICIT` statement.

Depending on the type of routines used by your application, other `USE` statements that include other Fortran modules may be needed. The description of each Intel Fortran routine in the *Language Reference* indicates the module that needs to be included for external routines (such as `USE IFCORE`; `USE IFPORT`).

Creating QuickWin Windows

Creating QuickWin Windows Overview

The QuickWin library contains many routines to create and control your QuickWin windows.

See Also

- Accessing Window Properties
- Creating Child Windows
- Giving a Window Focus and Setting the Active Window
- Keeping Child Windows Open
- Controlling Size and Position of Windows

Accessing Window Properties

`SETWINDOWCONFIG` and `GETWINDOWCONFIG` set and get the current *`virtual window`* properties. Virtual window properties set by SETWINDOWCONFIG contain the maximum amount of text and graphics for that unit. The `SETWSIZEQQ` routine sets the properties of the *`visible window`*, which is generally smaller than a virtual window.

If the size of the virtual window (`SETWINDOWCONFIG`) is larger than the size of the visible window (`SETWSIZEQQ`), scroll bars are automatically provided to allow all the text and graphics in a virtual window to be displayed.

These virtual window properties are stored in the `windowconfig` derived type, which contains the following parameters:

```
TYPE windowconfig
    INTEGER(2) numxpixels            ! Number of pixels on x-axis.
    INTEGER(2) numypixels            ! Number of pixels on y-axis.
    INTEGER(2) numtextcols           ! Number of text columns available.
    INTEGER(2) numtextrows           ! Number of scrollable text lines available.
    INTEGER(2) numcolors             ! Number of color indexes.
    INTEGER(4) fontsize              ! Size of default font. Set to
                                     ! QWIN$EXTENDFONT when using multibyte
                                     ! characters, in which case
                                     ! extendfontsize sets the font size.
    CHARACTER(80) title              ! Window title, where title is a C string.
    INTEGER(2) bitsperpixel          ! Number of bits per pixel. This value
                                     ! is calculated by the system and is an
                                     ! output-only parameter.
                                     ! The next three parameters support multibyte
                                     ! character sets (such as Japanese)
    CHARACTER(32) extendfontname     ! Any non-proportionally spaced font
                                     ! available on the system.
    INTEGER(4) extendfontsize        ! Takes same values as fontsize, but
                                     ! used for multiple-byte character sets
                                     ! when fontsize set to QWIN$EXTENDFONT.
    INTEGER(4) extendfontattributes  ! Font attributes such as bold and
                                     ! italic for multibyte character sets.
  END TYPE windowconfig
```

If you use `SETWINDOWCONFIG` to set the variables in `windowconfig` to -1, the highest resolution will be set for your system, given the other fields you specify, if any. You can set the actual size of the window by specifying parameters that influence the window size -- the number of *x* and *y* pixels, the number of rows and columns, and the font size. If you do not call `SETWINDOWCONFIG`, the window defaults to the best possible resolution and a font size of 8 by 16. The number of colors depends on the video driver used.

The font size, x pixels, y pixels, and columns and rows are related and cannot all be set arbitrarily. The following example specifies the number of *x* and *y* pixels and the font size and accepts the system calculation for the best number of rows and columns for the window:

```
USE IFQWIN
    TYPE (windowconfig) wc
    LOGICAL status
  ! Set the x & y pixels to 800X600 and font size to 8x12.
    wc%numxpixels  = 800              ! pixels on x-axis, window width
    wc%numypixels  = 600              ! pixels on y-axis, window height
    wc%numtextcols = -1               ! -1 requests system default/calculation
    wc%numtextrows = -1
    wc%numcolors   = -1
    wc%title       = " "C
    wc%fontsize    = #0008000C        ! Request 8x12 pixel fonts
    status         = SETWINDOWCONFIG(wc)
```

In this example:

- The variables wc%numxpixels and wc%numypixels specify the size of the window, in this case 800 by 600 pixels. Within this window size, you can choose to specify either the font size (wc%fontsize) or the number of text columns (wc%numtextcols) and rows (wc%numtextrows).

  This example specifies the window size and font size, and lets the system calculate the number of text columns and rows.

  If you choose to specify the number of text columns and rows, you can let the system calculate (specify -1) either the font size or the window size.
- The variable wc%fontsize is given as hexadecimal constant of #0008000C, which is interpreted in two parts:
  - The left side of 0008 (8) specifies the width of the font, in pixels.
  - The right side of 000C (12 in decimal) specifies the height of the font, in pixels.
- The variable wc%numtextrows is -1 and wc%numtextcols is -1, which allows the system to calculate the best available number of text columns and text rows to be used, as follows:
  - The number of text columns is calculated as wc%numypixels (800) divided by the width of the font 8 (decimal) or 100.
  - The number of text rows is calculated as wc%numxpixels (600) divided by the width of the font, 12 (decimal) or 50.

The requested font size is matched to the nearest available font size. If the matched size differs from the requested size, the matched size is used to determine the number of columns and rows.

If scroll bars are needed (virtual window size exceeds the visible window size), because of the size required by horizontal and vertical scroll bars for a window, you may need to set the number of lines and columns to a value 1 or 2 greater than the number of rows and columns needed to display the application's information.

If the requested configuration cannot be set, SETWINDOWCONFIG returns .FALSE. and calculates parameter values that will work and best fit the requested configuration. Another call to SETWINDOWCONFIG establishes these values:

```
IF(.NOT.status) status = SETWINDOWCONFIG(wc)
```

For information on setting the graphics mode with SETWINDOWCONFIG, see Setting the Graphics Mode.

Routines such as SETWINDOWCONFIG work on the window that is currently in focus. You can have multiple windows open as your application requires, but you need to decide which one gains focus. There is a single frame window and one or more child windows. A window is in focus right after it is opened, after I/O to it, and when FOCUSQQ is used. Clicking the mouse when the cursor is in a window will also bring the window into focus.

For example, to set the characteristics for the window associated with unit 10, either gain focus with either an OPEN, a subsequent READ or WRITE statement to unit 10, or FOCUSQQ. For example, use OPEN:

```
open(unit=10, file='user')
  result = setwindowconfig(wc)
```

After you open unit 10, focus can be regained by a READ or WRITE statement to that unit. For example:

```
write(10,*) "Hello, this is unit 10"
```

Or you can use FOCUSQQ:

```
result = focusqq(10)
  result = setwindowconfig(wc)
```

For more information about when a window gains focus, see Giving a Window Focus and Setting the Active Window.

Creating Child Windows

The FILE='USER' option in the `OPEN` statement opens a child window. The child window defaults to a scrollable text window, 30 rows by 80 columns. You can open up to 40 child windows.

Running a QuickWin application displays the frame window, but not the child window. You must call `SETWINDOWCONFIG` or execute an I/O statement or a graphics statement to display the child window. The window receives output by its unit number, as in:

```
OPEN (UNIT= 12, FILE= 'USER', TITLE= 'Product Matrix')
WRITE (12, *) 'Enter matrix type: '
```

Child windows opened with FILE='USER' must be opened as sequential-access formatted files (the default). Other file specifications (direct-access, binary, or unformatted) result in run-time errors.

The following example creates three child windows. A frame window is automatically created. Text is written to each so the child windows are visible:

```
program testch
use ifqwin
open(11,file="user")
write(11,*) "Hello 11"
open(12,file="user")
write(12,*) "Hello 12"
open(13,file="user")
write(13,*) "Hello 13"
write(13,*) "Windows 11, 12, and 13 can be read and written with normal"
write(13,*) "Fortran I/O statements. The size of each window on the screen"
write(13,*) "can be modified by SETWSIZEQQ. The size of the virtual window"
write(13,*) "(i.e., a data buffer) can be modified by SETWINDOWCONFIG."
read(13,*)
end
```

When this program is run, the output appears as follows:

Giving a Window Focus and Setting the Active Window

When a window is made *active*, it receives graphics output (from ARC, LINETO, and OUTGTEXT, for example) but is not brought to the foreground and thus does not have the *focus*. When a window acquires focus, either by a mouse click, I/O to it, or by a FOCUSQQ call, it also becomes the *active* window. When a window gains focus, the window that previously had focus will lose focus.

If a window needs to be brought to the foreground, it must be given *focus*. The window that has the focus is always on top, and all other windows have their title bars grayed out. A window can have the focus and yet not be active and not have graphics output directed to it. Graphical output is independent of focus.

Under most circumstances, *focus* and *active* should apply to the same window. This is the default behavior of QuickWin and a programmer must consciously override this default.

Certain QuickWin routines (such as GETCHARQQ, PASSDIRKEYSQQ, and SETWINDOWCONFIG) that do not take a unit number as an input argument usually affect the *active* window whether or not it is in *focus*.

If another window is made *active* but is not in *focus*, these routines affect the window *active* at the time of the routine call. This may appear unusual to the user since a GETCHARQQ under these circumstances will expect input from a grayed, background window. The user would then have to click on that window before input could be typed to it.

To use these routines (that effect the *active* window), either do I/O to the unit number of the window you wish to put in *focus* (and also make *active*), or call FOCUSQQ (with a unit number specified). If only one window is open, then that window is the one affected. If several windows are opened, then the last one opened is the one affected since that window will get *focus* and *active* as a side effect of being opened.

The OPEN (IOFOCUS) parameter also can determine whether a window receives the focus when a I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

With an explicit OPEN with FILE='USER', IOFOCUS defaults to .TRUE. For child windows opened implicitly (no OPEN statement before the READ, WRITE, or PRINT) as unit 0, 5, or 6, IOFOCUS defaults to .FALSE..

If IOFOCUS=.TRUE., the child window receives focus prior to each READ, WRITE, or PRINT. Calls to OUTTEXT or graphics functions (for example, OUTGTEXT, LINETO, and ELLIPSE) do not cause the focus to shift. If you use IOFOCUS with any unit other than a QuickWin child window, a run-time error occurs.

The focus shifts to a window when it is given the focus with FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. INQFOCUSQQ determines which unit has the focus. For example:

```
   USE IFQWIN
  INTEGER status, focusunit
  OPEN(UNIT = 10, FILE = 'USER', TITLE = 'Child Window 1')
  OPEN(UNIT = 11, FILE = 'USER', TITLE = 'Child Window 2')
! Give focus to Child Window 2 by writing to it:
  WRITE (11, *) 'Giving focus to Child 2.'
! Give focus to Child Window 1 with the FOCUSQQ function:
  status = FOCUSQQ(10)
   ...
! Find out the unit number of the child window that currently has focus:
  status = INQFOCUSQQ(focusunit)
```

SETACTIVEQQ makes a child window active without bringing it to the foreground. GETACTIVEQQ returns the unit number of the currently active child window. GETHWNDQQ converts the unit number into a Windows handle for functions that require it.

Keeping Child Windows Open

A child window remains open as long as its unit is open. The STATUS parameter in the CLOSE statement determines whether the child window remains open after the unit has been closed. If you set STATUS='KEEP', the associated window remains open but no further input or output is permitted. Also, the Close command is added to the child window's menu and the word Closed is appended to the window title. The default is STATUS='DELETE', which closes the window.

A window that remains open when you use STATUS='KEEP' counts as one of the 40 child windows available for the QuickWin application.

Controlling Size and Position of Windows

SETWSIZEQQ and GETWSIZEQQ set and get the size and position of the visible representation of a window. The positions and dimensions of visible child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels. The position and dimensions are returned in the derived type qwinfo defined in IFQWIN.MOD as follows:

```
TYPE QWINFO
  INTEGER(2) TYPE     ! Type of action performed by SETWSIZEQQ.
  INTEGER(2) X        ! x-coordinate for upper left corner.
  INTEGER(2) Y        ! y-coordinate for upper left corner.
  INTEGER(2) H        ! Window height.
  INTEGER(2) W        ! Window width.
END TYPE QWINFO
```

The options for the qwinfo type are listed under SETWSIZEQQ in the *Language Reference*.

GETWSIZEQQ returns the position and the current or maximum window size of the current frame or child window. To access information about a child window, specify the unit number associated with it. Unit numbers 0, 5, and 6 refer to the default startup window if you have not explicitly opened them with the OPEN statement. To access information about the frame window, specify the unit number as the symbolic constant QWIN$FRAMEWINDOW. For example:

```
  USE IFQWIN
 INTEGER status
 TYPE (QWINFO) winfo
 OPEN (4, FILE='USER')
  ...
! Get current size of child window associated with unit 4.
 status = GETWSIZEQQ(4, QWIN$SIZECURR, winfo)
 WRITE (*,*) "Child window size is ", winfo.H, " by ", winfo.W
! Get maximum size of frame window.
 status = GETWSIZEQQ(QWIN$FRAMEWINDOW, QWIN$SIZEMAX, winfo)
 WRITE (*,*) "Max frame window size is ", winfo.H, " by ", winfo.W
```

SETWSIZEQQ is used to set the visible window position and size. For example:

```
  USE IFQWIN
 INTEGER status
 TYPE (QWINFO) winfo
 OPEN (4, FILE='USER')
 winfo.H    = 30
 winfo.W    = 80
 winfo.TYPE = QWIN$SET
 status     = SETWSIZEQQ(4, winfo)
```

Using QuickWin Graphics Library Routines

Using Graphics Library Routines

Graphics routines are functions and subroutines that draw lines, rectangles, ellipses, and similar elements on the screen. Font routines create text in a variety of sizes and styles. The QuickWin graphics library provides routines that:

- Change the window's dimensions.
- Set coordinates.

- Set color palettes.
- Set line styles, fill masks, and other figure attributes.
- Draw graphics elements.
- Display text in several character styles.
- Display text in fonts compatible with Microsoft Windows.
- Store and retrieve screen images.

Selecting Display Options

The QuickWin run-time library provides a number of routines that you can use to define text and graphics displays. These routines determine the graphics environment characteristics and control the cursor.

To display graphics, you need to set the desired graphics mode using `SETWINDOWCONFIG`, and then call the routines needed to create the graphics. SETWINDOWCONFIG is the routine you use to configure window properties. You can use `DISPLAYCURSOR` to control whether the cursor will be displayed. The cursor becomes invisible after a call to `SETWINDOWCONFIG`. To display the cursor you must explicitly turn on cursor visibility with `DISPLAYCURSOR`($GCURSORON).

`SETGTEXTROTATION` sets the current orientation for font text output, and `GETGTEXTROTATION` returns the current setting. The current orientation is used in calls to `OUTGTEXT`.

For more information on these routines, see the *Language Reference* in the *Intel© Visual Fortran Compiler User and Reference Guides*.

Checking the Current Graphics Mode

Call `GETWINDOWCONFIG` to get the child window settings.

`GETWINDOWCONFIG` uses the derived type, `windowconfig`, as a parameter:

```
TYPE windowconfig
 INTEGER(2) numxpixels      ! Number of pixels on x-axis
 INTEGER(2) numypixels      ! Number of pixels on y-axis
 INTEGER(2) numtextcols     ! Number of text columns available
 INTEGER(2) numtextrows     ! Number of text rows available
 INTEGER(2) numcolors       ! Number of color indexes
 INTEGER(4) fontsize        ! Size of default font
 CHARACTER(80) title        ! window title
 INTEGER(2) bitsperpixel    ! Number of bits per pixel
END TYPE windowconfig
```

By default, a QuickWin child window is a scrollable text window 640x480 pixels, has 30 lines and 80 columns, and a font size of 8x16. Also by default, a Standard Graphics window is Full Screen. You can change the values of window properties at any time with `SETWINDOWCONFIG`, and retrieve the current values at any time with `GETWINDOWCONFIG`.

Setting the Graphics Mode

Use `SETWINDOWCONFIG` to configure the window for the properties you want. To set the highest possible resolution available with your graphics driver, assign a -1 value for *`numxpixels`*, *`numypixels`*, *`numtextcols`*, and *`numtextrows`* in the `windowconfig` derived type. This causes Fortran Standard Graphics applications to start in Full Screen mode.

If you specify less than the largest graphics area, the application starts in a window. You can use ALT+ENTER to toggle between Full Screen and windowed views. If your application is a QuickWin application and you do not call SETWINDOWCONFIG, the child window defaults to a scrollable text window with the dimensions of 640x480 pixels, 30 lines, 80 columns, and a font size of 8x16. The number of colors depends on the video driver used.

If SETWINDOWCONFIG returns .FALSE., the video driver does not support the options specified. The function then adjusts the values in the windowconfig derived type to ones that will work and are as close as possible to the requested configuration. You can then call SETWINDOWCONFIG again with the adjusted values, which will succeed. For example:

```
LOGICAL statusmode
TYPE (windowconfig) wc
wc%numxpixels  = 1000
wc%numypixels  = 300
wc%numtextcols = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%title = "Opening Title"C
wc%fontsize = #000A000C ! 10 X 12
statusmode = SETWINDOWCONFIG(wc)
IF (.NOT. statusmode) THEN statusmode = SETWINDOWCONFIG(wc)
```

If you use SETWINDOWCONFIG, you should specify a value for each field (-1 or your own number for numeric fields, and a C string for the title). Calling SETWINDOWCONFIG with only some of the fields specified can result in useless values for the other fields.

Setting Figure Properties

The output routines that draw arcs, ellipses, and other primitive figures do not specify color or line-style information. Instead, they rely on properties set independently by other routines.

GETCOLORRGB (or GETCOLOR) and SETCOLORRGB (or SETCOLOR) obtain or set the current color value (or color index), which FLOODFILLRGB (or FLOODFILL), OUTGTEXT, and the shape-drawing routines all use. Similarly, GETBKCOLORRGB (or GETBKCOLOR) and SETBKCOLORRGB (or SETBKCOLOR) retrieve or set the current background color.

GETFILLMASK and SETFILLMASK return or set the current fill mask. The mask is an 8-by-8-bit array with each bit representing a pixel. If a bit is 0, the pixel in memory is left untouched: the mask is transparent to that pixel. If a bit is 1, the pixel is assigned the current color value. The array acts as a template that repeats over the entire fill area. It "masks" the background with a pattern of pixels drawn in the current color, creating a large number of fill patterns. These routines are particularly useful for shading.

GETWRITEMODE and SETWRITEMODE return or set the current *logical write mode* used when drawing lines. The logical write mode, which can be set to $GAND, $GOR, $GPRESET, $GPSET, or $GXOR, determines the interaction between the new drawing and the existing screen and current graphics color. The logical write mode affects the LINETO, RECTANGLE, and POLYGON routines.

GETLINESTYLE and SETLINESTYLE retrieve and set the current line style. The line style is determined by a 16-bit-long mask that determines which of the five available styles is chosen. You can use these two routines to create a wide variety of dashed lines that affect the LINETO, RECTANGLE, and POLYGON routines.

See Also

Language Reference for a description of these routines.

Understanding Coordinate Systems

Understanding Coordinate Systems Overview

Several different coordinate systems are supported by the Intel® Fortran QuickWin Library.

Text coordinates use a coordinate system that divides the screen into rows and columns.

Graphics coordinates are specified using one of three coordinate systems:

- Physical coordinates, which are determined by the hardware and the video mode used
- Viewport coordinates, which you can define in the application
- Window coordinates, which you can define to simplify scaling of floating-point data values

Physical coordinates serve as an absolute reference and as a starting place for creating custom window and viewport coordinates. Conversion routines make it simple to convert between different coordinate systems.

Unless you change it, the viewport-coordinate system is identical to the physical-coordinate system. The physical origin (0, 0) is always in the upper-left corner of the *display*.

For QuickWin, *display* means a child window's client area, not the actual monitor screen (unless you go to Full Screen mode). The x-axis extends in the positive direction left to right, while the y-axis extends in the positive direction top to bottom. The default viewport has the dimensions of the selected mode. In a QuickWin application, you can draw outside of the child window's current client area. If you then make the child window bigger, you will see what was previously outside the frame.

You can also use coordinate routines to convert between physical-, viewport-, and window-coordinate systems.

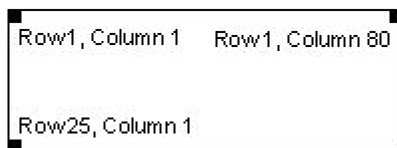
See Also

- Text Coordinates
- Graphics Coordinates

Text Coordinates

The text modes use a coordinate system that divides the screen into rows and columns as shown in the following figure:

Text Screen Coordinates



Text coordinates use the following conventions:

- Numbering starts at 1. An 80-column screen contains columns 1-80.
- The row is always listed before the column.

If the screen displays 25 rows and 80 columns (as shown in the above Figure), the rows are numbered 1-25 and the columns are numbered 1-80. The text-positioning routines, such as `SETTEXTPOSITION` and `SCROLLTEXTWINDOW`, use row and column coordinates.

Graphics Coordinates

Three coordinate systems describe the location of pixels on the screen:

- Physical coordinates
- Viewport coordinates
- Window coordinates

This topic provides information on each of these coordinate systems.

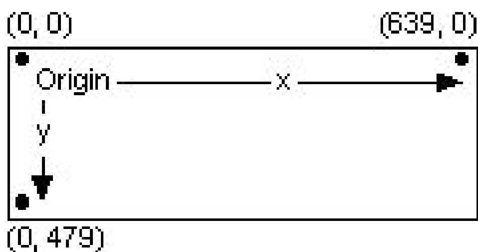
In all three coordinate systems, the x-coordinate is listed before the y-coordinate.

Physical Coordinates

Physical coordinates are integers that refer to pixels in a window's client area. By default, numbering starts at 0, not 1. If there are 640 pixels, they are numbered 0-639.

Suppose your program calls `SETWINDOWCONFIG` to set up a client area containing 640 horizontal pixels and 480 vertical pixels. Each individual pixel is referred to by its location relative to the x-axis and y-axis, as shown in the following figure:

Physical Coordinates



The upper-left corner is the *origin*. The x- and y-coordinates for the origin are always (0, 0).

Physical coordinates refer to each pixel directly and are therefore integers (that is, the window's client area cannot display a fractional pixel). If you use variables to refer to pixel locations, declare them as integers or use type-conversion routines when passing them to graphics functions. For example:

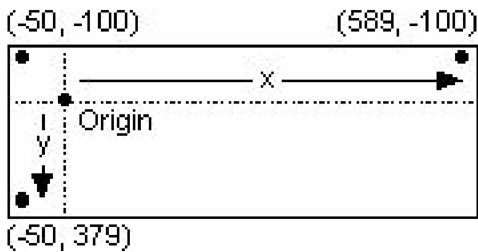
```
ISTATUS = LINETO( INT2(REAL_x), INT2(REAL_y))
```

If a program uses the default dimension of a window, the *viewport* (drawing area) is equal to 640x480. `SETVIEWORG` changes the location of the viewport's origin. You pass it two integers, which represent the x and y physical screen coordinates for the new origin. You also pass it an `xycoord` type that the routine fills with the physical coordinates of the previous origin. For example, the following line moves the viewport origin to the physical screen location (50, 100):

```
TYPE (xycoord) origin
  CALL SETVIEWORG(INT2(50), INT2(100), origin)
```

The effect on the screen is illustrated in the following figure:

Origin Coordinates Changed by SETVIEWORG



The number of pixels hasn't changed, but the coordinates used to refer to the points have changed. The x-axis now ranges from -50 to +589 instead of 0 to 639. The y-axis now covers the values -100 to +379.

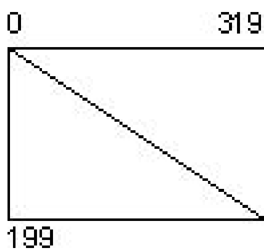
All graphics routines that use viewport coordinates are affected by the new origin, including `MOVETO`, `LINETO`, `RECTANGLE`, `ELLIPSE`, `POLYGON`, `ARC`, and `PIE`. For example, if you call `RECTANGLE` after relocating the viewport origin and pass it the values (0, 0) and (40, 40), the upper-left corner of the rectangle would appear 50 pixels from the left edge of the screen and 100 pixels from the top. It would not appear in the upper-left corner of the screen.

`SETCLIPRGN` creates an invisible rectangular area on the screen called a *clipping region*. You can draw inside the clipping region, but attempts to draw outside the region fail (nothing appears outside the clipping region).

The default clipping region occupies the entire screen. The QuickWin Library ignores any attempts to draw outside the screen.

You can change the clipping region by calling `SETCLIPRGN`. For example, suppose you entered a screen resolution of 320x200 pixels. If you draw a diagonal line from (0, 0) to (319, 199), the upper-left to the lower-right corner, the screen looks like the following figure:

Line Drawn on a Full Screen

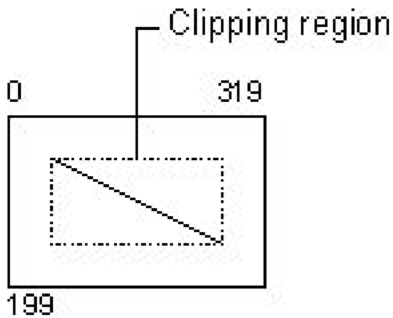


You could create a clipping region by entering:

```
CALL SETCLIPRGN(INT2(10), INT2(10), INT2(309), INT2(189))
```

With the clipping region in effect, the same LINETO command would put the line shown in the following figure on the screen:

Line Drawn Within a Clipping Region



The dashed lines indicate the outer bounds of the clipping region and do not actually print on the screen.

Viewport Coordinates

The viewport is the area of the screen displayed, which may be only a portion of the window's client area. Viewport coordinates represent the pixels within the current viewport. SETVIEWPORT establishes a new viewport within the boundaries of the physical client area. A standard viewport has two distinguishing features:

- The origin of a viewport is in the upper-left corner.
- The default clipping region matches the outer boundaries of the viewport.

SETVIEWPORT has the same effect as SETVIEWORGSETCLIPRGN and combined. It specifies a limited area of the screen in the same manner as SETCLIPRGN, then sets the viewport origin to the upper-left corner of the area.

Window Coordinates

Functions that refer to coordinates on the client-area screen and within the viewport require integer values. However, many applications need floating-point values -- for frequency, viscosity, mass, and so on. SETWINDOW lets you scale the screen to almost any size. In addition, window-related functions accept double-precision values.

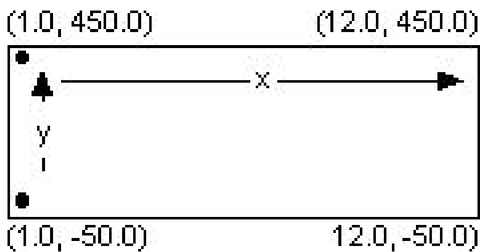
Window coordinates use the current viewport as their boundaries. A window overlays the current viewport. Graphics drawn at window coordinates beyond the boundaries of the window -- the same as being outside the viewport -- are clipped.

For example, to graph 12 months of average temperatures on the planet Venus that range from -50 to +450, add the following line to your program:

```
status = SETWINDOW(.TRUE., 1.0D0, -50.0D0, 12.0D0, 450.0D0)
```

The first argument is the invert flag, which puts the lowest y value in the lower-left corner. The minimum and maximum x- and y-coordinates follow; the decimal point marks them as floating-point values. The new organization of the screen is shown in the following figure:

Window Coordinates



January and December plot on the left and right edges of the screen. In an application like this, numbering the x-axis from 0.0 to 13.0 provides some padding space on the sides and would improve appearance.

If you next plot a point with `SETPIXEL_W LINETO_W` or draw a line with, the values are automatically scaled to the established window.

To use window coordinates with floating-point values:

1. Set a graphics mode with `SETWINDOWCONFIG`.
2. Use `SETVIEWPORT` to create a viewport area. This step is not necessary if you plan to use the entire screen.
3. Create a real-coordinate window with `SETWINDOWDOUBLE PRECISION` invert flag and four `LOGICAL`, passing a x- and y-coordinates for the minimum and maximum values.
4. Draw graphics shapes with `RECTANGLE_W` and similar routines. Do not confuse `RECTANGLE` (the viewport routine) with **RECTANGLE_W** (the window routine for drawing rectangles). All window function names end with an underscore and the letter `W` (_W).

Real-coordinate graphics give you flexibility and device independence. For example, you can fit an axis into a small range (such as 151.25 to 151.45) or into a large range (-50000.0 to +80000.0), depending on the type of data you graph. In addition, by changing the window coordinates, you can create the effects of zooming in or panning across a figure. The window coordinates also make your drawings independent of the computer's hardware. Output to the viewport is independent of the actual screen resolution.

Setting Graphics Coordinates

You can set the pixel dimensions of the x- and y-axes with `SETWINDOWCONFIG`. You can access these values through the *`wc%numxpixels`* and *`wc%numypixels`* values returned by `GETWINDOWCONFIG`. Similarly, `GETWINDOWCONFIG` also returns the range of colors available in the current mode through the *`wc%numcolors`* value.

You can also define the graphics area with `SETCLIPRGN` and `SETVIEWPORT`. Both of these functions define a subset of the available window area for graphics output.`SETCLIPRGN` does not change the viewport coordinates, but merely masks part of the screen. `SETVIEWPORT` resets the viewport bounds to the limits you give it and sets the origin to the upper-left corner of this region.

The origin of the viewport-coordinate system can be moved to a new position relative to the physical origin with `SETVIEWORG`. Regardless of the viewport coordinates, however, you can always locate the current graphics output position with `GETCURRENTPOSITION and GETCURRENTPOSITION_W`.

Using the window-coordinate system, you can easily scale any set of data to fit on the screen. You define any range of coordinates (such as 0 to 5000) that works well for your data as the range for the window-coordinate axes. By telling the program that you want the window-coordinate system to fit in a particular area on the screen (map to a particular set of viewport coordinates), you can scale a chart or drawing to any size you want. SETWINDOW defines a window-coordinate system bounded by the specified values.

GETPHYSCOORD converts viewport coordinates to physical coordinates, and GETVIEWCOORD translates from physical coordinates to viewport coordinates. Similarly, GETVIEWCOORD_W converts window coordinates to viewport coordinates, and GETWINDOWCOORD converts viewport coordinates to window coordinates.

For more information:

- On these routines, see their descriptions in the *Language Reference*.

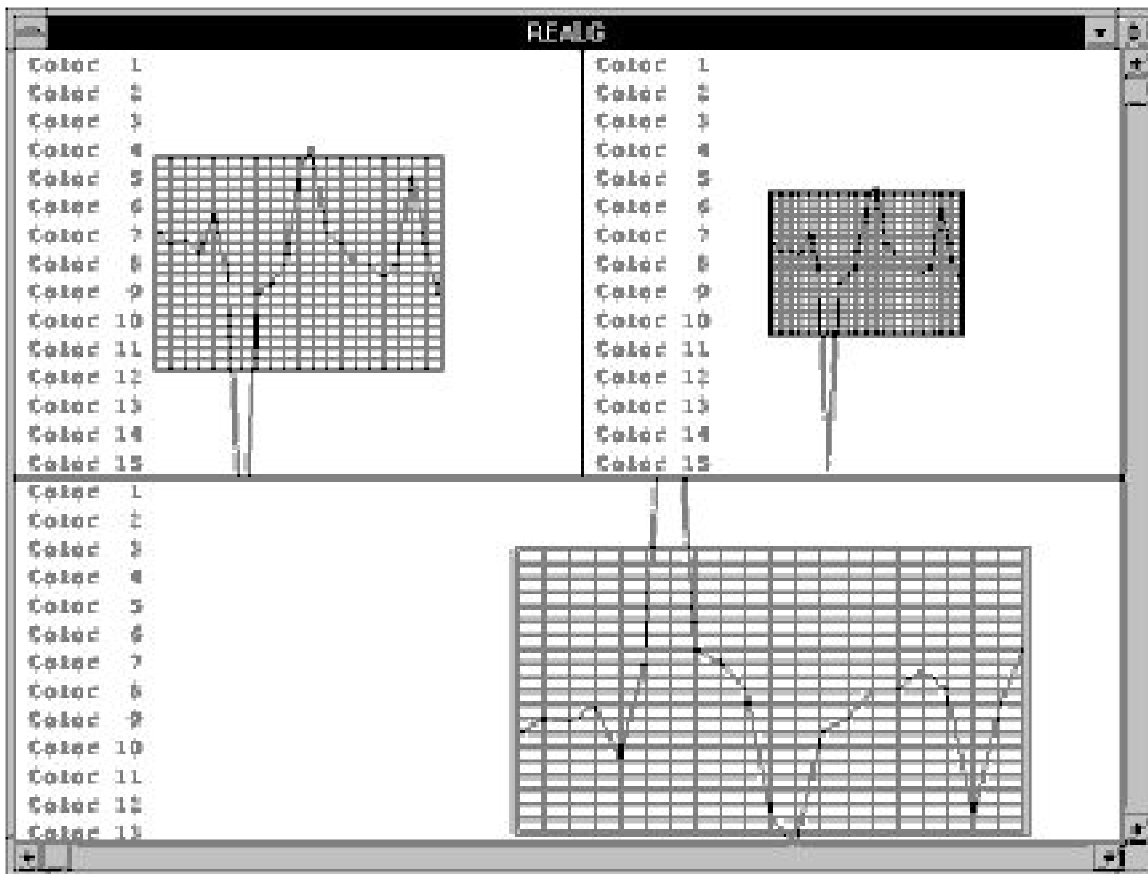
Real Coordinates Sample Program

The program REALG.F90 shows how to create multiple window-coordinate sets, each in a separate viewport, on a single screen.

```
! REALG.F90 (main program) - Illustrates coordinate graphics.
  !
     USE IFQWIN
     LOGICAL             statusmode
     TYPE (windowconfig)  myscreen
     COMMON              myscreen
  !
  !   Set the screen to the best resolution and maximum number of
  !   available colors.
     myscreen.numxpixels  = -1
     myscreen.numypixels  = -1
     myscreen.numtextcols = -1
     myscreen.numtextrows = -1
     myscreen.numcolors   = -1
     myscreen.fontsize    = -1
     myscreen.title       = " "C
     statusmode = SETWINDOWCONFIG(myscreen)
     IF(.NOT. statusmode) statusmode = SETWINDOWCONFIG(myscreen)
     statusmode = GETWINDOWCONFIG( myscreen )
     CALL threegraphs( )
     END
     .
     .
     .
```

The main body of the program is very short. It sets the window to the best resolution of the graphics driver (by setting the first four fields to -1) and the maximum number of colors (by setting `numcolors` to -1). The program then calls the threegraphs subroutine that draws three graphs. The program output is shown in the following figure:

REALG Program Output



The gridshape subroutine, which draws the graphs, uses the same data in each case. However, the program uses three different coordinate windows.

The two viewports in the top half are the same size in physical coordinates, but have different window sizes. Each window uses different maximum and minimum values. In all three cases, the graph area is two units wide. The window in the upper-left corner has a range in the x-axis of four units (4 units wide); the window in the upper-right corner has a range in the x-axis of six units, which makes the graph on the right appear smaller.

In two of the three graphs, one of the lines goes off the edge, outside the clipping region. The lines do not intrude into the other viewports, because defining a viewport creates a clipping region.

Finally, the graph on the bottom inverts the data with respect to the two graphs above it.

The next section describes and discusses the subroutine invoked by REALG.F90:

Drawing the Graphs

The main program calls `threegraphs`, which prints the three graphs:

```
 SUBROUTINE threegraphs()
  USE IFQWIN
  INTEGER(2)           status, halfx, halfy
  INTEGER(2)           xwidth, yheight, cols, rows
  TYPE (windowconfig)  myscreen
  COMMON               myscreen
  CALL CLEARSCREEN( $GCLEARSCREEN )
  xwidth  = myscreen.numxpixels
  yheight = myscreen.numypixels
  cols    = myscreen.numtextcols
  rows    = myscreen.numtextrows
  halfx   = xwidth / 2
  halfy   = (yheight / rows) * ( rows / 2 )
!
!  First window
!
  CALL SETVIEWPORT( INT2(0), INT2(0), halfx - 1, halfy - 1 )
  CALL SETTEXTWINDOW( INT2(1), INT2(1), rows / 2, cols / 2 )
  status = SETWINDOW( .FALSE., -2.0_8, -2.0_8, 2.0_8, 2.0_8 )
!  The 2.0_8 notation makes these constants REAL(8)
  CALL gridshape( rows / 2 )
  status = RECTANGLE( $GBORDER,INT2(0),INT2(0),halfx-1,halfy-1 )
!
!  Second window
!
  CALL SETVIEWPORT( halfx, INT2(0), xwidth - 1, halfy - 1 )
  CALL SETTEXTWINDOW( INT2(1), (cols/2) + 1, rows/2, cols )
  status = SETWINDOW( .FALSE., -3.0D0, -3.0D0, 3.0D0, 3.0D0 )
!  The 3.0D0 notation makes these constants REAL(8)
  CALL gridshape( rows / 2 )
  status = RECTANGLE_W( $GBORDER, -3.0_8,-3.0_8,3.0_8, 3.0_8 )
!
!  Third window
!
  CALL SETVIEWPORT( 0, halfy, xwidth - 1, yheight - 1 )
  CALL SETTEXTWINDOW( (rows / 2 ) + 1, 1_2, rows, cols )
  status = SETWINDOW( .TRUE., -3.0_8, -1.5_8, 1.5_8, 1.5_8 )
  CALL gridshape( INT2( (rows / 2) + MOD( rows, INT2(2) ) ) )
  status = RECTANGLE_W( $GBORDER, -3.0_8, -1.5_8, 1.5_8, 1.5_8 )
 END SUBROUTINE
```

Although the screen is initially clear, `threegraphs` makes sure by calling the `CLEARSCREEN` routine to clear the window:

```
CALL CLEARSCREEN( $GCLEARSCREEN )
```

The $GCLEARSCREEN constant clears the entire window. Other options include $GVIEWPORT and $GWINDOW, which clear the current viewport and the current text window, respectively.

After assigning values to some variables, `threegraphs`creates the first window:

```
CALL SETVIEWPORT( INT2(0), INT2(0), halfx - 1, halfy - 1 )
CALL SETTEXTWINDOW( INT2(1), INT2(1), rows / 2, cols / 2 )
status = SETWINDOW( .FALSE., -2.0_8, -2.0_8, 2.0_8, 2.0_8 )
```

The first instruction defines a viewport that covers the upper-left quarter of the screen. The next instruction defines a text window within the boundaries of that border. Finally, the third instruction creates a window with both x and y values ranging from -2.0 to 2.0. The .FALSE. constant causes the y-axis to increase from top to bottom, which is the default. The _8 notation identifies the constants as REAL(8).

Next, the function `gridshape` inserts the grid and plots the data, and a border is added to the window:

```
CALL gridshape( rows / 2 )
status = RECTANGLE( $GBORDER,INT2(0),INT2(0),halfx-1,halfy-1 )
```

This is the standard `RECTANGLE` routine, which takes coordinates relative to the viewport, not the window.

The `gridshape` subroutine plots the data on the screen.

```
! GRIDSHAPE - This subroutine plots data for REALG.F90
!
 SUBROUTINE gridshape( numc )

  USE IFQWIN
  INTEGER(2)           numc, i, status
  INTEGER(4)           rgbcolor, oldcolor
  CHARACTER(8)         str
  REAL(8)              bananas(21), x
  TYPE (windowconfig)  myscreen
  TYPE (wxycoord)      wxy
  TYPE (rccoord)       curpos
  COMMON               myscreen
 !
 !  Data for the graph:
 !
   DATA bananas / -0.3, -0.2, -0.224, -0.1, -0.5, 0.21, 2.9, &
   & 0.3, 0.2, 0.0, -0.885, -1.1, -0.3, -0.2, &
   & 0.001, 0.005, 0.14, 0.0, -0.9, -0.13, 0.31 /
 !
 !  Print colored words on the screen.
 !
   IF(myscreen.numcolors .LT. numc) numc = myscreen.numcolors-1
   DO i = 1, numc
       CALL SETTEXTPOSITION( i, INT2(2), curpos )
       rgbcolor = 12**i -1
       rgbcolor = MODULO(rgbcolor, #FFFFFF)
       oldcolor = SETTEXTCOLORRGB( rgbcolor )
       WRITE ( str, '(I8)' ) rgbcolor
       CALL OUTTEXT( 'Color ' // str )
   END DO
 !
 !  Draw a double rectangle around the graph.
 !
   oldcolor = SETCOLORRGB( #0000FF ) ! full red
   status = RECTANGLE_W( $GBORDER, -1.00_8, -1.00_8, 1.00_8,1.00_8 )
 ! constants made REAL(8) by appending _8
   status = RECTANGLE_W( $GBORDER, -1.02_8, -1.02_8, 1.02_8, 1.02_8 )
 !
 !  Plot the points.
 !
   x = -0.90
   DO i = 1, 19
       oldcolor = SETCOLORRGB( #00FF00 ) ! full green
       CALL MOVETO_W( x, -1.0_8, wxy )
       status = LINETO_W( x, 1.0_8 )
       CALL MOVETO_W( -1.0_8, x, wxy )
       status = LINETO_W( 1.0_8, x )
```

```
        oldcolor = SETCOLORRGB( #FF0000 ) ! full blue
        CALL MOVETO_W( x - 0.1_8, bananas( i ), wxy )
        status = LINETO_W( x, bananas( i + 1 ) )
        x = x + 0.1
    END DO
    CALL MOVETO_W( 0.9_8, bananas( i ), wxy )
    status = LINETO_W( 1.0_8, bananas( i + 1 ) )
    oldcolor = SETCOLORRGB( #00FFFF ) ! yellow
  END SUBROUTINE
```

The routine names that end with `_W` work in the same way as their viewport equivalents, except that you pass double-precision floating-point values instead of integers. For example, you pass INTEGER(2) to `LINETO`, but REAL(8) values to `LINETO_W`.

The two other windows are similar to the first. All three call the gridshape function, which draws a grid from location (-1.0, -1.0) to (1.0, 1.0). The grid appears in different sizes because the coordinates in the windows vary. The second window ranges from (-3.0, -3.0) to (3.0, 3.0), and the third from (-3.0, -1.5) to (1.5, 1.5), so the sizes change accordingly.

The third window also contains a .TRUE. inversion argument. This causes the y-axis to increase from bottom to top, instead of top to bottom. As a result, this graph appears upside down with respect to the other two.

After calling `gridshape`, the program frames each window, using a statement such as the following:

```
  status = RECTANGLE_W( $GBORDER, -3.0_8, -1.5_8, 1.5_8, 1.5_8 )
```

The first argument is a fill flag indicating whether to fill the rectangle's interior or just to draw its outline. The remaining arguments are the x and y coordinates for the upper-left corner followed by the x and y coordinates for the lower-right corner.`RECTANGLE` takes integer arguments that refer to the viewport coordinates.**RECTANGLE_W** takes four double-precision floating-point values referring to window coordinates.

After you create various graphics elements, you can use the font-oriented routines to polish the appearance of titles, headings, comments, or labels. Using Fonts from the Graphics Library describes in more detail how to print text in various fonts with font routines.

Advanced Graphics Using OpenGL

OpenGL is a library of graphic functions that create sophisticated graphic displays such as 3-D images and animation. OpenGL is commonly available on workstations. Writing to this standard allows your program to be ported easily to other platforms.

OpenGL windows are used independently of and in addition to any console, QuickWin and regular Windows windows your application uses. Every window in OpenGL uses a pixel format, and the pixels carry, among other things, RGB values, opacity values, and depth values so that pixels with a small depth (shallow) overwrite deeper pixels. The basic steps in creating OpenGL applications are:

- Specify the pixel format
- Specify how the pixels will be rendered on the video device
- Call OpenGL commands

OpenGL programming is straightforward, but requires a particular initialization and order, like other software tools. References to get you started are:

- *The OpenGL Reference Manual*, Addison-Wesley, ISBN 0-201-46140-4.
- *The OpenGL Programming Guide*, Addison-Wesley, ISBN 0-201-46138-2.

- *OpenGL SuperBible: The Complete Guide to OpenGL Programming on Windows NT and Windows 95*, Richard Wright and Michael Sweet, Waite Group Press (Division of Sams Publishing), 1996, ISBN 1-57169-073-5.
- OpenGL documentation in the Platform SDK title in HTML Help Viewer.
- The OpenGL description from the Microsoft Visual C++ manuals.

NOTE. Microsoft no longer provides the glAux procedures.

Intel Visual Fortran provides an OpenGL module, IFOPNGL.MOD, invoked with the `USE` statement line:

```
USE IFOPNGL
```

When you use this module, all constants and interfaces that bind Fortran to the OpenGL routines become available. Any link libraries required to link with an OpenGL program are automatically searched if **USE IFOPNGL** is present in your Fortran program.

An OpenGL window can be opened from a console, Windows, or QuickWin application. The OpenGL window uses OpenGL calls exclusively, not normal Graphic Device Interface (GDI) calls. Likewise, OpenGL calls cannot be made within an ordinary Windows window or QuickWin child window, because special initialization is required for OpenGL calls.

The Fortran OpenGL identifiers are the same as the C identifiers (such as using a GL_ prefix for constants), except that the gl prefix is changed to fgl for routines and identifier lengths are limited to 31 characters . The data types in the OpenGL C binding are translated to Fortran types as shown in the following table:

OpenGL/C Type	Fortran Data Type
GLbyte	INTEGER(1)
GLshort	INTEGER(2)
GLint, GLsizei	INTEGER(4)
GLfloat, GLclampf	REAL(4)
GLdouble, GLclampd	REAL(8)
GLubyte	INTEGER(1)
GLboolean	LOGICAL
GLushort	INTEGER(2)
GLuint, GLenum, GLbitfield	INTEGER(4)
GLvoid	not needed
pointers	INTEGER

If you include (`USE`) the parameter constant definitions from `IFOPNGLT.F90` (such as by `USE IFOPNGL`), you can use the constants to specify the kind type, such as `INTEGER(K_GLint)` instead of `INTEGER(4)`.

Adding Color

Adding Color Overview

The Intel® Fortran QuickWin Library supports color graphics. The number of total available colors depends on the current video driver and video adapter you are using. The number of available colors you use depends on the graphics functions you choose.

If you have a VGA machine, you are restricted to displaying at most 256 colors at a time. These 256 colors are held in a palette. You can choose the palette colors from a range of 262,144 colors (256K), but only 256 at a time. The palette routines REMAPPALETTERGB and REMAPALLPALETTERGB assign Red-Green-Blue (RGB) colors to palette indexes.

Functions and subroutines that use color indexes create graphic outputs that depend on the mapping between palette indexes and RGB colors.REMAPPALETTERGB remaps one color index to an RGB color, and REMAPALLPALETTERGB remaps the entire palette, up to 236 colors, (20 colors are reserved by the system). You cannot remap the palette on machines capable of displaying 20 colors or fewer.

SVGA and true color video adapters are capable of displaying 262,144 (256K) colors and 16.7 million colors respectively. If you use a palette, you are restricted to the colors available in the palette.

To access the entire set of available colors, not just the 256 or fewer colors in the palette, you should use functions that specify a color value directly. These functions end in RGB and use Red-Green-Blue color values, not indexes to a palette. For example, SETCOLORRGB, SETTEXTCOLORRGB, and SETPIXELRGB specify a direct color value, while SETCOLOR, SETTEXTCOLOR, and SETPIXEL each specify a palette color index. If you are displaying more than 256 colors simultaneously, you need to use the RGB direct color value functions exclusively.

To set the physical display properties of your monitor, open the Control Panel and click the Display icon.

QuickWin only supports a 256-color palette, regardless of the number of colors set for the monitor.

The different color modes and color functions are discussed and demonstrated in the following sections:

See Also

- Color Mixing
- VGA Color Palette
- Using Text Colors

Color Mixing

If you have a VGA machine, you are restricted to displaying at most 256 colors at a time. These 256 colors are held in a palette. You can choose the palette colors from a range of 262,144 colors (256K), but only 256 at a time. Some display adapters (most SVGAs) are capable of displaying all of the 256K colors and some (true color display adapters) are capable of displaying 256 * 256 * 256 = 16.7 million colors.

If you use a palette, you are restricted to the colors available in the palette. In order to access all colors available on your system, you need to specify an explicit Red-Green-Blue (RGB) value, not a palette index.

When you select a color index, you specify one of the colors in the system's predefined palette. SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR set the current color, background color, and text color to a palette index.

SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB set the colors to a color value chosen from the entire available range. When you select a color value, you specify a level of intensity with a range of 0 - 255 for each of the red, green, and blue color values. The long integer that defines a color value consists of 3 bytes (24 bits) as follows:

```
MSB                    LSB
BBBBBBBB GGGGGGGG RRRRRRRR
```

where R, G, and B represent the bit values for red, green, and blue intensities. To mix a light red (pink), turn red all the way up and mix in some green and blue:

```
10000000 10000000 11111111
```

In hexadecimal notation, this number equals #8080FF. To set the current color to this value, you can use the function:

```
i = SETCOLORRGB (#8080FF)
```

You can also pass decimal values to this function. Keep in mind that 1 (binary 00000001, hex 01) represents a low color intensity and that 255 (binary 11111111, hex FF) equals full color intensity. To create pure yellow (100-percent red plus 100-percent green) use this line:

```
i = SETCOLORRGB( #00FFFF )
```

For white, turn all of the colors on:

```
i = SETCOLORRGB( #FFFFFF)
```

For black, set all of the colors to 0:

```
i = SETCOLORRGB( #000000)
```

RGB values for example colors are in the following table.

RGB Color Values

Color	RGB Value	Color	RGB Value
Black	#000000	Bright White	#FFFFFF
Dull Red	#000080	Bright Red	#0000FF
Dull Green	#008000	Bright Green	#00FF00
Dull Yellow	#008080	Bright Yellow	#00FFFF
Dull Blue	#800000	Bright Blue	#FF0000
Dull Magenta	#800080	Bright Magenta	#FF00FF
Dull Turquoise	#808000	Bright Turquoise	#FFFF00
Dark Gray	#808080	Light Gray	#C0C0C0

If you have a 64K-color machine and you set an RGB color value that is not equal to one of the 64K preset RGB color values, the system approximates the requested RGB color to the closest available RGB value. The same thing happens on a VGA machine when you set an RGB color that is not in the palette. (You can remap your VGA color palette to different RGB values; see VGA Color Palette.)

However, although your graphics are drawn with an approximated color, if you retrieve the color with GETCOLORRGB, GETBKCOLORRGB, or GETTEXTCOLORRGB, the color you specified is returned, not the actual color used. This is because the SETCOLORRGB functions do not execute any graphics, they simply set the color and the approximation is made when the drawing is made (by ELLIPSE or ARC , for example).

GETPIXELRGB and GETPIXELSRGB do return the approximated color actually used, because SETPIXELRGB and SETPIXELSRGB actually set a pixel to a color on the screen and the approximation, if any, is made at the time they are called.

VGA Color Palette

A VGA machine is capable of displaying at most 256 colors at a time. QuickWin provides support for VGA monitors and more advanced monitors that are set at 256 colors. Only a 256-color palette (or less) is supported internally regardless of the current number of colors set for the display (in the Control Panel). The number of colors you select for your VGA palette depends on your application, and is set by setting the *wc%numcolors* variable in the windowconfig derived type to 2, 16, or 256 with SETWINDOWCONFIG.

An RGB color value must be in the palette to be accessible to your VGA graphic displays. You can change the default colors and customize your color palette by using REMAPPALETTERGB to change a palette color index to any RGB color value. The following example remaps the color index 1 (default blue color) to the pure red color value given by the RGB value #0000FF. After this is executed, whatever was displayed as blue will appear as red:

```
USE IFQWIN
INTEGER status
status = REMAPPALETTERGB( 1, #0000FF )  ! Reassign color index 1
                                       ! to RGB red
```

REMAPALLPALETTERGB remaps one or more color indexes simultaneously. Its argument is an array of RGB color values that are mapped into the palette. The first color number in the array becomes the new color associated with color index 0, the second with color index 1, and so on. At most 236 indexes can be mapped, because 20 indexes are reserved for system use.

If you request an RGB color that is not in the palette, the color selected from the palette is the closest approximation to the RGB color requested. If the RGB color was previously placed in the palette with REMAPPALETTERGB or REMAPALLPALETTERGB, then that exact RGB color is available.

Remapping the palette has no effect on 64K-color machines, SVGA, or true-color machines, unless you limit yourself to a palette by using color index functions such as SETCOLOR. On a VGA machine, if you remap all the colors in your palette and display that palette in graphics, you cannot then remap and simultaneously display a second palette.

For instance, in VGA 256-color mode, if you remap all 256 palette colors and display graphics in one child window, then open another child window, remap the palette and display graphics in the second child window, you are attempting to display more than 256 colors at one time. The machine cannot do this, so whichever child window has the focus will appear correct, while the one without the focus will change color.

NOTE. Machines that support more than 256 colors will not be able to do animation by remapping the palette. Windows operating systems create a logical palette that maps to the video hardware palette. On video hardware that supports a palette of 256 colors or less, remapping the palette maps over the current palette and redraws the screen in the new colors.

On large hardware palettes that support more than 256 colors, remapping is done into the unused portion of the palette. It does not map over the current colors nor redraw the screen. So, on machines with large palettes (more than 256 colors), the technique of changing the screen through remapping, called palette animation, cannot be used. See the MSDN* *Platform SDK Manual* for more information.

Symbolic constants (names) for the default color numbers are supplied in the graphics modules. The names are self-descriptive; for example, the color numbers for black, yellow, and red are represented by the symbolic constants $BLACK, $YELLOW, and $RED.

Using Text Colors

`SETTEXTCOLORRGB` (or `SETTEXTCOLOR`) and `SETBKCOLORRGB` (or `SETBKCOLOR`) set the foreground and background colors for text output. All use a single argument specifying the color value (or color index) for text displayed with `OUTTEXT` and `WRITE`. For the color index functions, colors are represented by the range 0-31. Index values in the range of 16-31 access the same colors as those in the range of 0-15.

You can retrieve the current foreground and background color values with `GETTEXTCOLORRGB` and `GETBKCOLORRGB` or the color indexes with `GETTEXTCOLOR` and `GETBKCOLOR`. Use `SETTEXTPOSITION` to move the cursor to a particular row and column. `OUTTEXT` and `WRITE` print the text at the current cursor location.

For more information on these routines, see the appropriate routines in the *Language Reference*.

Writing a Graphics Program

Writing a Graphics Program Overview

Like many programs, graphics programs work well when written in small units. Using discrete routines aids debugging by isolating the functional components of the program. The following example program and its associated subroutines show the steps involved in initializing, drawing, and closing a graphics program.

The SINE program draws a sine wave. Its procedures call many of the common graphics routines. The main program calls five subroutines that carry out the actual graphics commands (also located in the SINE.F90 file):

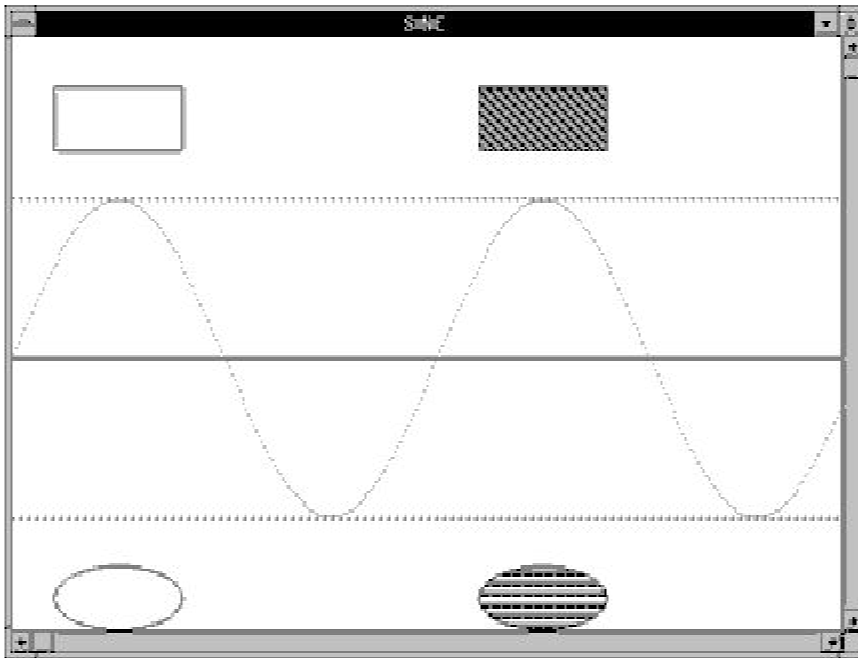
```
! SINE.F90 - Illustrates basic graphics commands.
!   USE IFQWIN
 CALL graphicsmode( )
 CALL drawlines( )
 CALL sinewave( )
 CALL drawshapes( )
 END
  .
  .
  .
```

For information on the subroutines used in the SINE program, see:

- `graphicsmode` in section Activating a Graphics Mode
- `drawlines` in section Drawing Lines on the Screen
- `sinewave` in section Drawing a Sine Curve
- `drawshapes` in section Adding Shapes

The SINE program's output appears in the following figure. The project is built as a Fortran Standard Graphics application.

Sine Program Output



Activating a Graphics Mode

If you call a graphics routine without setting a graphics mode with SETWINDOWCONFIG, QuickWin automatically sets the graphics mode with default values.

The SINE program shown in the previous topic selects and sets the graphics mode in the subroutine graphicsmode, which selects the highest possible resolution for the current video driver:

```
 SUBROUTINE graphicsmode( )
  USE IFQWIN
  LOGICAL             modestatus
  INTEGER(2)          maxx, maxy
  TYPE (windowconfig) myscreen
  COMMON              maxx, maxy
!  Set highest resolution graphics mode.
  myscreen.numxpixels=-1
  myscreen.numypixels=-1
  myscreen.numtextcols=-1
  myscreen.numtextrows=-1
  myscreen.numcolors=-1
  myscreen.fontsize=-1
  myscreen.title = " "C ! blank
  modestatus=SETWINDOWCONFIG(myscreen)
!  Determine the maximum dimensions.
  modestatus=GETWINDOWCONFIG(myscreen)
  maxx=myscreen.numxpixels - 1
  maxy=myscreen.numypixels - 1
 END SUBROUTINE
```

Pixel coordinates start at zero, so, for example, a screen with a resolution of 640 horizontal pixels has a maximum x-coordinate of 639. Thus, maxx (the highest available x-pixel coordinate) must be 1 less than the total number of pixels. The same applies to maxy.

To remain independent of the video mode set by `graphicsmode`, two short functions convert an arbitrary screen size of 1000x1000 pixels to whatever video mode is in effect. From now on, the program assumes it has 1000 pixels in each direction. To draw the points on the screen, `newx` and `newy` map each point to their physical (pixel) coordinates:

```
 ! NEWX - This function finds new x-coordinates.
  INTEGER(2) FUNCTION newx( xcoord )
  INTEGER(2) xcoord, maxx, maxy
  REAL(4) tempx
  COMMON maxx, maxy
  tempx = maxx / 1000.0
  tempx = xcoord * tempx + 0.5
  newx = tempx
  END FUNCTION
 ! NEWY - This function finds new y-coordinates.
 !
  INTEGER(2) FUNCTION newy( ycoord )
  INTEGER(2) ycoord, maxx, maxy
  REAL(4) tempy
  COMMON maxx, maxy
  tempy = maxy / 1000.0
  tempy = ycoord * tempy + 0.5
  newy = tempy
  END FUNCTION
```

You can set up a similar independent coordinate system with *`window coordinates`*, described in Understanding Coordinate Systems.

Drawing Lines on the Screen

SINE next calls the subroutine `drawlines`, which draws a rectangle around the outer edges of the screen and three horizontal lines that divide the screen into quarters. (See Sine Program Output.)

```
!  DRAWLINES - This subroutine draws a box and
!  several lines.
SUBROUTINE drawlines( )   USE IFQWIN   EXTERNAL        newx, newy
  INTEGER(2)      status, newx, newy, maxx, maxy
  TYPE (xycoord)  xy
  COMMON          maxx, maxy
!
!  Draw the box.
  status = RECTANGLE( $GBORDER, INT2(0), INT2(0), maxx, maxy )
  CALL SETVIEWORG( INT2(0), newy( INT2( 500 ) ), xy )
!  This sets the new origin to 0 for x and 500 for y.
!  Draw the lines.
  CALL MOVETO( INT2(0), INT2(0), xy )
  status = LINETO( newx( INT2( 1000 )), INT2(0))
  CALL SETLINESTYLE( INT2( #AA3C ))
  CALL MOVETO( INT2(0), newy( INT2( -250 )), xy )
  status = LINETO(newx( INT2( 1000 )),newy( INT2( -250 )))
  CALL SETLINESTYLE( INT2( #8888 ))
  CALL MOVETO(INT2(0), newy( INT2( 250 )), xy )
  status = LINETO( newx( INT2( 1000 )),newy( INT2( 250 ) ) )
END SUBROUTINE
```

The first argument to RECTANGLE is the *fill flag*, which can be either $GBORDER or $GFILLINTERIOR. Choose $GBORDER if you want a rectangle of four lines (a border only, in the current line style), or $GFILLINTERIOR if you want a solid rectangle (filled in with the current color and fill pattern). Choosing the color and fill pattern is discussed in Adding Color and Adding Shapes.

The second and third RECTANGLE arguments are the x- and y-coordinates of the upper-left corner of the rectangle. The fourth and fifth arguments are the coordinates for the lower-right corner. Because the coordinates for the two corners are (0, 0) and (maxx, maxy), the call to RECTANGLE frames the entire screen.

The program calls SETVIEWORG to change the location of the viewport origin. By resetting the origin to (0, 500) in a 1000x1000 viewport, you effectively make the viewport run from (0, -500) at the top left of the screen to (1000, 500) at the bottom right of the screen:

```
CALL SETVIEWORG( INT2(0), newy( INT2( 500 ) ), xy )
```

Changing the coordinates illustrates the ability to alter the viewport coordinates to whatever dimensions you prefer. (Viewports and the SETVIEWORG routine are explained in more detail in Understanding Coordinate Systems.)

The call to SETLINESTYLE changes the line style from a solid line to a dashed line. A series of 16 bits tells the routine which pattern to follow. Five possible line patterns are available. For more information, see SETLINESTYLE in the Intel Fortran Language Reference.

When drawing lines, first set an appropriate line style. Then, move to where you want the line to begin and call LINETO, passing to it the point where you want the line to end. The drawlines subroutine uses the following code:

```
CALL SETLINESTYLE(INT2( #AA3C ) )
CALL MOVETO( INT2(0), newy( INT2( -250 ) ), xy )
dummy = LINETO( newx( INT2( 1000 )), newy( INT2( -250 )))
```

MOVETO positions an imaginary pixel cursor at a point on the screen (nothing appears on the screen), and LINETO draws a line. When the program called SETVIEWORG, it changed the viewport origin, and the initial y-axis range of 0 to 1000 now corresponds to a range of -500 to +500. Therefore, the negative value -250 is used as the y-coordinate of LINETO to draw a horizontal line across the center of the top half of the screen, and the value of 250 is used as the y-coordinate to draw a horizontal line across the center of the bottom half of the screen.

Drawing a Sine Curve

With the axes and frame in place, SINE is ready to draw the sine curve. The sinewave routine calculates the x and y positions for two cycles and plots them on the screen:

```
 ! SINEWAVE - This subroutine calculates and plots a sine wave.
SUBROUTINE sinewave( )
     USE IFQWIN
   INTEGER(2)    dummy, newx, newy, locx, locy, i
   INTEGER(4)    color
   REAL          rad
   EXTERNAL      newx, newy
   PARAMETER    ( PI = 3.14159 )
 !
 !  Calculate each position and display it on the screen.
   color = #0000FF ! red
 !
   DO i = 0, 999, 3
      rad   = -SIN( PI * i / 250.0 )
      locx  = newx( i )
      locy  = newy( INT2( rad * 250.0 ) )
      dummy = SETPIXELRGB( locx, locy, color )
   END DO
 END SUBROUTINE
```

SETPIXELRGB takes the two location parameters, locx and locy, and sets the pixel at that position with the specified color value (red).

Adding Shapes

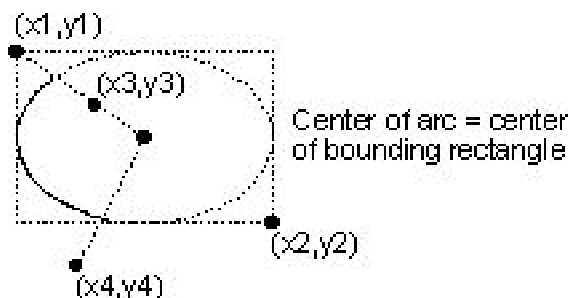
After drawing the sine curve, SINE calls drawshapesto put two rectangles and two ellipses on the screen. The fill flag alternates between $GBORDER and $GFILLINTERIOR:

```
 ! DRAWSHAPES - Draws two boxes and two ellipses.
 !
  SUBROUTINE drawshapes( )
     USE IFQWIN
     EXTERNAL    newx, newy
     INTEGER(2)  dummy, newx, newy
  !
  !  Create a masking (fill) pattern.
  !
     INTEGER(1) diagmask(8), horzmask(8)
     DATA diagmask / #93, #C9, #64, #B2, #59, #2C, #96, #4B /
     DATA horzmask / #FF, #00, #7F, #FE, #00, #00, #00, #CC /
  !
  !  Draw the rectangles.
  !
     CALL SETLINESTYLE( INT2(#FFFF ))
     CALL SETFILLMASK( diagmask )
     dummy = RECTANGLE( $GBORDER,newx(INT2(50)),newy(INT2(-325)), &
     & newx(INT2(200)),newy(INT2(-425)))
     dummy = RECTANGLE( $GFILLINTERIOR,newx(INT2(550)), &
     & newy(INT2(-325)),newx(INT2(700)),newy(INT2(-425)))
  !
  !  Draw the ellipses.
  !
     CALL SETFILLMASK( horzmask )
     dummy = ELLIPSE( $GBORDER,newx(INT2(50)),newy(INT2(325)), &
     & newx(INT2(200)),newy(INT2(425)))
     dummy = ELLIPSE( $GFILLINTERIOR,newx(INT2(550)), &
     & znewy(INT2(325)),newx(INT2(700)),newy(INT2(425)))
   END SUBROUTINE
```

The call to SETLINESTYLE resets the line pattern to a solid line. Omitting this routine causes the first rectangle to appear with a dashed border, because the drawlines subroutine called earlier changed the line style to a dashed line.

ELLIPSE draws an ellipse using parameters similar to those for RECTANGLE. It, too, requires a fill flag and two corners of a bounding rectangle. The following figure shows how an ellipse uses a bounding rectangle:

Bounding Rectangle



The $GFILLINTERIOR constant fills the shape with the current fill pattern. To create a pattern, pass the address of an 8-byte array to SETFILLMASK. In drawshapes, the diagmaskarray is initialized with the pattern shown in the following table:

Fill Patterns

```
Bit pattern                  Value in diagmask
Bit No. 7 6 5 4 3 2 1 0
      x o o x o o x x       diagmask(1) = #93
      x x o o x o o x       diagmask(2) = #C9
      o x x o o x o o       diagmask(3) = #64
      x o x x o o x o       diagmask(4) = #B2
      o x o x x o o x       diagmask(5) = #59
      o o x o x x o o       diagmask(6) = #2C
      x o o x o x x o       diagmask(7) = #96
      o x o o x o x x       diagmask(8) = #4B
```

Displaying Graphics Output

Displaying Graphics Output Overview

The run-time graphics library routines can draw geometric features, display text, display font-based characters, and transfer images between memory and the screen.

The graphics routines provided with Intel® Visual Fortran set points, draw lines, draw text, change colors, and draw shapes such as circles, rectangles, and arcs.

This section uses the following terms:

- The *origin* (point 0, 0) is the upper-left corner of the screen or the client area (defined user area) of the child window being written to. The x-axis and y-axis start at the origin. You can change the origin in some coordinate systems.
- The horizontal direction is represented by the *x-axis*, increasing to the right.
- The vertical direction is represented by the *y-axis*, increasing down.
- Some graphics adapters offer a *color palette* that can be changed.
- Some graphics adapters (VGA and SVGA) allow you to change the color that a color index refers to by providing a *color value* that describes a new color. The color value indicates the mix of red, green, and blue in a screen color. A color value is always an INTEGER number.

See Also

- Drawing Graphics
- Displaying Character-Based Text
- Displaying Font-Based Characters

Drawing Graphics

If you want anything other than the default line style (solid), mask (no mask), background color (black), or foreground color (white), you must call the appropriate routine before calling the drawing routine. Subsequent output routines employ the same attributes until you change them or open a new child window.

The following is a list of routines that provide information on the current graphics settings, set new graphics settings, and draw graphics:

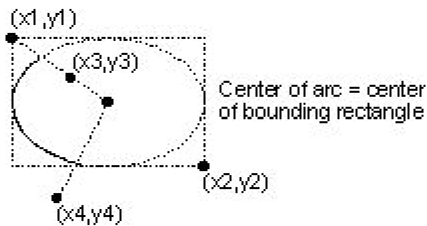
Routine	Description
ARC, ARC_W	Draws an arc
CLEARSCREEN	Clears the screen, viewport, or text window
ELLIPSE, ELLIPSE_W	Draws an ellipse or circle
FLOODFILL, FLOODFILL_W	Fills an enclosed area of the screen with the current color index using the current fill mask
FLOODFILLRGB, FLOODFILLRGB_W	Fills an enclosed area of the screen with the current RGB color using the current fill mask
GETARCINFO	Determines the endpoints of the most recently drawn arc or pie
GETCURRENTPOSITION, GETCURRENTPOSITION_W	Returns the coordinates of the current graphics-output position
GETPIXEL, GETPIXEL_W	Returns a pixel's color index
GETPIXELRGB, GETPIXELRGB_W	Returns a pixel's Red-Green-Blue color value
GETPIXELS	Gets the color indices of multiple pixels
GETPIXELSRGB	Gets the Red-Green-Blue color values of multiple pixels
GRSTATUS	Returns the status (success or failure) of the most recently called graphics routine
INTEGERTORGB	Convert a true color value into its red, green, and blue components
LINETO, LINETO_W	Draws a line from the current graphics-output position to a specified point
LINETOAR, LINETOAREX	Draws lines from arrays at x,y coordinate points
MOVETO, MOVETO_W	Moves the current graphics-output position to a specified point
PIE, PIE_W	Draws a pie-slice-shaped figure
POLYGON, POLYGON_W	Draws a polygon
RECTANGLE, RECTANGLE_W	Draws a rectangle
RGBTOINTEGER	Convert a trio of red, green, and blue values to a true color value for use with RGB functions and subroutines
SETPIXEL, SETPIXEL_W	Sets a pixel at a specified location to a color index
SETPIXELRGB, SETPIXELRGB_W	Sets a pixel at a specified location to a Red-Green-Blue color value

Routine	Description
SETPIXELS	Set the color indices of multiple pixels
SETPIXELSRGB	Set the Red-Green-Blue color value of multiple pixels

Most of these routines have multiple forms. Routine names that end with _W use the window-coordinate system and REAL(8) argument values. Routines without this suffix use the viewport-coordinate system and INTEGER(2) argument values.

Curved figures, such as arcs and ellipses, are centered within a *bounding rectangle*, which is specified by the upper-left and lower-right corners of the rectangle. The center of the rectangle becomes the center for the figure, and the rectangle's borders determine the size of the figure. In the following figure, the points (*x1*, *y1*) and (*x2*, *y2*) define the bounding rectangle.

Bounding Rectangle



Displaying Character-Based Text

The routines in the following table ask about screen attributes that affect text display, prepare the screen for text and send text to the screen. To print text in specialized fonts, see Displaying Font-Based Characters and Using Fonts from the Graphics Library.

In addition to these general text routines, you can customize the text in your menus with MODIFYMENUSTRINGQQ. You can also customize any other string that QuickWin produces, including status bar messages, the state message (for example, "Paused" or "Running"), and dialog box messages, with SETMESSAGEQQ. Use of these customization routines is described in Customizing QuickWin Applications.

The following routines recognize text-window boundaries:

Routine	Description
CLEARSCREEN	Clears the screen, viewport, or text window
DISPLAYCURSOR	Sets the cursor on or off
GETBKCOLOR	Returns the current background color index
GETBKCOLORRGB	Returns the current background Red-Green-Blue color value
GETTEXTCOLOR	Returns the current text color index
GETTEXTCOLORRGB	Returns the current text Red-Green-Blue color value
GETTEXTPOSITION	Returns the current text-output position

Routine	Description
GETTEXTWINDOW	Returns the boundaries of the current text window
OUTTEXT	Sends text to the screen at the current position
SCROLLTEXTWINDOW	Scrolls the contents of a text window
SETBKCOLOR	Sets the current background color index
SETBKCOLORRGB	Sets the current background Red-Green-Blue color value
SETTEXTCOLOR	Sets the current text color to a new color index
SETTEXTCOLORRGB	Sets the current text color to a new Red-Green-Blue color value
SETTEXTPOSITION	Changes the current text position
SETTEXTWINDOW	Sets the current text-display window
WRAPON	Turns line wrapping on or off

These routines do not provide text-formatting capabilities. If you want to print integer or floating-point values, you must convert the values into a string (using an internal WRITE statement) before calling these routines. The text routines specify all screen positions in character-row and column coordinates.

SETTEXTWINDOW is the text equivalent of the SETVIEWPORT graphics routine, except that it restricts only the display area for text printed with OUTTEXT, PRINT, and WRITE.GETTEXTWINDOW returns the boundaries of the current text window set by SETTEXTWINDOW. SCROLLTEXTWINDOW scrolls the contents of a text window.OUTTEXT, PRINT, and WRITE display text strings written to the current text window.

NOTE. The WRITE statement sends its carriage return (CR) and line feed (LF) to the screen at the beginning of the first I/O statement following the WRITE statement. This can cause unpredictable text positioning if you mix the graphics routines SETTEXTPOSITION and OUTTEXT with the WRITE statement. To minimize this effect, use the backslash (\ ) or dollar sign ($) format descriptor (to suppress CR-LF) in the associated FORMAT statement.

See more information on routines mentioned here, see the routine descriptions in the *Language Reference*.

Displaying Font-Based Characters

Because the Intel Visual Fortran Graphics Library provides a variety of fonts, you must indicate which font to use when displaying font-based characters. After you select a font, you can make inquiries about the width of a string printed in that font or about font characteristics.

The following functions control the display of font-based characters:

Routine	Description
GETFONTINFO	Returns the current font characteristics
GETGTEXTEXTENT	Determines the width of specified text in the current font

Routine	Description
GETGTEXTROTATION	Gets the current orientation for font text output in 0.1º increments
INITIALIZEFONTS	Initializes the font library
OUTGTEXT	Sends text in the current font to the screen at the current graphics output position
SETFONT	Finds a single font that matches a specified set of characteristics and makes it the current font used by OUTGTEXT
SETGTEXTROTATION	Sets the current orientation for font text output in 0.1º increments

Characters may be drawn ("mapped") in one of two ways: as bitmapped letters (a "picture" of the letter) or as TrueType characters. See Using Fonts from the Graphics Library, for detailed explanations and examples of how to use the font routines from the QuickWin Library.

For more information on these routines, see the *Language Reference*.

Using Fonts from the Graphics Library

Using Fonts from the Graphics Library Overview

The Intel® Fortran Graphics Library includes routines that print text in various sizes and type styles. These routines provide control over the appearance of your text and add visual interest to your screen displays.

This section assumes you have read Displaying Graphics Output and that you understand the general terminology it introduces. You should also be familiar with the basic properties of both the `SETWINDOWCONFIG` and `MOVETO` routines. Also, remember that graphics programs containing graphics routines must be built as Fortran QuickWin or Fortran Standard Graphics applications.

The project type is set in the visual development environment when you select New from the File menu, then click on the Projects tab, and select Fortran QuickWin or Fortran Standard Graphics Application from the application types listed. Graphics applications can also be built with the `/libs:qwin` or `/libs:qwins` compiler option.

See Also

- Available Typefaces
- Initializing Fonts
- Setting the Font and Displaying Text
- SHOWFONT.F90 Example

Available Typefaces

A *font*is a set of text characters of a particular size and style.

A *typeface* (or *type style*) refers to the style of the displayed characters -- Arial, for example, or Times New Roman.

Type size measures the screen area occupied by individual characters. The term comes from the printer's lexicon, but uses screen pixels as the unit of measure rather than the traditional points. For example, "Courier 12 9" denotes the Courier typeface, with each character occupying a screen area of 12 vertical pixels by 9 horizontal pixels. The word "font", therefore implies both a typeface and a type size.

The QuickWin Library font routines use all Windows operating system installed fonts. The first type of font used is a *bitmap* (or *raster-map*) font. Bitmap fonts have each character in a binary data map. Each bit in the map corresponds to a screen pixel. If the bit equals 1, its associated pixel is set to the current screen color. Bit values of 0 appear in the current background color.

The second type of font is called a TrueType font. Some screen fonts look different on a printer, but TrueType fonts print exactly as they appear on the screen. TrueType fonts may be bitmaps or soft fonts (fonts that are downloaded to your printer before printing), depending on the capabilities of your printer. TrueType fonts are scalable and can be sized to any height. It is recommended that you use TrueType fonts in your graphics programs.

Each type of font has advantages and disadvantages. Bitmapped characters appear smoother on the screen because of the predetermined pixel mapping. However, they cannot be scaled. You can scale TrueType text to any size, but the characters sometimes do not look quite as solid as the bitmapped characters on the screen. Usually this screen effect is hardly noticeable, and when printed, TrueType fonts are as smooth or smoother than bitmapped fonts.

The bitmapped typefaces come in preset sizes measured in pixels. The exact size of any font depends on screen resolution and display type.

Initializing Fonts

A program that uses fonts must first organize the fonts into a list in memory, a process called initializing. The list gives the computer information about the available fonts.

Initialize the fonts by calling the `INITIALIZEFONTS` routine:

```
 USE IFQWIN
INTEGER(2) numfonts
numfonts = INITIALIZEFONTS( )
```

If the computer successfully initializes one or more fonts, INITIALIZEFONTS returns the number of fonts initialized. If the function fails, it returns a negative error code.

Setting the Font and Displaying Text

Before a program can display text in a particular font, it must know which of the initialized fonts to use. `SETFONT` makes one of the initialized fonts the current (or "active") font. SETFONT has the following syntax:

```
result =SETFONT (options)
```

The function's argument consists of letter codes that describe the desired font: typeface, character height and width in pixels, fixed or proportional, and attributes such as bold or italic. These options are discussed in detail in the SETFONT entry in the *Language Reference* in the *Intel® Visual Fortran Compiler User and Reference Guides*. For example:

```
USE IFQWIN
INTEGER(2) index, numfonts
numfonts = INITIALIZEFONTS ( )
index = SETFONT('t''Cottage''h18w10')
```

This sets the typeface to Cottage, the character height to 18 pixels and the width to 10 pixels.

The following example sets the typeface to Arial, the character height to 14, with proportional spacing and italics (the pi codes):

```
index = SETFONT('t''Arial''h14pi')
```

If `SETFONT` successfully sets the font, it returns the font's index number. If the function fails, it returns a negative integer. Call `GRSTATUS` to find the source of the problem; its return value indicates why the function failed. If you call `SETFONT` before initializing fonts, a run-time error occurs.

`SETFONT` updates the font information when it is used to select a font. `GETFONTINFO` can be used to obtain information about the currently selected font.`SETFONT` sets the user fields in the `fontinfo` type (a derived type defined in DFLIB.MOD), and `GETFONTINFO` returns the user-selected values. The following user fields are contained in `fontinfo`:

```
TYPE fontinfo
 INTEGER(2) type ! 1 = truetype, 0 = bit map
 INTEGER(2) ascent ! Pixel distance from top to baseline
 INTEGER(2) pixwidth ! Character width in pixels, 0=prop
```

```
   INTEGER(2) pixheight ! Character height in pixels
   INTEGER(2) avgwidth ! Average character width in pixels
   CHARACTER(32) facename ! Font name
  END TYPE fontinfo
```

To find the parameters of the current font, call GETFONTINFO. For example:

```
   USE IFQWIN
  TYPE (fontinfo) font
  INTEGER(2) i, numfonts
  numfonts = INITIALIZEFONTS()
  i = SETFONT ( ' t ' 'Arial ' )
  i = GETFONTINFO(font)
  WRITE (*,*) font.avgwidth, font.pixheight, font.pixwidth
```

After you initialize the fonts and make one the active font, you can display the text on the screen.

To display text on the screen after selecting a font:

1. Select a starting position for the text with MOVETO.
2. Optionally, set a text display angle with SETGTEXTROTATION.
3. Send the text to the screen (in the current font) with OUTGTEXT.

MOVETO moves the current graphics point to the pixel coordinates passed to it when it is invoked. This becomes the starting position of the upper-left corner of the first character in the text. SETGTEXTROTATION can set the text's orientation in increments of one-tenth of a degree.

SHOWFONT.F90 Example

The program SHOWFONT.F90 displays text in the fonts available on your system. (Once the screen fills with text, press Enter to display the next screen.) An abbreviated version follows. SHOWFONT calls SETFONT to specify the typeface. MOVETO then establishes the starting point for each text string. The program sends a message of sample text to the screen for each font initialized:

```
! Abbreviated version of SHOWFONT.F90.
   USE IFQWIN
   INTEGER(2) grstat, numfonts,indx, curr_height
   TYPE (xycoord) xyt
   TYPE (fontinfo) f
   CHARACTER(6) str     ! 5 chars for font num
                        ! (max. is 32767), 1 for 'n'
! Initialization.
   numfonts=INITIALIZEFONTS( )
   IF (numfonts.LE.0) PRINT *,"INITIALIZEFONTS error"
   IF (GRSTATUS().NE.$GROK) PRINT *,'INITIALIZEFONTS GRSTATUS error.'
   CALL MOVETO (0,0,xyt)
   grstat=SETCOLORRGB(#FF0000)
   grstat=FLOODFILLRGB(0, 0, #00FF00)
   grstat=SETCOLORRGB(0)
! Get default font height for comparison later.
   grstat = SETFONT('n1')
   grstat = GETFONTINFO(f)
   curr_height = f.pixheight
! Done initializing, start displaying fonts.
   DO indx=1,numfonts
      WRITE(str,10)indx
      grstat=SETFONT(str)
      IF (grstat.LT.1) THEN
         CALL OUTGTEXT('SetFont error.')
      ELSE
         grstat=GETFONTINFO(f)
         grstat=SETFONT('n1')
```

```
         CALL OUTGTEXT(f.facename(:len_trim(f.facename)))
         CALL OUTGTEXT(' ')
! Display font.
         grstat=SETFONT(str)
         CALL OUTGTEXT('ABCDEFGabcdefg12345!@#$%')
      END IF
! Go to next line.
      IF (f.pixheight .GT. curr_height) curr_height=f.pixheight
      CALL GETCURRENTPOSITION(xyt)
      CALL MOVETO(0,INT2(xyt.ycoord+curr_height),xyt)
   END DO
10  FORMAT ('n',I5.5)
   END
```

Storing and Retrieving Images

Working With Screen Images

The routines described in the following topics offer the following ways to store and retrieve images:

- Transfer images between memory buffers and the screen

  Transferring images from buffers is a quick and flexible way to move things around the screen. Memory images can interact with the current screen image; for example, you can perform a logical AND of a memory image and the current screen or superimpose a negative of the memory image on the screen.

- Transfer images between the screen and Windows bitmap files

  Transferring images from files gives access to images created by other programs, and saves graphs and images for later use. However, images loaded from bitmap files overwrite the portion of the screen they are pasted into and retain the attributes they were created with, such as the color palette, rather than accepting current attributes.

- Transfer images between the screen and the Clipboard from the QuickWin Edit menu

  Editing screen images from the QuickWin **Edit** menu is a quick and easy way to move and modify images interactively on the screen, retaining the current screen attributes, and also provides temporary storage (the Clipboard) for transferring images among applications.

These routines allow you to cut, paste, and move images around the screen.

Transferring Images in Memory

The `GETIMAGE` and `PUTIMAGE` routines transfer images between memory and the screen and give you options that control the way the image and screen interact. When you hold an image in memory, the application allocates a memory buffer for the image. The `IMAGESIZE` routines calculate the size of the buffer needed to store a given image.

Routines that end with _W use window coordinates; the other functions use viewport coordinates.

Routine	Description
`GETIMAGE, GETIMAGE_W`	Stores a screen image in memory
`IMAGESIZE, IMAGESIZE_W`	Returns image size in bytes
`PUTIMAGE, PUTIMAGE_W`	Retrieves an image from memory and displays it

Loading and Saving Images to Files

The LOADIMAGE and SAVEIMAGE routines transfer images between the screen and Windows bitmap files:

Routine	Description
LOADIMAGE, LOADIMAGE_W	Reads a Windows bitmap file (.BMP) from disk and displays it as specified coordinates
SAVEIMAGE, SAVEIMAGE_W	Captures a screen image from the specified portion of the screen and saves it as a Windows bitmap file

You can use a Windows format bitmap file created with a graphics program as a backdrop for graphics that you draw with the Intel Visual Fortran graphics functions and subroutines.

Editing Text and Graphics from the QuickWin Edit Menu

From the QuickWin Edit menu you can choose the Select Text, Select Graphics, or Select All options. You can then outline your selection with the mouse or the keyboard arrow keys. When you use the Select Text option, your selection is highlighted. When you use the Select Graphics or Select All option, your selection is marked with a box whose dimensions you control.

Once you have selected a portion of the screen, you can copy it onto the Clipboard by using the Edit/Copy option or by using the Ctrl+INS key combination. If the screen area you have selected contains only text, it is copied onto the Clipboard as text. If the selected screen area contains graphics, or a mix of text and graphics, it is copied onto the Clipboard as a bitmap.

The Edit menu's Paste option will only paste text. Bitmaps can be pasted into other Windows applications from the Clipboard (with the Ctrl+V or Shift+INS key combinations).

Remember the following when selecting portions of the screen:

- If you have chosen the Select All option from the Edit menu, the whole screen is selected and you cannot then select a portion of the screen.
- Text selections are not bounded by the current text window set with SETTEXTWINDOW.
- When text is copied to the Clipboard, trailing blanks in a line are removed.
- Text that is written to a window can be overdrawn by graphics. In this case, the text is still present in the screen text buffer, though not visible on the screen. When you select a portion of the screen to copy, you can select text that is actually present but not visible, and that text will be copied onto the Clipboard.
- When you choose Select Text or Select Graphics from the Edit menu, the application is paused, a caret (^) appears at the top left corner of the currently active window, all user-defined callbacks are disabled, and the window title changes to "Mark Text - *windownam* "or "Mark Graphics - *windowname*", where *windowname* is the name of the currently active window.

  As soon as you begin selection (by pressing an arrow key or a mouse button), the Window title changes to "Select Text - *windowname*" or "Select Graphics - *windowname*" and selection begins at that point. If you do not want selection to begin in the upper-left corner, your first action when "Mark Text" or "Mark Graphics" appears in the title is to use the mouse to place the cursor at the position where selection is to be begin.

Customizing QuickWin Applications

Customizing QuickWin Applications Overview

The QuickWin library is a set of routines you can use to create graphics programs or simple applications for Windows. For a general overview of QuickWin and a description of how to create and size child windows, see the beginning of this section. For information on how to compile and link QuickWin applications, see the *Intel® Visual Fortran Compiler User and Reference Guides*.

The following topics describe how to customize and fine-tune your QuickWin applications:

- Enhancing QuickWin Applications
- Controlling Menus
- Changing Status Bar and State Messages
- Displaying Message Boxes
- Defining an About Box
- Using Custom Icons
- Using a Mouse

Enhancing QuickWin Applications

In addition to the basic QuickWin features, you can optionally customize and enhance your QuickWin applications with the features described in the following table. These features let you create customized menus, respond to mouse events, and add custom icons.

Category	QuickWin Function	Description
Initial settings	`INITIALSETTINGS`	Controls initial menu settings and/or initial frame window
Display/add box	`MESSAGEBOXQQ`	Displays a message box
	`ABOUTBOXQQ`	Adds an About Box with customized text
Menu items	`CLICKMENUQQ`	Simulates the effect of clicking or selecting a menu item
	`APPENDMENUQQ`	Appends a menu item
	`DELETEMENUQQ`	Deletes a menu item
	`INSERTMENUQQ`	Inserts a menu item
	`MODIFYMENUFLAGSQQ`	Modifies a menu item's state
	`MODIFYMENUROUTINEQQ`	Modifies a menu item's callback routine
	`MODIFYMENUSTRINGQQ`	Changes a menu item's text string
	`SETWINDOWMENUQQ`	Sets the menu to which a list of current child window names are appended

Category	QuickWin Function	Description
Directional keys	PASSDIRKEYSQQ	Enables (or disables) use of the arrow directional keys and page keys as input
QuickWin messages	SETMESSAGEQQ	Changes any QuickWin message, including status bar messages, state messages and dialog box messages
Mouse actions	REGISTERMOUSEEVENT	Registers the application defined routines to be called on mouse events
	UNREGISTERMOUSEEVENT	Removes the routine registered by REGISTERMOUSEEVENT
	WAITONMOUSEEVENT	Blocks return until a mouse event occurs

Controlling Menus

You do not have to use the default QuickWin menus. You can eliminate and alter menus, menu item lists, menu titles or item titles. The QuickWin functions that control menus are described in the following sections:

- Controlling the Initial Menu and Frame Window
- Deleting, Inserting, and Appending Menu Items
- Modifying Menu Items
- Creating a Menu List of Available Child Windows
- Simulating Menu Selections

Controlling the Initial Menu and Frame Window

You can change the initial appearance of an application's default frame window and menus by defining an INITIALSETTINGS function. If no user-defined INITIALSETTINGS function is supplied, QuickWin calls a predefined (default)INITIALSETTINGS routine that provides a default frame window and menu.

Your application does not call INITIALSETTINGS. If you supply the function in your project, QuickWin calls it automatically.

If you supply it, INITIALSETTINGS can call QuickWin functions that set the initial menus and the size and position of the frame window. Besides the menu functions, SETWSIZEQQ can be called from your INITIALSETTINGS function to adjust the frame window size and position before the window is first drawn.

The following is a sample of INITIALSETTINGS:

```
   LOGICAL FUNCTION INITIALSETTINGS( )
    USE IFQWIN
    LOGICAL result
    TYPE (qwinfo) qwi
 ! Set window frame size.
    qwi%x = 0
    qwi%y = 0
    qwi%w = 400
    qwi%h = 400
    qwi%type = QWIN$SET
    i = SetWSizeQQ( QWIN$FRAMEWINDOW, qwi )
 ! Create first menu called Games.
    result = APPENDMENUQQ(1, $MENUENABLED, '&Games'C, NUL )
```

```
! Add item called TicTacToe.
   result = APPENDMENUQQ(1, $MENUENABLED, '&TicTacToe'C, WINPRINT)
! Draw a separator bar.
   result = APPENDMENUQQ(1, $MENUSEPARATOR, ''C, NUL )
! Add item called Exit.
   result = APPENDMENUQQ(1, $MENUENABLED, 'E&xit'C, WINEXIT )
! Add second menu called Help.
   result = APPENDMENUQQ(2, $MENUENABLED, '&Help'C, NUL )
   result = APPENDMENUQQ(2, $MENUENABLED, '&QuickWin Help'C, WININDEX)
   INITIALSETTINGS= .true.
 END FUNCTION INITIALSETTINGS
```

QuickWin executes your INITIALSETTINGS function during initialization, before creating the frame window. When your function is done, control returns to QuickWin and it does the remaining initialization. The control then passes to the Fortran application.

You only need to include the code for an INITIALSETTINGS function in your project. If it is part of your project, you do not need to call your INITIALSETTINGS function.

The INITIALSETTINGS function can specify the position and size of the frame window and the contents of the menus. Because the INITIALSETTINGS function is executed before creating the frame window, it must not call any routines that require that frame window initialization be complete. For example, routines that refer to the child window in focus (such as SETWINDOWCINFIG) or a specific child window unit number (such as OPEN) should not be called from the INITIALSETTINGS function.

Your INITIALSETTINGS function should return .TRUE. if it succeeds, and .FALSE. otherwise. The QuickWin default function returns a value of .TRUE. only.

Note that default menus are created after INITIALSETTINGS has been called, and only if you do not create your own menus. Therefore, using DELETEMENUQQ, INSERTMENUQQ, APPENDMENUQQ, and the other menu configuration QuickWin functions while in INITIALSETTINGS affects your custom menus, not the default QuickWin menus.

Deleting, Inserting, and Appending Menu Items

Menus are defined from left to right, starting with 1 at the far left. Menu items are defined from top to bottom, starting with 0 at the top (the menu title itself). Within INITIALSETTINGS, if you supply it, you can delete, insert, and append menu items in custom menus. Outside INITIALSETTINGS, you can alter the default QuickWin menus as well as custom menus at any point in your application. (Default QuickWin menus are not created until after INITIALSETTINGS has run and only if you do not create custom menus.)

To delete a menu item, specify the menu number and item number in DELETEMENUQQ. To delete an entire menu, delete item 0 of that menu. For example:

```
USE IFQWIN
LOGICAL status
status = DELETEMENUQQ(1, 2) ! Delete the second menu item from
                           ! menu 1 (the default FILE menu).
status = DELETEMENUQQ(5, 0) ! Delete menu 5 (the default Windows
                           ! menu).
```

INSERTMENUQQ inserts a menu item or menu and registers its callback routine. QuickWin supplies several standard callback routines such as WINEXIT to terminate a program, WININDEX to list QuickWin Help, and WINCOPY which copies the contents of the current window to the Clipboard. A list of available callbacks is given in the *Language Reference* for INSERTMENUQQ and APPENDMENUQQ.

Often, you will supply your own callback routines to perform a particular action when a user selects something from one of your menus.

In general, you should not assign the same callback routine to more than one menu item because a menu item's state might not be properly updated when you change it (put a check mark next to it, gray it out, or disable, or enable it). You cannot insert a menu item or menu beyond the existing number; for example, inserting item 7 when 5 and 6 have not been defined yet. To insert an entire menu, specify menu item 0. The new menu can take any position among or immediately after existing menus.

If you specify a menu position occupied by an existing menu, the existing menu and any menus to the right of the one you add are shifted right and their menu numbers are incremented.

For example, the following code inserts a fifth menu item called `Position` into menu 5 (the default Windows menu):

```
USE IFQWIN
LOGICAL status
status = INSERTMENUQQ (5, 5, $MENUCHECKED, 'Position'C, WINPRINT)
```

The next code example inserts a new menu called `My List` into menu position 3. The menu currently in position 3 and any menus to the right (the default menus View, State, Windows, and Help) are shifted right one position:

```
USE IFQWIN
LOGICAL status
status = INSERTMENUQQ(3,0, $MENUENABLED, 'My List'C, WINSTATE)
```

You can append a menu item with . The item is added to the bottom of the menu list. If there is no item yet for the menu, your appended item is treated as the top-level menu item, and the string you assign to it appears on the menu bar.

The QuickWin menu routines like `INSERTMENUQQ` and `APPENDMENUQQ` let you to create a single level of menu items beneath a menu name. You cannot create submenus with the QuickWin project type.

The following code uses `APPENDMENUQQ` to append the menu item called `Cascade Windows` to the first menu (the default File menu):

```
 USE IFQWIN
LOGICAL status
status = APPENDMENUQQ(1, $MENUCHECKED, 'Cascade Windows'C,   &
& WINCASCADE)
```

The $MENUCHECKED flag in the example puts a check mark next to the menu item. To remove the check mark, you can set the flag to $MENUUNCHECKED in the `MODIFYMENUFLAGSQQ` function. Some predefined routines (such as WINSTATUS) take care of updating their own check marks. However, if the routine is registered to more than one menu item, the check marks might not be properly updated. See `APPENDMENUQQ` or `INSERTMENUQQ` in the *Language Reference* for the list of callback routines and other flags.

Modifying Menu Items

`MODIFYMENUSTRINGQQ` can modify the string identifier of a menu item, `MODIFYMENUROUTINEQQ` can modify the callback routine called when the item is selected, and `MODIFYMENUFLAGSQQ` can modify a menu item's state (such as enabled, grayed out, checked, and so on).

The following example code uses `MODIFYMENUSTRINGQQ` to modify the menu string for the fourth item in the first menu (the File menu by default) to `Tile Windows`, it uses `MODIFYMENUROUTINEQQ` to change the callback routine called if the item is selected to `WINTILE`, and uses `MODIFYMENUFLAGSQQ` to put a check mark next to the menu item:

```
status = MODIFYMENUSTRINGQQ( 1, 4, 'Tile Windows'C)
status = MODIFYMENUROUTINEQQ( 1, 4, WINTILE)
status = MODIFYMENUFLAGSQQ( 1, 4, $MENUCHECKED)
```

Creating a Menu List of Available Child Windows

By default, the Windows menu contains a list of all open child windows in your QuickWin applications. `SETWINDOWMENUQQ` changes the menu which lists the currently open child windows to the menu you specify. The list of child window names is appended to the end of the menu you choose and deleted from any other menu that previously contained it. For example:

```
 USE IFQWIN
LOGICAL status
...
! Append list of open child windows to menu 1 (the default File menu)
status = SETWINDOWMENUQQ(1)
```

Simulating Menu Selections

`CLICKMENUQQ` simulates the effect of clicking or selecting a menu command from the Window menu. The QuickWin application behaves as though the user had clicked or selected the command. The following code fragment simulates the effect of selecting the Tile item from the Window menu:

```
USE IFQWIN
INTEGER status
status = CLICKMENUQQ(QWIN$TILE)
```

Items from the Window menu can be specified in `CLICKMENUQQ`. Other predefined routines such as WINFULLSCREEN and WINSIZETOFIT (see the callback subroutine names listed in `APPENDMENUQQ`) can be invoked by calling `CLICKMENUQQ` with an argument containing the `LOC` intrinsic function before the callback routine name. For example, the following program calls WINSIZETOFIT:

```
! Some of the callback subroutine names listed in APPENDMENUQQ may not
! work or be useful in this context, but they will be "called."
! Run the program and note how the scroll bars disappear
     use IFQWIN
     integer result
     character*1 ch
     print *,'type a character'
     read(*,*) ch
     result = clickmenuqq(loc(WINSIZETOFIT))
     print *,'type a character'
     read(*,*) ch
     end
```

See Also

- Related Information

Changing Status Bar and State Messages

Any string QuickWin produces can be changed by calling `SETMESSAGEQQ` with the appropriate message ID. Unlike other QuickWin message functions, `SETMESSAGEQQ` uses regular Fortran strings, not null-terminated C strings. For example, to change the PAUSED state message to `I am waiting`:

```
USE IFQWIN
CALL SETMESSAGEQQ('I am waiting', QWIN$MSG_PAUSED)
```

This function is useful for localizing your QuickWin applications for countries with different native languages. A list of message IDs is given in `SETMESSAGEQQ` in the *Language Reference* in the *Intel© Visual Fortran Compiler User and Reference Guides*.

Displaying Message Boxes

MESSAGEBOXQQ causes your program to display a message box. You can specify the message the box displays and the caption that appears in the title bar. Both strings must be null-terminated C strings. You can also specify the type of message box. Box types are symbolic constants defined in DFLIB.MOD, and can be combined by means of the IOR intrinsic function or the .OR. operator. The available box types are listed under MESSAGEBOXQQ in the *Language Reference*. For example:

```
 USE IFQWIN
INTEGER response
response = MESSAGEBOXQQ('Retry or Cancel?'C, 'Smith Chart   &
& Simulator'C, MB$RETRYCANCELQWIN .OR. MB$DEFBUTTON2)
```

Defining an About Box

The ABOUTBOXQQ function specifies the message displayed in the message box that appears when the user selects the About command from a QuickWin application's Help menu. (If your program does not call ABOUTBOXQQ, the QuickWin run-time library supplies a default string.) The message string must be a null-terminated C string. For example:

```
USE IFQWIN
INTEGER status
status = ABOUTBOXQQ ('Sound Speed Profile Tables Version 1.0'C)
```

Using Custom Icons

The QuickWin run-time library provides default icons that appear when the user minimizes the application's frame window or its child windows. You can add custom-made icons to your executable files, and Windows will display them instead of the default icons.

To add a custom child window icon to your QuickWin program:

1. From the **File** >menu, select **Add New Item..**.
2. Select **Resource** and enter a new name. Click **Open**.
3. From the **Edit** menu, select **Add Resource...**

   From the list of possible resources, select **Icon**. Click **New**. The screen becomes an icon drawing tool.
4. Draw the icon. (For more information about using the Graphics Editor in the integrated development environment, see the *Visual C++ User's Guide*.)

   -- or --

   If your icon already exists (for example, as a bitmap) and you want to import it, not draw it, select Resource from the Insert menu, then select Import from the buttons in the Resource dialog. You will be prompted for the file containing your icon.
5. Display the Icon Properties dialog box by double-clicking in the icon editor area outside the icon's grid or pressing ALT+ENTER.

   In the ID field on the General tab of Icon Properties dialog box, replace the default icon name with either "frameicon" or "childicon." The frame window's icon must have the name "frameicon," and the child window's icon must have the name "childicon." These names must be entered with quotation marks in order to be interpreted as strings.

   Your icon will be saved in a file with the extension .ICO.

6. Create an `.RC` file to hold your icons. Select File>Save As. You will be prompted for the name of the file that will contain your icons. It must end with the extension .RC; for example, myicons.rc. Using this method, the icons and their string values will be automatically saved in the `.RC` file. (Alternatively, you can create an `.RC` file with any editor and add the icon names and their string values by hand.)
7. Add the file to the project that contains your QuickWin application. Select Build and the `.RC` file will be built into the application's executable.

When you run your application, the icon you created will take the place of the default child or frame icon. Your custom icon appears in the upper-left corner of the window frame. When you minimize the window, the icon appears on the left of the minimized window bar.

Using a Mouse

Your applications can detect and respond to mouse events, such as left mouse button down, right mouse button down, or double-click. Mouse events can be used as an alternative to keyboard input or for manipulating what is shown on the screen.

QuickWin provides two types of mouse functions:

1. *Event-based functions*, which call an application-defined callback routine when a mouse click occurs
2. *Blocking (sequential) functions*, which provide blocking functions that halt an application until mouse input is made

The mouse is an asynchronous device, so the user can click the mouse anytime while the application is running (mouse input does not have to be synchronized to anything). When a mouse-click occurs, Windows sends a message to the application, which takes the appropriate action. Mouse support in applications is most often event-based, that is, a mouse-click occurs and the application does something.

However, an application can use blocking functions to wait for a mouse-click. This allows an application to execute in a particular sequential order and yet provide mouse support. QuickWin performs default processing based on mouse events.

To change the shape of the mouse cursor, use the `SETMOUSECURSOR` routine.

Event-Based Functions

The QuickWin function `REGISTERMOUSEEVENT` registers the routine to be called when a particular mouse event occurs (left mouse button, right mouse button, double-click, and so on). You define what events you want it to handle and the routines to be called if those events occur. `UNREGISTERMOUSEEVENT` unregisters the routines so that QuickWin doesn't call them but uses default handling for the particular event.

By default, QuickWin typically ignores events except when mouse-clicks occur on menus or dialog controls. Note that no events are received on a minimized window. A window must be restored or maximized in order for mouse events to happen within it.

For example:

```
USE IFQWIN
INTEGER result
OPEN (4, FILE= 'USER')
...
result = REGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK, CALCULATE)
```

This registers the routine `CALCULATE`, to be called when the user double-clicks the left mouse button while the mouse cursor is in the child window opened as unit 4. The symbolic constants available to identify mouse events are:

Mouse Event [1]	Description
MOUSE$LBUTTONDOWN	Left mouse button down
MOUSE$LBUTTONUP	Left mouse button up
MOUSE$LBUTTONDBLCLK	Left mouse button double-click
MOUSE$RBUTTONDOWN	Right mouse button down
MOUSE$RBUTTONUP	Right mouse button up
MOUSE$RBUTTONDBLCLK	Right mouse button double-click
MOUSE$MOVE	Mouse moved

[1] For every BUTTONDOWN and BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double-clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double-click interval, set in the system's CONTROL PANEL/MOUSE.

To unregister the routine in the preceding example, use the following code:

```
result = UNREGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK)
```

If `REGISTERMOUSEEVENT` is called again without unregistering a previous call, it overrides the first call. A new callback routine is then called on the specified event.

The callback routine you create to be called when a mouse event occurs should have the following prototype:

```
INTERFACE
 SUBROUTINE MouseCallBackRoutine (unit, mouseevent, keystate, &
  & MouseXpos,MouseYpos)
    INTEGER unit
    INTEGER mouseevent
    INTEGER keystate
    INTEGER MouseXpos
    INTEGER MouseYpos
 END SUBROUTINE
END INTERFACE
```

The `unit` parameter is the unit number associated with the child window where events are to be detected, and the `mouseevent` parameter is one of those listed in the preceding table. The `MouseXpos` and the `MouseYpos` parameters specify the x and y positions of the mouse during the event. The `keystate` parameter indicates the state of the shift and control keys at the time of the mouse event, and can be any ORed combination of the following constants:

Keystate Parameter	Description
MOUSE$KS_LBUTTON	Left mouse button down during event
MOUSE$KS_RBUTTON	Right mouse button down during event
MOUSE$KS_SHIFT	Shift key held down during event

Keystate Parameter	Description
MOUSE$KS_CONTROL	Control key held down during event

QuickWin callback routines for mouse events should do a minimum of processing and then return. While processing a callback, the program will appear to be non-responsive because messages are not being serviced, so it is important to return quickly. If more processing time is needed in a callback, another thread should be started to perform this work; threads can be created by calling the Windows API CreateThread. If a callback routine does not start a new thread, the callback will not be re-entered until it is done processing.

NOTE. In event-based functions, there is no buffering of events. Therefore, issues such as multithreading and synchronizing access to shared resources must be addressed. To avoid multithreading problems, use blocking functions rather than event-based functions. Blocking functions work well in applications that proceed sequentially. Applications where there is little sequential flow and the user jumps around the application are probably better implemented as event-based functions.

Blocking (Sequential) Functions

The QuickWin blocking function `WAITONMOUSEEVENT` blocks execution until a specific mouse input is received. This function is similar to `INCHARQQ`, except that it waits for a mouse event instead of a keystroke.

For example:

```
  USE IFQWIN
 INTEGER mouseevent, keystate, x, y, result
 ...
 mouseevent = MOUSE$RBUTTONDOWN .OR. MOUSE$LBUTTONDOWN
 result = WAITONMOUSEEVENT (mouseevent, keystate, x , y) ! Wait
! until right or left mouse button clicked, then check the keystate
! with the following:
 if ((MOUSE$KS_SHIFT .AND. keystate) == MOUSE$KS_SHIFT) then      &
 & write (*,*) 'Shift key was down'
 if ((MOUSE$KS_CONTROL .AND. keystate) == MOUSE$KS_CONTROL) then  &
 & write (*,*) 'Ctrl key was down'
```

Your application passes a mouse event parameter, which can be any ORed combination of mouse events, to `WAITONMOUSEEVENT`. The function then waits and blocks execution until one of the specified events occurs. It returns the state of the Shift and Ctrl keys at the time of the event in the parameter keystate, and returns the position of the mouse when the event occurred in the parameters x and y.

A mouse event must happen in the window that had focus when `WAITONMOUSEEVENT` was initially called. Mouse events in other windows will not end the wait. Mouse events in other windows cause callbacks to be called for the other windows, if callbacks were previously registered for those windows.

Default QuickWin Processing

QuickWin performs some actions based on mouse events. It uses mouse events to return from the FullScreen mode and to select text and/or graphics to copy to the Clipboard. Servicing the mouse event functions takes precedence over return from FullScreen mode. Servicing mouse event functions does not take precedence over Cut/Paste selection modes. Once selection mode is over, processing of mouse event functions resumes.

QuickWin Programming Precautions

QuickWin Programming Precautions Overview

Two features of QuickWin programming need to be applied thoughtfully to avoid non-responsive programs that halt an application. For example, an application may halt while waiting for a process to execute or input to be entered in a child window.

See Also

- Blocking Procedures
- Callback Routines

Using Blocking Procedures

Procedures that block execution of the program until a given event occurs, such as `READ` or `WAITONMOUSEEVENT`, both of which wait for user input, are called *blocking procedures*. QuickWin child processes can contain multiple callback routines; for example, a different routine to be called for each menu selection and each kind of mouse-click (left button, right button, double-click, and so on).

Problems can arise when a process and its callback routine, or two callback routines within the same process, both contain blocking procedures. This is because each QuickWin child process supports a primary and secondary thread.

As a result of selecting a menu item, a menu procedure may call a blocking procedure, while the main thread of the process has also called a blocking procedure. For example, say you have created a file menu, which contains an option to LOAD a file. Selecting the LOAD menu option calls a blocking function that prompts for a filename and waits for the user to enter the name. However, a blocking call such as `WAITONMOUSEEVENT` can be pending in the main process thread when the user selects the LOAD menu option, so two blocking functions are initiated.

When QuickWin has two blocking calls pending, it displays a message in the status bar that corresponds to the blocking call first encountered. If there are further callbacks with other blocking procedures in the two threads, the status bar may not correspond to the actual input pending, execution can appear to be taking place in one thread when it is really blocked in another, and the application can be confusing and misleading to the user.

To avoid this confusion, you should try not to use blocking procedures in your callback routines. QuickWin will not accept more than one `READ` or `INCHARQQ` request through user callbacks from the same child window at one time. If one `READ` or `INCHARQQ` request is pending, subsequent `READ` or `INCHARQQ` requests will be ignored and -1 will be returned to the caller.

If you have a child window that, in some user scenario, might call multiple callback routines containing `READ` or `INCHARQQ` requests, you need to check the return value to make sure the request has been successful, and if not, take appropriate action, for example, request again.

This protective QuickWin behavior does not guard against multiple blocking calls through mouse selection of menu input options. As a general rule, using blocking procedures in callback routines is not advised, since the results can lead to program flow that is unexpected and difficult to interpret.

Using Callback Routines

All callback routines run in a separate thread from the main program. As a result, all multithread issues apply. In particular, sharing data, drawing to windows, and doing I/O must be properly coordinated and controlled.

QuickWin callback routines, both for menu callbacks and mouse callbacks, should do a minimum of processing and then return. While processing a callback, the program will appear to be non-responsive because messages are not being serviced. This is why it is important to return quickly.

If more processing time is needed in a callback, another thread should be started to perform this work; threads can be created by calling the Windows API CreateThread. If a callback routine does not start a new thread, the callback will not be reentered until it is done processing.

Simulating Nonblocking I/O

QuickWin does not accept unsolicited input. You get beeps if you type into an active window if no READ or GETCHARQQ has been done. Because of this, it is necessary to do a READGETCHARQQ or in order for a character to be accepted. But this type of blocking I/O puts the program to sleep until a character has been typed.

In Fortran Console applications, PEEKCHARQQ can be used to see if a character has already been typed. However, PEEKCHARQQ does not work under Fortran QuickWin applications, since QuickWin has no console buffer to accept unsolicited input. Because of this limitation, PEEKCHARQQc annot be used as it is with Fortran Console applications to see whether a character has already been typed.

One way to simulate PEEKCHARQQ with QuickWin applications is to add an additional thread:

- One thread does a READGETCHARQQ or and is blocked until a character typed.
- The other thread (the main program) is in a loop doing useful work and checking in the loop to see if the other thread has received input.

Using Dialog Boxes for Application Controls 5

Using Dialog Boxes for Application Controls Overview

Dialog boxes are a user-friendly way to solicit application control. As your application executes, you can make a dialog box appear on the screen and the user can click on a dialog box control to enter data or choose what happens next.

Using the dialog routines provided with Intel® Fortran, you can add dialog boxes to your application. These routines define dialog boxes and their controls (scroll bars, buttons, and so on), and call your subroutines to respond to user selections.

There are two types of dialog boxes:

- *Modal* dialog boxes, which you can use with any Fortran project type, including Fortran Windows, Fortran QuickWin (multiple document), Fortran Standard Graphics (QuickWin single document), Fortran Console, Fortran DLL, and Fortran Static library project types.
- *Modeless* dialog boxes, which are typically used with the Fortran Windows Application project type.

When your program displays a modal dialog box (any project type), the user must explicitly enter data and close the dialog box before your application resumes execution.

When your program displays a modeless dialog box, your application continues executing. Unlike a modal dialog box, the user can switch between the modeless dialog box and the other windows in the application.

There are two steps to make a dialog:

1. Specify the appearance of the dialog box and the names and properties of the controls it contains.
2. Write an application that activates those controls by recognizing and responding to user selections.

See Also

- Using the Resource Editor to Design a Dialog Box
- Writing a Dialog Application
- Dialog Routines
- Dialog Controls
- Using Dialog Controls
- Using ActiveX* Controls

Using the Resource Editor to Design a Dialog Box

Designing a Dialog Box Overview

You design the appearance of the dialog box, choose and name the dialog controls within it, and set other control properties with the Dialog Editor. The Dialog Editor is one of the Resource Editors provided by the Microsoft Visual Studio* integrated development environment (IDE).

NOTE. The Dialog Editor is not available from the Visual Studio Shell.

A program's resources are defined in a resource file (typically with an .rc extension). A Microsoft Visual Studio project typically contains a single resource file. The contents of the resource file are displayed in the Resource Editor by double-clicking on the .RC file. The resource file can be created by one of the following:

- When you create a project and use one of the Fortran AppWizards (for example, when using the Fortran Windows Application AppWizard).
- When you save the resources that you define using one of the Resource Editors.

If you create the resource file from the Resource Editors, be sure to add the resource file to your project. It is possible to include additional resource files in a project (see Including Resources Using Multiple Resource Files).

This section describes the steps needed to design a dialog box, and uses as an example a dialog box that converts temperatures between Celsius and Fahrenheit. The code in the example is explained as you read through this section.

In this section, there is an example of how to include the dialog box in a Fortran Console project.

To create a Fortran Console application:

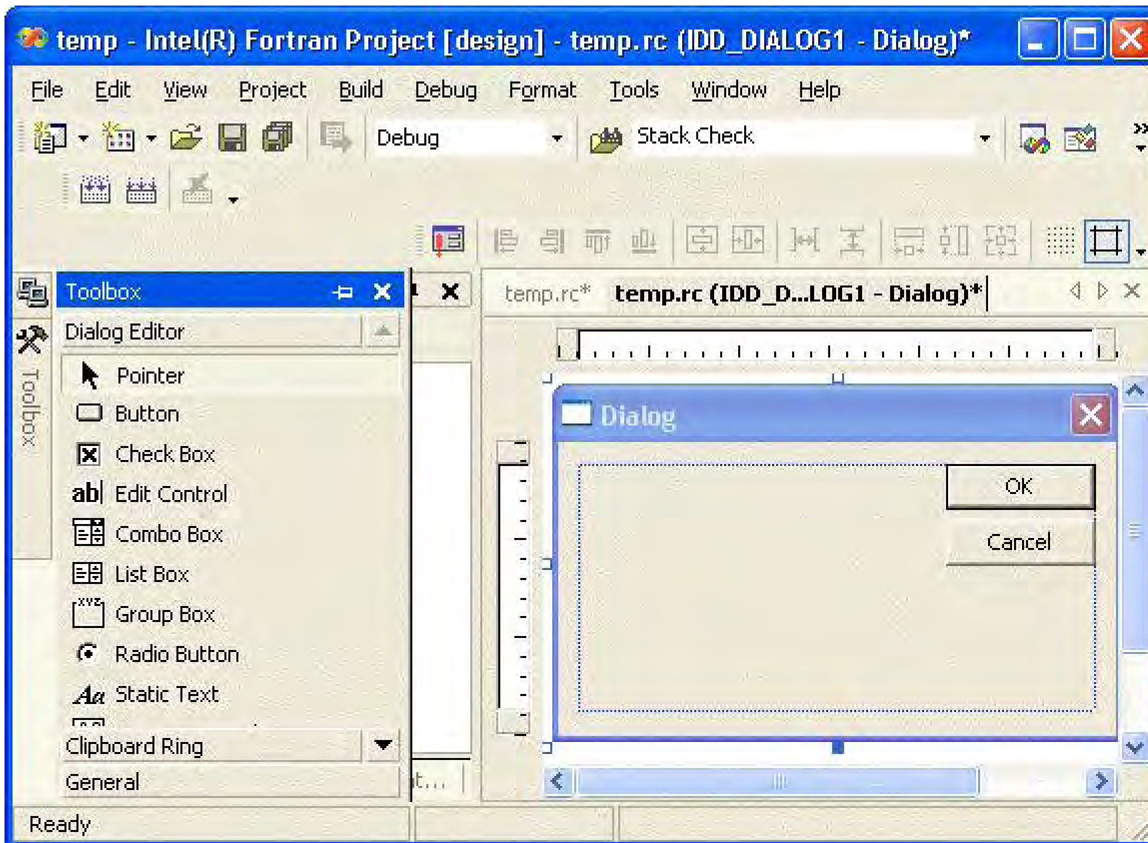
1. From the **File** menu, select **New>Project...**.
2. Select the Intel® Fortran Projects item.
3. From the list of project types, select Console Application.
4. Enter `Temp` as the name for the project, verify where the project will be created, and click OK.
5. In the Fortran Console Application Wizard, click Finish.
6. The Solution View displays the solution.
7. Click the plus sign (+) next to the project name to display the categories of source files, if necessary .

To open the Dialog Editor:

1. From the **File** menu, select **Add New Item...**
2. Select **Resource** and change the name to `Temp.rc`. Click **Open**.
3. Right-click on the .rc file in the Solution View, select **Open With>Resource Editor**.
4. From the **Edit** >menu, select **Add Resource...**. From the list of possible resources, select **Dialog**.

5. Click the New button. The Dialog Editor (one of the resource editors) appears on the screen as shown below.

Dialog Editor Sample 1



A blank dialog box appears along with a toolbox of available controls. If the toolbox does not appear, select **View>Toolbox**.

The controls that are supported by Intel Visual Fortran follow:

Button

Check box

Combo box (such as a drop-down list box)

Edit box

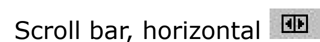
Group box

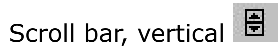
List box

Picture

Progress bar

Radio button

Scroll bar, horizontal 

Scroll bar, vertical 	Slider
Spin control	Static text
Tab control	

You can also add ActiveX* controls to your dialog box (see Using ActiveX* Controls).

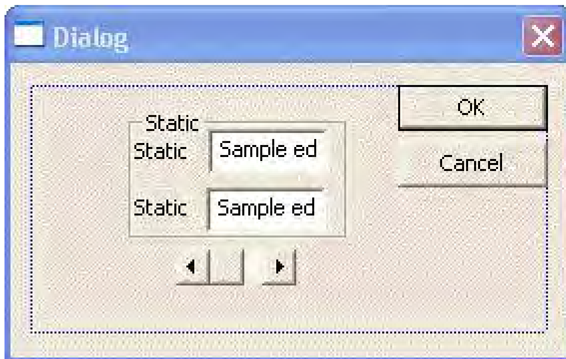
To add controls to the dialog box:

1. Point at one of the available controls in the Toolbox, then hold down the left mouse button and drag the control to the dialog box.
2. Place the dialog control where you want it on the dialog box and release the mouse button. You can delete controls by selecting them with the mouse, then pressing the Delete (or DEL) key.

The following figure shows the dialog box after adding two Static text lines (which currently say "Static"), two Edit boxes (which currently say "Sample edit"), a Horizontal Scroll bar, and a Group box. The Group box is the outlined rectangular area in the dialog box that encloses the other related controls.

The OK and CANCEL buttons were added for you by the Resource Editor. You can delete (select the control and press DEL key), move (drag the control), resize (drag one of the anchor points), or rename the OK and CANCEL buttons or any of the controls that you add.

Dialog Editor Sample 2

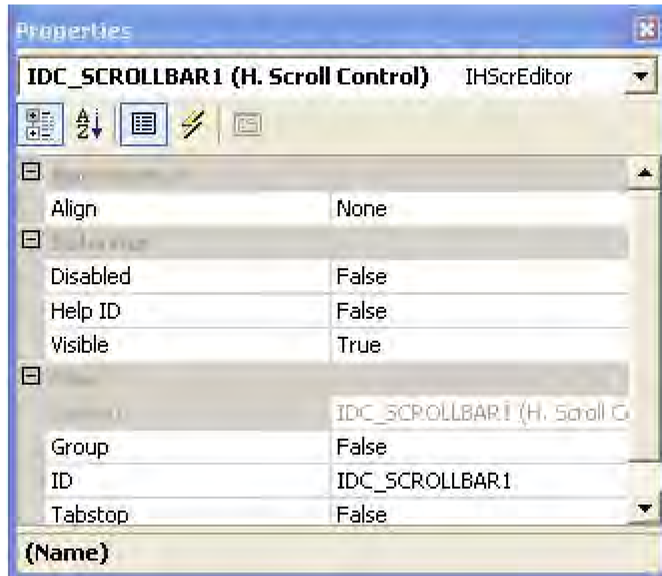


To specify the names and properties of the added controls:

1. Click on one of the controls in your dialog box and select **Properties** from the pop-up (right-click) menu. A Properties window appears showing the default name and properties for that control.

   The following figure shows the Properties box for the Horizontal Scroll bar.

   Dialog Editor Sample 3



2. Change the control name by modifying the ID property (currently, `IDC_SCROLLBAR1`) to another name (for example, `IDC_SCROLLBAR_TEMPERATURE`).
3. Select the available options in the left column to change the control's properties.

Repeat the same process for each control and for the dialog box itself.

To use the controls from within a program, you need symbolic names for each of them. In the previous series of steps, the Horizontal Scroll bar symbolic name has been changed in the Properties box to `IDC_SCROLLBAR_TEMPERATURE`. Use this name to refer to the control; for example:

```
INTEGER slide_position
retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, &
               slide_position, DLG_POSITION)
```

To save the dialog box as a resource file:

1. Select **File>Save All**.

   Typically, only one resource file is used with each Intel Visual Fortran project (see Including Resources Using Multiple Resource Files).
2. When you save your resource file for the first time, the resource editor also creates a Resource.h file. This file is included by the .RC file. To create the Fortran equivalent of this file (resource.fd), do the following:
   a. Add Resource.h to your project using **File>Add Existing Item...**.
   b. Select the Resource.h file in the Solution View and the select **View>Property Pages** from the main menu bar.
   c. Set the Command Line option to use the deftofd tool by entering:

      ```
      deftofd resource.h resource.fd
      ```

      (For more information on the deftofd tool, see The Include (.FD and .H) Files.)
   d. Set the Description option to **Generating Fortran include file...**
   e. Set the Outputs option to `resource.fd`.
   f. Click **OK**.

To open an existing dialog box in the Resource Editor:

1. From the **File** menu, open the project .
2. Double-click the RC file.
3. Click the plus sign (+) next to Dialog.
4. Double-click the appropriate dialog name, such as IDD_DIALOG1.
5. Use the Resource Editor to add a new control or modify an existing control. To modify an existing control, use the Tab key to select the appropriate control. Now select **View>Properties Window** (or, on the pop-up menu, click **Properties**) to view or modify its properties.

At this point, the appearance of the dialog box is finished and the controls are named, but the box cannot function on its own. An application must be created to run it.

Not all the controls on the Resource Editor Controls toolbar are supported by Intel Visual Fortran dialog routines. The supported dialog controls are:

- Button
- Check box
- Combo box (such as a drop-down list box)
- Edit box
- Group box
- List box
- Picture
- Progress bar
- Radio button
- Scroll bar (Horizontal and Vertical)
- Slider
- Spin control
- Static text
- Tab control

You can also add ActiveX controls to your dialog box. For information, see Using ActiveX* Controls.

For further information on resources and control properties, see:

- Setting Control Properties
- Including Resources Using Multiple Resource Files
- The Include (.FD and.H) File

For information on creating an application for your dialog box, see Writing a Dialog Application.

Setting Control Properties

In addition to changing the properties of its individual controls, you can change the properties of the dialog box itself. To change the dialog box properties, select the dialog box and then right-click and select Properties from the pop-up menu in any clear area in the box. The Properties window opens for the dialog.

To specify the location of your dialog box on the screen, do one of the following:

- In the Properties window (in the Position category), you can change the X Pos and Y Pos values. These specify the pixel position of the dialog box's upper-left corner, relative to its parent window. For a modal or a popup modeless dialog box, the parent window is the screen. For example, specifying the X position as 40 and Y position as 40 would place a modal dialog box in the upper-left corner of the screen.
- You can also center the dialog box by setting the Center option to True in the Position category. This displays the dialog box in the center of its parent window. If you set the Center Mouse option to True in the Misc category, the dialog is centered at the current mouse pointer position.

To change the size of the dialog box, hold down the left mouse button as you drag the right or lower perimeter of the box. If you have sized your dialog window to be larger than the edit window, use the scroll bars to move the view region.

Help is available at the bottom of the Properties window to explain the options for each of the dialog controls.

You can edit the appearance of the dialog box later. For more information, see Designing a Dialog Box Overview.

Including Resources Using Multiple Resource Files

Normally it is easy and convenient to work with the default arrangement of all resources in one resource definition `(.RC)` file. However, you can add resources in other files to your current project at compile time by listing them in the compile-time directives box in the Resource Includes dialog box.

There are several reasons to place resources in a file other than the main .RC file:

- To include resources that have already been developed and tested and do not need further modification.
- To include resources that are being used by several different projects, or that are part of a source code version-control system, and thus must exist in a central location where modifications will affect all projects.
- To include resources that are in a custom format.
- To include statements in your resource file that execute conditionally at compile time using compiler directives, such as #ifdef and #else. For example, your project may have a group of resources that are bracketed by #ifdef _DEBUG ... #endif and are thus included only if the constant _DEBUG is defined at compile time.
- To include statements in your resource file that modify resource-file syntax by using the #define directive to implement simple macros.

If you have sections in your existing .RC files that meet any of these conditions, you should place the sections in one or more separate .RC files and include them in your project using the Resource Includes dialog box.

To include resource files that will be added to your project at compile time:

1. Place the resources in an `RC` file with a unique filename. (Do not use *`projectname`*.rc, since this is the default filename used for the main resource script file.)
2. From the Edit menu, choose Resource Includes.
3. In the Compile-time directives box, use the #include compiler directive to include the new resource file in the main resource file in the development environment. The resources in files included in this way are made a part of your executable file at compile time. They are not directly available for editing or modification when you are working on your project's main .RC file. You need to open included .RC files separately.
4. Click **OK**.

The Include (.FD and .H) Files

Each control in a dialog box has a unique integer identifier. When the Resource Editor creates the C language include file (.H), it assigns each control and the dialog box an integer value. You can read the list of names and values in your dialog boxes include file (for example, `RESOURCE.H`). To view and modify the named constants, click Resource Symbols in the Edit menu.

Intel Fortran provides a tool named DEFTOFD that recognizes the definitions in a C language include file (.H) and creates a corresponding Fortran include file (.FD). The Fortran include file uses the PARAMETER attribute to define named constants for each resource.

The resource .H file must be added to your project and assigned a Custom Build step in order to recreate the resource.fd file when resource.h changes. (See Designing a Dialog Box Overview for more information.)

When your application uses a control, it can refer to the control or dialog box by its name (for example, `IDC_SCROLLBAR_TEMPERATURE` or `IDD_TEMP`), or by its integer value. If you want to rename a control or make some other change to your dialog box, you should make the change through the Resource Editor in the integrated development environment. Do not use a text editor to alter your .H or .FD include file because the dialog resource will not be able to access the changes.

Writing a Dialog Application

Writing a Dialog Application Overview

When creating a dialog box with the Resource Editor, you specify the types of controls that are to be included in the box. You then must provide procedures to make the dialog box active. These procedures use both dialog routines and your subroutines to control your program's response to the user's dialog box input.

You give your application access to your dialog resource by adding the .RC file to your project, giving your application access to the dialog include file, and associating the dialog properties in these files with the `dialog` type (see Initializing and Activating the Dialog Box).

Your application must include the statement `USE IFLOGM` to access the dialog routines, and it must include the `.FD` file that the Resource Editor created for your dialog `using DEFTOFD`. For example:

```
PROGRAM TEMPERATURE
USE IFLOGM
IMPLICIT NONE
INCLUDE 'RESOURCE.FD'
.
. ! Call dialog routines, such as DlgInit, DlgModal, and DlgUninit
.
END PROGRAM TEMPERATURE
```

The following sections describe how to code a dialog application:

- Initializing and Activating the Dialog Box
- Using Dialog Callback Routines
- Using a Modeless Dialog Box
- Using Fortran AppWizards to Help Add Modal Dialog Box Coding
- Using Fortran AppWizards to Help Add Modeless Dialog Box Coding
- Using Dialog Controls in a DLL

Initializing and Activating the Dialog Box

Each dialog box has an associated variable of the derived type `dialog`. The `dialog` derived type is defined in the IFLOGM.F90 module; you access it with **USE IFLOGM**. When you write your dialog application, refer to your dialog box as a variable of type `dialog`. For example:

```
  USE IFLOGM
  INCLUDE 'RESOURCE.FD'
  TYPE (dialog) dlg
  LOGICAL return

return = DLGINIT( IDD_TEMP, dlg )
```

This code associates the `dialog` type with the dialog (`IDD_TEMP` in this example) defined in your resource and include files (`TEMP.RC` and `RESOURCE.FD` in this example).

You give your application access to your dialog resource by adding the .RC file to your project. You give your application access to the dialog include file by including the .FD file in each subprogram. You associate the dialog properties in these files with the `dialog` type by calling `DLGINIT` with your dialog name.

An application that controls a dialog box should perform the following actions:

1. Call `DLGINIT or DLGINITWITHRESOURCEHANDLE` to initialize the `dialog` type and associate your dialog and its properties with the type.
2. Initialize the controls with the dialog set routines, such as `DLGSET`.
3. Set the callback routines to be executed when a user manipulates a control in the dialog box with `DLGSETSUB`.
4. Depending on whether you want a modal or modeless dialog type:
    - To use a modal dialog, run the dialog with `DLGMODAL`.
    - To use a modeless dialog, call `DLGMODELESS` and use `DLGISDLGMESSAGE` in your message loop.
5. Retrieve control information with the dialog get functions, such as `DLGGET`.
6. Free resources from the dialog with `DLGUNINIT`.

As an example of activating a dialog box and controls, the following code initializes the temperature dialog box and controls created in the TEMP project example. It also sets the callback routine as `UpdateTemp`, displays the dialog box, and releases the dialog resources when done:

```
  SUBROUTINE DoDialog( )
  USE IFLOGM
  IMPLICIT NONE
  INCLUDE 'RESOURCE.FD'
  INTEGER retint
  LOGICAL retlog
  TYPE (dialog) dlg
  EXTERNAL UpdateTemp
! Initialize.
  IF ( .not. DlgInit( idd_temp, dlg ) ) THEN
   WRITE (*,*) "Error: dialog not found"
  ELSE
! Set up temperature controls.
   retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGEMAX)
   retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, "100" )
   CALL UpdateTemp( dlg, IDC_EDIT_CELSIUS, DLG_CHANGE)
   retlog = DlgSetSub( dlg, IDC_EDIT_CELSIUS, UpdateTemp )
   retlog = DlgSetSub( dlg, IDC_EDIT_FAHRENHEIT, UpdateTemp )
```

```
   retlog = DlgSetSub( dlg, IDC_SCROLLBAR_TEMPERATURE, UpdateTemp )
! Activate the modal dialog.
   retint = DlgModal( dlg )
! Release dialog resources.
   CALL DlgUninit( dlg )
  END IF
  END SUBROUTINE DoDialog
```

The dialog routines, such as DLGSETDLGSETSUB and, refer to the dialog controls by the names you assigned to them in the Properties box while creating the dialog box in the Resource Editor. For example:

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGEMAX)
```

In this statement, the dialog function DLGSET assigns the control named IDC_SCROLLBAR_TEMPERATURE a value of 200. The index DLG_RANGEMAX specifies that this value is a scroll bar maximum range. Consider the following:

```
retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, "100" )
CALL UpdateTemp( dlg, IDC_EDIT_CELSIUS, DLG_CHANGE)
```

The preceding statements set the dialog's top Edit box, named IDC_EDIT_CELSIUS in the Resource Editor, to an initial value of 100, and calls the routine UpdateTemp to inform the application that the value has changed. Consider the following:

```
retlog = DlgSetSub( dlg, IDC_EDIT_CELSIUS, UpdateTemp )
retlog = DlgSetSub( dlg, IDC_EDIT_FAHRENHEIT, UpdateTemp )
retlog = DlgSetSub( dlg, IDC_SCROLLBAR_TEMPERATURE, UpdateTemp )
```

The preceding statements associate the callback routine UpdateTemp with the three controls. This results in the UpdateTemp routine being called whenever the value of any of the three controls changes.

Routines are assigned to the controls with the function DLGSETSUB. Its first argument is the dialog variable, the second is the control name, the third is the name of the routine you have written for the control, and the optional fourth argument is an index to select between multiple routines. You can set the callback routines for your dialog controls anywhere in your application: before opening your dialog with either DLGMODAL or DLGMODELESS, or from within another callback routine.

In the TEMP example, the main program calls the DoDialog subroutine to display the Temperature Conversion dialog box.

Using Dialog Callback Routines

All callback routines should have the following interface:

```
SUBROUTINE callback ( dlg, control_name, callbacktype)
!DEC$ ATTRIBUTES DEFAULT:callback-routine-name
```

Where:

dlg	Refers to the dialog box and allows the callback to change values of the dialog controls.
control_name	Is the name of the control that caused the callback.
callbacktype	Indicates what callback is occurring (for example, DLG_CLICKED, DLG_CHANGE, DLG_DBLCLICK).

In the !DEC$ ATTRIBUTES directive, the *callback-routine-name* is the name of the callback routine.

The last two arguments let you write a single subroutine that can be used with multiple callbacks from more than one control. Typically, you do this for controls comprising a logical group. For example, all the controls in the temperature dialog in the TEMP example are associated with the same callback routine, UpdateTemp. You can also associate more than one callback routine with the same control, but you must then provide an index parameter to indicate which callback is to be used.

The following is an example of a callback routine:

```
  SUBROUTINE UpdateTemp( dlg, control_name, callbacktype )
  !DEC$ ATTRIBUTES  DEFAULT:UpdateTemp
  USE IFLOGM
  IMPLICIT NONE
  TYPE (dialog) dlg
  INTEGER control_name
  INTEGER callbacktype
  INCLUDE 'RESOURCE.FD'
  CHARACTER(256) text
  INTEGER cel, far, retint
  LOGICAL retlog
! Suppress compiler warnings for unreferenced arguments.
  INTEGER local_callbacktype
  local_callbacktype = callbacktype
  SELECT CASE (control_name)
   CASE (IDC_EDIT_CELSIUS)
   ! Celsius value was modified by the user so
   ! update both Fahrenheit and Scroll bar values.
     retlog = DlgGet( dlg, IDC_EDIT_CELSIUS, text )
     READ (text, *, iostat=retint) cel
     IF ( retint .eq. 0 ) THEN
       far = (cel-0.0)*((212.0-32.0)/100.0)+32.0
       WRITE (text,*) far

retlog = DlgSet( dlg, IDC_EDIT_FAHRENHEIT,                    &
   &                    TRIM(ADJUSTL(text)) )
       retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel,         &
   &                    DLG_POSITION )
     END IF
   CASE (IDC_EDIT_FAHRENHEIT)
   ! Fahrenheit value was modified by the user so
   ! update both celsius and Scroll bar values.
      retlog = DlgGet( dlg, IDC_EDIT_FAHRENHEIT, text )
      READ (text, *, iostat=retint) far
      IF ( retint .eq. 0 ) THEN
        cel = (far-32.0)*(100.0/(212.0-32.0))+0.0
        WRITE (text,*) cel
        retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, TRIM(ADJUSTL(text)) )
        retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel,       &
   &                     DLG_POSITION )
      END IF
   CASE (IDC_SCROLLBAR_TEMPERATURE)
   ! Scroll bar value was modified by the user so
   ! update both Celsius and Fahrenheit values.
      retlog = DlgGet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel,         &
   &                  DLG_POSITION  )
      far = (cel-0.0)*((212.0-32.0)/100.0)+32.0
      WRITE (text,*) far
      retlog = DlgSet( dlg, IDC_EDIT_FAHRENHEIT, TRIM(ADJUSTL(text)) )
      WRITE (text,*) cel
      retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, TRIM(ADJUSTL(text)) )
   END SELECT
  END SUBROUTINE UpdateTemp
```

Each control in a dialog box, except a pushbutton, has a default callback that performs no action. The default callback for a pushbutton's click event sets the return value of the dialog to the pushbutton's name and then exits the dialog. This makes all pushbuttons exit the dialog by default, and gives the OK and CANCEL buttons good default behavior. A routine that calls `DLGMODAL` can then test to see which pushbutton caused the modal dialog to exit.

Callbacks for a particular control are called after the value of the control has been changed by the user's action. Calling DLGSETdoes not cause a callback to be called for the changing value of a control. In particular, when inside a callback, performing a DLGSET on a control will not cause the associated callback for that control to be called.

Calling DLGSET before or afterDLGMODELESS or DLGMODAL has been called also does not cause the callback to be called. If the callback needs to be called, it can be called manually using CALL after the DLGSET is performed.

Using a Modeless Dialog Box

When an application displays a modeless dialog box, the application does not halt, waiting for user input, but continues executing. The user can freely switch between the modeless dialog box and the other windows in the application.

To display a modeless dialog box, call the DLGMODELESS function. A modeless dialog box remains displayed until the DLGEXIT routine is called, either explicitly or by a default button callback. The application must provide a message loop to process Windows messages and must call the DLGISDLGMESSAGE function at the beginning of the message loop.

The variable of type DIALOG passed to DLGMODELESS must remain in memory for the duration of the dialog box (from the DLGINIT call through the DLGUNINIT call). The variable can be declared as global data in a Fortran module, as a variable with the STATIC attribute (or statement), or in a calling procedure that is active for the duration on the dialog box. For more information, see the Syntax for DLGMODELESS.

Modeless dialog boxes are typically used in a Fortran Windows project. A modeless dialog box can be used in a Fortran Console, Fortran DLL, or Fortran Static Library application as long as the requirements for using a modeless dialog box (discussed in the previous paragraphs) are met.

As an example of using a modeless dialog box, the following code is the WinMain function of an application that displays a modeless dialog box as its main window:

```
    INTEGER(DWORD) function WinMain (hInstance, hPrevInstance, &
& lpszCmdLine, nCmdShow)
    !DEC$ ATTRIBUTES STDCALL, DECORATE, ALIAS:”WinMain”  :: WinMain
    use IFWIN
    use IFLOGM
    INTEGER(HANDLE), INTENT(IN) :: hInstance, hPrevInstance
    INTEGER(LPVOID), INTENT(IN) :: lpszCmdLine
    INTEGER(DWORD), INTENT(IN) :: nCmdShow
    ! Include the constants provided by the Resource Editor
    include 'resource.fd'
    ! A dialog box callback
    external ThermometerSub
    ! Variables
    type (dialog) dlg
    type (T_MSG)  mesg
    integer ret
    logical lret
    ! Create the thermometer dialog box and set up the controls and callbacks
    lret = DlgInit(IDD_THERMOMETER, dlg_thermometer)
    lret = DlgSetSub(dlg_thermometer, IDD_THERMOMETER, ThermometerSub)
    lret = DlgSet(dlg_thermometer, IDC_PROGRESS1, 32, DLG_RANGEMIN)
    lret = DlgSet(dlg_thermometer, IDC_PROGRESS1, 212, DLG_RANGEMAX)
    lret = DlgSet(dlg_thermometer, IDC_PROGRESS1, 32)
    lret = DlgModeless(dlg_thermometer, nCmdShow)

    ! Read and process messages until GetMessage returns 0 because
    ! PostQuitMessage has been called
    do while( GetMessage (mesg, NULL, 0, 0) )
       ! Note that DlgIsDlgMessage must be called in order to give
```

```
       ! the dialog box first chance at the message.
       if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
           lret = TranslateMessage( mesg )
           ret  = DispatchMessage( mesg )
       end if
    end do
    ! Cleanup dialog box memory
    call DlgUninit(dlg)
    ! The return value is the wParam of the Quit message
    WinMain = mesg.wParam
    return
  end function
```

Using Fortran AppWizards to Help Add Modal Dialog Box Coding

When an application displays a modal dialog box, the user must explicitly enter data and close the dialog box before the application resumes execution. Any Fortran project type can use a modal dialog box.

The following creates a "Hello World" Fortran Console application that uses a modal dialog box to display "Hello World!" The first step may vary, depending on your version of Visual Studio*.

1. From the list of Fortran project types, select **Console Application**. In the right pane, select **Main Program Code**. At the bottom of the window, specify a file name (`HelloDlg`) and location. Your project and source file (`HelloDlg.f90`) will be created for you.
2. In the **File** menu, select **Add New Item...** and select **Resource**. Specify a name of `HelloDlg.rc` and click **Open**. Select **Edit>Add Resource...**, select **Dialog**, and click **New**. Create the box using the dialog resource editor, as follows:
   a. Delete the **Cancel** button (click the **Cancel** button and press the **Delete** key).
   b. Add a new static text control to the dialog box.
   c. Enlarge or resize the static text control if needed.
   d. Select the static text control, then select **View>Properties Window** or (right-click and choose Properties) to edit its properties. Change the Caption to "Hello World!." You may also want to change the **Align Text** option to Center.
   e. Dismiss the dialog box by clicking the x in the upper-right corner of the window.
3. In the **File** menu, select **Save All**.
4. In the **Project** menu, select **Add Existing Item....** Change the **Files of Type:** selection to **All Files** and select Resource.h. Click **Open**.
5. Select the Resource.h file in the Solution View and the select **View>Property Pages**.
6. Set the options as follows:
   - Command Line: deftofd resource.h resource.fd.
   - Description: Generating Fortran include file...
   - Outputs: resource.fd.
7. Click **OK**.
8. Edit the file `HelloDlg.f90` as follows:
   - After the `program HELLODLG` line, add the following line:

     ```
     USE IFLOGM
     ```

- Replace the line: `print *, 'Hello World'`

  With the following lines:

```
include 'resource.fd'
type (DIALOG) dlg
logical lret
integer iret
lret = DlgInit(IDD_DIALOG1, dlg)
iret = DlgModal(dlg)
call DlgUninit(dlg)
```

In the code above:

- The `USE IFLOGM` line includes the IFLOGM module interfaces to the Dialog routines.
- The line `include 'resource.fd'` includes the .fd file.
- The function reference to `DLGINIT` initializes the dialog box.
- The function reference to `DLGMODAL` displays the dialog box.
- The call to `DLGUNINIT` frees the dialog box resources.

9. Build the Hellodlg Fortran Console project application. When you execute the application, the dialog box you created appears in front of the Console window:

10. Click **OK** to dismiss the dialog box.

For Intel Visual Fortran applications using the Fortran Windows project type, you can use the Fortran Windows Project AppWizard to help you add dialog coding for a modeless dialog box.

For information about coding requirements for modal and modeless dialog boxes, see Initializing and Activating the Dialog Box.

Using Fortran AppWizards to Help Add Modeless Dialog Box Coding

To use a modeless dialog box, you typically use a Fortran Windows project type. The Fortran Windows Project AppWizard helps you add coding for using a modeless dialog box.

When you create a project by selecting **Fortran Windows Application>Windowing Application**, a number of wizards are available. However, only two allow creation of a dialog box that serves as the main window of the application. These are:

- Sample SDI (Single Document Interface) code

  To create an application where a dialog box is the main window of the application with a menu bar, choose the SDI Code wizard or ActiveX SDI Code wizard.

  This also creates the skeleton of an entire application that you can immediately build and run. You can add a dialog box to the client area of the main window (as explained later).
- Sample Dialog Code

  To create an application where a dialog box is the main window of the application, without a menu bar, choose the Dialog Code wizard or ActiveX Dialog Code wizard.

  This creates the skeleton of an entire application that you can immediately build and run to display a sample dialog box. You can add controls to the dialog box and add dialog procedure calls to manipulate the controls and handle dialog callbacks.

Single Document Interface Sample Code

In the template-like code generated when you select the SDI Code or ActiveX SDI Code option, do the following to add the dialog box to the client area of the main window:

1. Double-click the .rc file, select **Edit>Add Resource...** and create a new Dialog box. Edit the Dialog Properties. Set Style to Child, Border to Thin, and Title Bar to False .
2. In the main source file, add the following USE statement:
   ```
   USE iflogm
   ```
3. In the main source file, in the function MainWndProc, add a case to handle the WM_CREATE message. In this case, initialize the dialog box in the normal manner. To display the dialog box, call:
   ```
   lret = DlgModeless(dlg, SW_SHOWNA, hwndParent)
   ```
   In this call, hwndParent is the window handle of the application's main window.
4. In the main source file, add a call to `DlgIsDlgMessage` to the message loop, before the call to the Windows API routine TranslateAccelerator, as shown in the following:
   ```
   !  Read and process messages
   do while( GetMessage (mesg, NULL, 0, 0) )
   if ( DlgIsDlgMessage(mesg, dlg) .EQV. .FALSE. ) then
   ```

```
if ( TranslateAccelerator (mesg%hwnd, haccel, mesg) == 0) then
lret = TranslateMessage( mesg )
ret  = DispatchMessage( mesg )
end if
end if
end do
```

5. Optionally, if you want to allow the user to resize the main window, add a case to handle the WM_RESIZE message and change the layout of the dialog box based upon its size.

Dialog-Based Sample Code

In the template-like code generated when you select the Dialog code or ActiveX Dialog code option, do the following:

- Build the project and execute it. The following dialog box is displayed:

- Some of the code specific to the dialog routine interfaces and data declarations follows. For this example, the project name is FWin. The project name is used in some of the data declarations:

```
use iflogm
use FWin_dialogGlobals
.
.
.
include 'resource.fd'
external FWin_dialogSub
external FWin_dialogApply
! Variables
type (T_MSG) mesg
integer    ret
logical    lret
```

The FWin_dialogGlobals module is defined in a separate source file in that project. The FWin_dialogSub and FWin_dialogApply are subroutines defined later in the main source file that are callback routines for different controls for the dialog box.

- The code specific to creating the dialog follows:

```
lret = DlgInit(IDD_FWIN_DIALOG_DIALOG, gdlg)           1
if (lret == .FALSE.) goto 99999
lret = DlgSetSub(gdlg, IDD_FWIN_DIALOG_DIALOG, FWin_dialogSub) 2
lret = DlgSetSub(gdlg, IDM_APPLY, FWin_dialogApply)      3
lret = DlgModeless(gdlg, nCmdShow)                      4
if (lret == .FALSE.) goto 99999
```

Notes for this example:

[1] `DlgInit` initializes the dialog box.

[2] The first call to `DlgSetSub` assigns a callback subroutine to the Exit button. It associates the FWin_dialogSub subroutine with the dialog box identifier IDD_FWIN_DIALOG_DIALOG (project name is FWin_Dialog). The FWin_dialogSub routine contains code to terminate the program.

[3] The second call to `DlgSetSub` associates FWin_dialogApply with the Apply button identifier IDM_APPLY. The user should add code in the FWin_dialogApply subroutine to take appropriate action.

[4] `DlgModeless` displays the initialized modeless dialog box, which is ready for user input.

- The code specific to processing messages (message loop) to react to user input follows:

```
! Read and process messages
do while( GetMessage (mesg, NULL, 0, 0) )         1
if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then  2
lret = TranslateMessage( mesg )            3
ret  = DispatchMessage( mesg )             4
end if
end do
```

Notes for this example:

[1] The GetMessage Windows API call inside a `DO WHILE` loop returns a message from the calling thread's message queue.

[2] `DlgIsDlgMessage` determines whether the specified message is intended for one of the currently displayed modeless dialog boxes, or a specific dialog box.

[3] The TranslateMessage Windows API call translates virtual-key messages into character messages.

[4] The DispatchMessage Windows API call dispatches a message to a window procedure.

- The dialog box is terminated and its resources are released by calling `DlgUninit`:

  ```
  call DlgUninit(gdlg)
  ```

Using Dialog Controls in a DLL

You can use a dialog box that is defined in a DLL. To do so, you must inform the dialog routines that the dialog resource is located in the DLL, rather than in the main application. The dialog routines will look for the dialog resource in the main application by default.

To do this, initialize your dialog box using `DlgInitWithResourceHandleDlgInit` rather than `DlgInit` . As compared to `DlgInit`, `DlgInitWithResourceHandle` takes an additional argument named "hinst". The "hinst" argument is the module instance handle in which the dialog resource can be found. For a DLL, this handle is passed into the DLL entry point, DllMain.

An example of a DllMain function follows:

```
module dll_globals
  integer ghInst    ! DLL instance handle
end module dll_globals
!*********************************************************************
!*   FUNCTION: DllMain(HANDLE, DWORD, LPVOID)
!*
!*   PURPOSE:  DllMain is called by Windows when
!*     the DLL is initialized, Thread Attached, and other times.
!*     Refer to SDK documentation, as to the different ways this
!*     may be called.
!*
!*     The DllMain function should perform additional initialization
!*     tasks required by the DLL. DllMain should return a value of 1
!*     if the initialization is successful.
!*
!*********************************************************************
integer(DWORD) function DllMain (hInstDLL, fdwReason, lpvReserved)
!DEC$ ATTRIBUTES STDCALL, DECORATE, ALIAS:"DllMain" :: DllMain
  use IFWIN
  use dll_globals
  integer(HANDLE), intent(IN) :: hinstFDLL
  integer(DWORD), intent(IN) :: fswReason
  integer(LPVOID), intent(IN) :: lpvReserved
  !  Save the module instance handle in a global variable
  !  This would typically be in a Module or a COMMON block.
  ghInst =  hInst
  DllMain = 1
  return
end
```

One way to use `DlgInitWithResourceHandle` is to build a resource-only DLL. A resource-only DLL contains an .RC file, but no code. It is useful for building an application that supports multiple languages. You can create a main application and several resource-only DLLs (one for each language) and call the Windows API LoadLibrary routine at the beginning of your application to load the appropriate resource-only DLL. To use a dialog box from the resource-only DLL, first call LoadLibrary (see the Platform SDK online documentation) to return the instance handle that you can use when you call `DlgInitWithResourceHandle`.

When you create a Fortran DLL project, you can create a resource-only DLL using the Fortran Dynamic Link Library AppWizard.

To create a resource-only DLL:

1. Select **Library** as the Intel® Fortran project type.
2. Select **Dynamic-link Library** in the right pane.
3. Complete creating the project.
4. In the Project menu, select Add to Project... Files to add your .RC file and the RESOURCE.H file that defines the identifiers of the controls.
5. In the Project menu:
   - Select Properties
   - Select **Linker>Advanced**
   - Set **Resource Only DLL** to **Yes**

Summary of Dialog Routines

You can use dialog routines as you would any intrinsic procedure or run-time routine.

As described in Using Dialog Boxes in Application Controls, Intel Fortran supports two types of dialog boxes: modal and modeless. You can use a modal dialog box with any Fortran project type. You can use a modeless dialog box only with the Fortran Windows project types.

The dialog routines can:

- Initialize and close the dialog box
- Retrieve user input from a dialog box
- Display data in the dialog box
- Modify the dialog box controls

The include file (.FD) of the dialog box contains the names of the dialog controls that you specified in the Properties Window of the Resource Editor when you created the dialog box. The module IFLOGM.MOD contains predefined variable names and type definitions. These control names, variables, and type definitions are used in the dialog routine argument lists to manage your dialog box.

The dialog routines are listed in the following table:

Dialog Routine	Description
DLGEXIT	Closes an open dialog
DLGFLUSH	Updates the dialog display
DLGGET	Gets the value of a control variable
DLGGETCHAR	Gets the value of a character control variable
DLGGETINT	Gets the value of an integer control variable
DLGGETLOG	Gets the value of a logical control variable
DLGINIT	Initializes the dialog
DLGINITWITHRESOURCEHANDLE	Initializes the dialog (alternative to DLGINIT)
DLGISDLGMESSAGE	Determines whether a message is intended for a modeless dialog box
DLGISDLGMESSAGEWITHDLG	Determines whether a message is intended for a modeless dialog box (alternative to DLGISDLGMESSAGE)
DLGMODAL	Displays a modal dialog box
DLGMODALWITHPARENT	Displays a modal dialog box (alternative to DLGMODAL)
DLGMODELESS	Displays a modeless dialog box

Dialog Routine	Description
DLGSENDCTRLMESSAGE	Sends a message to a control
DLGSET	Assigns a value to a control variable
DLGSETCHAR	Assigns a value to a character control variable
DLGSETCTRLEVENTHANDLER	Assigns a routine to handle an ActiveX control event
DLGSETINT	Assigns a value to an integer control variable
DLGSETLOG	Assigns a value to a logical control variable
DLGSETRETURN	Sets the return value for DLGMODAL
DLGSETSUB	Assigns a defined callback routine to a control
DLGSETTITLE	Sets the dialog title
DLGUNINIT	Deallocates memory for an initialized dialog

These routines are described in the *Language Reference* (see also `Dialog Procedures: table`).

Understanding Dialog Controls

Understanding Dialog Controls Overview

Each dialog control in a dialog box has a unique integer identifier and name. You specify the name in the Properties window for each control within the Resource Editor, and the Resource Editor assigns an integer value to each control name. You can refer to a control by its name, for example `IDC_SCROLLBAR_TEMPERATURE`, or by its integer value, which you can read from the include (.FD) file.

Each dialog control has one or more variables associated with it, called *control indexes*. These indexes can be integer, logical, character, or external. For example, a plain Button has three associated variables: one is a logical value associated with its current enabled state, one is a character variable that determines its title, and the third is an external variable that indicates the subroutine to be called if a mouse click occurs.

Dialog controls can have multiple variables of the same type. For example, the scroll bar control has four integer variables associated with it:

- Scroll bar position
- Scroll bar minimum range
- Scroll bar maximum range
- Position change if the user clicks on the scroll bar space next to the slide (big step)

See Also

- Using Control Indexes
- Available Indexes for Each Dialog Control
- Specifying Control Indexes

Using Control Indexes

The value of a dialog control's index is set with the DLGSET functions: DLGSET, DLGSETSUB, and DLGSETCHAR , DLGSETLOG, DLGSETINT. The control name and control index name are arguments to the DLGSET functions and specify the particular control index being set. For example:

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45, DLG_POSITION )
```

The index DLG_POSITION specifies the scroll bar position is set to 45. Consider the following:

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGEMAX )
```

In this statement, the index DLG_RANGEMAX specifies the scroll bar maximum range is set to 200. The DLGSET functions have the following syntax:

```
result = DLGSET (dlg, control_name, value, control_index_name)
```

The *control_index_name* determines what the *value* in the DLGSET function means.

The control index names are declared in the module IFLOGM.MOD and should not be declared in your routines. Available control indexes and how they specify the interpretation of the *value* argument are listed in the following Control Indexes table.

Control Index	How the Value is Interpreted
DLG_ADDSTRING	Used with DLGSETCHAR to add an entry to a List box or Combo box
DLG_BIGSTEP	The amount of change that occurs in a Scroll bar's or Slider's position when the user clicks beside the Scroll bar's or slider's slide (default = 10)
DLG_CHANGE	A subroutine called after the user has modified a control and the control has been updated on the screen
DLG_CLICKED	A subroutine called when the control receives a mouse-click
DLG_DBLCLICK	A subroutine called when a control is double-clicked
DLG_DEFAULT	Same as not specifying a control index
DLG_ENABLE	The enable state of the control (*value* = .TRUE. means enabled, *value* = .FALSE. means disabled)
DLG_GAINFOCUS	A subroutine called when an Edit Box receives input focus.
DLG_IDISPATCH	The object pointer of an ActiveX control
DLG_LOSEFOCUS	A subroutine called when an Edit Box loses input focus
DLG_NUMITEMS	The total number of items in a List box, Combo box, or Tab control
DLG_POSITION	The current position of the Scroll bar, Spin, Slider, or Progress bar. Also, the current cursor position in the edit box.
DLG_RANGEMIN	The minimum value of a Scroll bar's, Spin's, Slider's, or Progress' position (default = 1 for scroll bar, 0 for other controls)

Control Index	How the Value is Interpreted
DLG_RANGEMAX	The maximum value of a Scroll bar's, Spin's, Slider's, or Progress' position (default = 100)
DLG_SELCHANGE	A subroutine called when the selection in a List Box or Combo Box changes
DLG_SELCHANGING	A subroutine called when the selected Tab control is about to be changed. In this subroutine, calling `DLGGETINT` with the index DLG_STATE refers to the Tab that was active before the change.
DLG_SMALLSTEP	The amount of change that occurs in a Slider's position when the user presses the keyboard arrow keys (default = 1)
DLG_STATE	The user changeable state of a control
DLG_TEXTLENGTH	The length of text in an edit box
DLG_TICKFREQ	The interval frequency for tick marks in a Slider (default = 1)
DLG_TITLE	The title text associated with a control
DLG_UPDATE	A subroutine called after the user has modified the control state but before the control has been updated on the screen

The index names associated with dialog controls do not need to be used unless there is more than one variable of the same type for the control and you do not want the default variable. For example:

```
retlog = DlgSet(dlg, IDC_SCROLLBAR_TEMPERATURE, 45, DLG_POSITION)
retlog = DlgSet(dlg, IDC_SCROLLBAR_TEMPERATURE, 45)
```

These statements both set the Scroll bar position to 45, because `DLG_POSITION` is the default control index for the scroll bar.

Dialog Indexes

The control identifier specified in `DLGSETSUB` can also be the identifier of the dialog box. In this case, the index must be one of the values listed in the Dialog Indexes table:

Dialog Index	How the Value is Interpreted
DLG_INIT	A subroutine called after the dialog box is created but before it is displayed (with callbacktype=DLG_INIT) and immediately before the dialog box is destroyed (with callbacktype=DLG_DESTROY).
DLG_SIZECHANGE	A subroutine called after the dialog box is resized.

For more information on dialog controls, see Available Indexes for Each Dialog Control.

Available Indexes for Each Dialog Control

The available indexes and defaults for each of the controls are listed in the following table:

Dialog Controls and Their Indexes

Control Type	Integer Index Name	Logical Index Name	Character Index Name	Subroutine Index Name
ActiveX* control	DLG_IDISPATCH	DLG_ENABLE		
Button		DLG_ENABLE	DLG_TITLE	DLG_CLICKED
Check box		DLG_STATE (default) DLG_ENABLE	DLG_TITLE	DLG_CLICKED
Combo box	DLG_NUMITEMS Sets or returns the total number of items in a list	DLG_ENABLE	Use DLG_STATE, DLG_ADDSTRING, or an index: DLG_STATE By default, sets or returns the text of the selected item or first item in the list DLG_ADDSTRING Used with DLGSETCHAR to add a new item. It automatically increments DLG_NUMITEMS. An index, 1 to *n* Sets or returns the text of a particular item	DLG_SELCHANGE (default) DLG_DBLCLICK DLG_CHANGE DLG_UPDATE
Drop-down list box	Use DLG_NUMITEMS or DLG_STATE: DLG_NUMITEMS (default) Sets or returns the total number of items in a list DLG_STATE Sets or returns the index of the selected item	DLG_ENABLE	Use DLG_STATE, DLG_ADDSTRING, or an index: DLG_STATE By default, sets or returns the text of the selected item or first item in the list, or you can include an index, 1 to *n*, to set or return indicates the text of a particular item DLG_ADDSTRING Used with DLGSETCHAR to add a new item. It automatically increments DLG_NUMITEMS.	DLG_SELCHANGE (default) DLG_DBLCLICK
Edit box	DLG_TEXTLENGTH (default)	DLG_ENABLE	DLG_STATE	DLG_CHANGE (default)

Control Type	Integer Index Name	Logical Index Name	Character Index Name	Subroutine Index Name
	Sets or returns the length of the text in the edit box. DLG_POSITION Sets or returns the cursor position			DLG_UPDATE DLG_GAINFOCUS DLG_LOSEFOCUS
Group box		DLG_ENABLE	DLG_TITLE	
List box	Use DLG_NUMITEMS or an index: DLG_NUMITEMS Sets or returns the total number of items in a list An index, 1 to *n* Determines which list items have been selected and their order	DLG_ENABLE	Use DLG_STATE, DLG_ADDSTRING, or an index: DLG_STATE By default, returns the text of the first selected item DLG_ADDSTRING Used with `DLGSETCHAR` to add a new item. It automatically increments DLG_NUMITEMS. An index, 1 to *n* Sets or returns the text of a particular item	DLG_SELCHANGE (default) DLG_DBLCLICK
Picture		DLG_ENABLE		
Progress bar	DLG_POSITION (default) DLG_RANGEMIN DLG_RANGEMAX	DLG_ENABLE		
Radio button		DLG_STATE (default) DLG_ENABLE	DLG_TITLE	DLG_CLICKED
Scroll bar	DLG_POSITION (default) DLG_RANGEMIN DLG_RANGEMAX DLG_BIGSTEP	DLG_ENABLE		DLG_CHANGE
Slider	DLG_POSITION (default) DLG_RANGEMIN DLG_RANGEMAX DLG_SMALLSTEP	DLG_ENABLE		DLG_CHANGE

Control Type	Integer Index Name	Logical Index Name	Character Index Name	Subroutine Index Name
	DLG_BIGSTEP DLG_TICKFREQ			
Spin controls	DLG_POSITION (default) DLG_RANGEMIN DLG_RANGEMAX	DLG_ENABLE		DLG_CHANGE
Static text		DLG_ENABLE	DLG_TITLE	
Tab control	Use DLG_NUMITEMS (default), DLG_STATE, or an index: DLG_NUMITEMS Sets or returns the total number of tabs DLG_STATE Sets or returns the currently selected tab An index, 1 to *n* Sets or returns the dialog name of the dialog box associated with a particular tab	DLG_ENABLE	Use DLG_STATE or an index: DLG_STATE By default, sets or returns the currently selected tab An index, 1 to *n* Sets or returns the text of a particular Tab	DLG_SELCHANGE (default) DLG_SELCHANGING

For an overview on control indexes, see Using Control Indexes.

Specifying Control Indexes

Where there is only one possibility for a particular dialog control's index type (integer, logical, character, or subroutine), you do not need to specify the control index name in an argument list. For example, you can set the Static text control `IDC_TEXT_CELSIUS` to a new value with either of the following statements:

```
  retlog = DLGSETCHAR (dlg, IDC_TEXT_CELSIUS, "New Celsius Title",  &
 &                     DLG_TITLE)
  retlog = DLGSET (dlg, IDC_TEXT_CELSIUS, "New Celsius Title")
```

You do not need the control index `DLG_TITLE` because there is only one character index for a Static text control. The generic function `DLGSET` chooses the control index to change based on the argument type, in this case CHARACTER.

For each type of index, you can use the generic `DLGSET` function or the specific `DLGSET` function for that type: `DLGSETINT`, `DLGSETLOG`, or `DLGSETCHAR`.

For example, you can disable the Static text control `IDC_TEXT_CELSIUS` by setting its logical value to .FALSE. with either `DLGSET` or `DLGSETLOG`:

```
  retlog = DLGSETLOG (dlg, IDC_TEXT_CELSIUS, .FALSE., DLG_ENABLE)
  retlog = DLGSET (dlg, IDC_TEXT_CELSIUS, .FALSE., DLG_ENABLE)
```

In both these cases, the control index DLG_ENABLE can be omitted because there is only one logical control index for Static text controls.

You can query the value of a particular control index with the DLGGET functions, DLGGET, DLGGETINT, DLGGETLOG, and DLGGETCHAR. For example:

```
INTEGER current_val
LOGICAL are_you_enabled
  retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, current_val,      &
 &                 DLG_RANGEMAX)
  retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, are_you_enabled,  &
 &                 DLG_ENABLE)
```

This code returns the maximum range and the enable state of the scroll bar. The arguments you declare (current_val and are_you_enabled in the preceding example) to hold the queried values must be of the same type as the values retrieved. If you use specific DLGGET functions such as DLGGETINT or DLGGETCHAR, the control index value retrieved must be the appropriate type. For example, you cannot use DLGGETCHAR to retrieve an integer or logical value. The DLGGET functions return .FALSE. for illegal type combinations. You cannot query for the name of an external callback routine.

In general, it is better to use the generic functions DLGSET and DLGGET rather than their type-specific variations because then you do not have to worry about matching the function to the type of value set or retrieved. DLGSET and DLGGET perform the correct operation automatically, based on the type of argument you pass to them.

More information on these routines is available in the *Language Reference*.

Using Dialog Controls

Using Dialog Controls Overview

The dialog controls provided in the Resource Editor are versatile and flexible; when used together, they can provide a sophisticated user-friendly interface for your application. This section discusses the available dialog controls.

Any control can be disabled by your application at any time, so that it no longer changes or responds to the user. This is done by setting the control index DLG_ENABLE to .FALSE. using DLGSET or DLGSETLOG. For example:

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_CHECKBOX1, .FALSE., DLG_ENABLE)
```

This example disables the control named IDC_CHECKBOX1.

When you create your dialog box in the Resource Editor, the dialog controls are given a tab order. When the user hits the Tab key, the dialog box focus shifts to the next control in the tab order. By default, the tab order of the controls follows the order in which they were created. This may not be the order you want.

You can change the order by opening the Format menu and choosing Tab Order (or by pressing the key combination Ctrl+D) in the Resource Editor. A tab number will appear next to each control. Click the mouse on the control you want to be first, then on the control you want to be second in the tab order and so on. Tab order also determines which control gets the focus if the user presses the Group box hotkey. (See Using Group Boxes.)

The following sections describe the function and use of the dialog controls:

- Using Static Text
- Using Edit Boxes
- Using Group Boxes
- Using Check Boxes and Radio Buttons

- Using Buttons
- Using List Boxes and Combo Boxes
- Using Scroll Bars
- Using Pictures
- Using Progress Bars
- Using Spin Controls
- Using Sliders
- Using Tab Controls
- Setting Return Values and Exiting

For information on using ActiveX controls in a dialog, see Using ActiveX* Controls.

Using Static Text

Static text is an area in the dialog that your application writes text to. The user cannot change it. Your application can modify the Static text at any time, for instance to display a current user selection, but the user cannot modify the text. Static text is typically used to label other controls or display messages to the user.

Using Edit Boxes

An Edit box is an area that your application can write text to at anytime. However, unlike Static Text, the user can write to an Edit box by clicking the mouse in the box and typing. The following statements write to an Edit box:

```
CHARACTER(20) text /"Send text"/
retlog = DLGSET (dlg, IDC_EDITBOX1, text)
```

The next statement reads the character string in an Edit box:

```
retlog = DLGGET (dlg, IDC_EDITBOX1, text)
```

The values a user enters into the Edit box are always retrieved as character strings, and your application needs to interpret these strings as the data they represent. For example, numbers entered by the user are interpreted by your application as character strings. Likewise, numbers you write to the Edit box are sent as character strings. You can convert between numbers and strings by using internal read and write statements to make type conversions.

To read a number in the Edit box, retrieve it as a character string with `DLGGET` or `DLGGETCHAR`, and then execute an internal read using a variable of the numeric type you want (such as integer or real). For example:

```
REAL    x
LOGICAL retlog
CHARACTER(256) text
retlog = DLGGET (dlg, IDC_EDITBOX1, text)
READ (text, *) x
```

In this example, the real variable x is assigned the value that was entered into the Edit box, including any decimal fraction.

Complex and double complex values are read the same way, except that your application must separate the Edit box character string into the real part and imaginary part. You can do this with two separate Edit boxes, one for the real and one for the imaginary part, or by requiring the user to enter a separator between the two parts and parsing the string for the separator before converting. If the separator is a comma (,) you can read the string with two real edit descriptors without having to parse the string.

To write numbers to an Edit box, do an internal write to a string, then send the string to the Edit box with `DLGSET`. For example:

```
INTEGER  j
LOGICAL  retlog
CHARACTER(256) text
WRITE (text,'(I4)') j
retlog = DLGSET (dlg, IDC_EDITBOX1, text)
```

Use the DLG_TEXTLENGTH control index to get or set the length of the characters in the edit box. The length is automatically updated when:

- Your program calls `DLGSET` to set the text in the edit box (trailing blanks are stripped before setting the edit box).
- The user modifies the text in the edit box.

If you want to set the text with significant trailing blanks, call `DLGSET` to set the text followed by DLGSET with the DLG_TEXTLENGTH index to set the length that you want.

Use the DLG_POSITION index to get or set the current cursor position in the edit box. Note that setting the cursor position cancels any current selection in the edit box.

Using Group Boxes

A Group box visually organizes a collection of controls as a group. When you select Group box in Resource Editor, you create an expanding (or shrinking) box around the controls you want to group and give the group a title. You can add a hotkey to your group title with an ampersand (&). For example, consider the following group title:

```
&Temperature
```

This causes the "T" to be underlined in the title and makes it a hotkey. When the user presses the key combination ALT+T, the focus of the dialog box shifts to the next control after the Group box in the tab order. This control should be a control in the group. (You can view and change the tab order from the Format>Tab Order menu option in the Resource Editor.)

Disabling the Group box disables the hotkey, but does not disable any of the controls within the group. As a matter of style, you should generally disable the controls in a group when you disable the Group box.

Using Check Boxes and Radio Buttons

Check boxes and Radio buttons present the user with an either-or choice. A Radio button is pushed or not, and a Check box is checked or not. You use `DLGGET or DLGGETLOG` to check the state of these controls. Their state is a logical value that is .TRUE. if they are pushed or checked, and .FALSE. if they are not. For example:

```
LOGICAL pushed_state, checked_state, retlog
retlog = DLGGET (dlg, IDC_RADIOBUTTON1, pushed_state)
retlog = DLGGET (dlg, IDC_CHECKBOX1, checked_state)
```

If you need to change the state of the button, for initialization or in response to other user input, you use `DLGSET or DLGSETLOG`. For example:

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_RADIOBUTTON1, .TRUE.)
retlog = DLGSET (dlg, IDC_CHECKBOX1, .TRUE.)
```

Radio buttons are typically used in a group where the user can select only one of a set of options. When using Radio buttons with the Dialog routines, use the following guidelines:

- Each Radio button should have the "Auto" style set. This is the default for a new Radio button.

- The first Radio button in a group should have the "Group" style set. This is not the default for a new Radio button.
- The remaining Radio buttons in the group should not have the "Group" style set, and should immediately follow the first button in the dialog box "Tab order." The default tab order is the order in which you create the controls. You can view and change the tab order from the Format>Tab Order menu option in the Resource Editor.
- When the user selects a Radio button in a group, its state is set to .TRUE. and the state of the other Radio buttons in the group is set to .FALSE..
- To set the currently selected Radio button from your code, call DLGSETLOG to set the selected Radio button to .TRUE.. Do not set the other Radio buttons to .FALSE.. This is handled automatically.

Using Buttons

Unlike Check boxes and Radio buttons, Buttons do not have a state. They do not hold the value of being pushed or not pushed. When the user clicks on a Button with the mouse, the Button's callback routine is called. Thus, the purpose of a Button is to initiate an action. The external procedure you assign as a callback determines the action initiated. For example:

```
LOGICAL retlog
EXTERNAL DisplayTime
retlog = DlgSetSub( dlg, IDC_BUTTON_TIME, DisplayTime)
```

Intel Visual Fortran dialog routines do not support user-drawn Buttons.

Using List Boxes and Combo Boxes

List boxes and Combo boxes are used when the user needs to select a value from a set of many values. They are similar to a set of Radio buttons except that List boxes and Combo boxes are scrollable and can contain more items than a set of Radio buttons which are limited by the screen display area. Also, unlike Radio buttons, the number of entries in a List box or Combo box can change at run-time.

The difference between a List box and a Combo box is that a List box is simply a list of items, while a Combo box is a combination of a List box and an Edit box. A List box allows the user to choose multiple selections from the list at one time, while a Combo box allows only a single selection, but a Combo box allows the user to edit the selected value while a List box only allows the user to choose from the given list.

A Drop-down list box looks like a Combo box since it has a drop-down arrow to display the list. Like a Combo box, only one selection can be made at a time in a Drop-down list box, but, like a List box, the selected value cannot be edited. A Drop-down list box serves the same function as a List box except for the disadvantage that the user can choose only a single selection, and the advantage that it takes up less dialog screen space.

Intel Visual Fortran dialog routines do not support user-drawn List boxes or user-drawn Combo boxes. You must create List boxes and Combo boxes with the Resource Editor.

The following sections describe how to use List boxes and Combo boxes:

- Using List boxes
- Using Combo boxes
- Using Drop-down List boxes

Using List Boxes

For both List boxes and Combo boxes, the control index DLG_NUMITEMS determines how many items are in the box. Once this value is set, you set the text of List box items by specifying a character string for each item index. Indexes run from 1 to the total number of list items set with DLG_NUMITEMS. For example:

```
LOGICAL retlog
retlog = DlgSet ( dlg, IDC_LISTBOX1, 3, DLG_NUMITEMS )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Moe", 1 )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Larry", 2 )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Curly", 3 )
```

These function calls to DLGSET put three items in the List box. The initial value of each List box entry is a blank string and the value becomes nonblank after it has been set.

You can change the list length and item values at any time, including from within callback routines. If the list is shortened, the set of entries is truncated. If the list is lengthened, blank entries are added. In the preceding example, you could extend the list length and define the new item with the following:

```
retlog = DLGSET ( dlg, IDC_LISTBOX1, 4)
retlog = DLGSET ( dlg, IDC_LISTBOX1, "Shemp", 4)
```

Since List boxes allow selection of multiple entries, you need a way to determine which entries are selected. When the user selects a List box item, it is assigned an integer index. You can test which list items are selected by reading the selection indexes in order until a zero value is read. For example, if in the previous List box the user selected Moe and Curly, the List box selection indexes would have the following values:

Selection Index	Value
1	1 (for Moe)
2	3 (for Curly)
3	0 (no more selections)

If Larry alone had been selected, the List box selection index values would be:

Selection Index	Value
1	2 (for Larry)
2	0 (no more selections)

To determine the items selected, the List box values can be read with DLGGET until a zero is encountered. For example:

```
INTEGER j, num, test
INTEGER, ALLOCATABLE :: values(:)
LOGICAL retlog
retlog = DLGGET (dlg, IDC_LISTBOX1, num, DLG_NUMITEMS)
ALLOCATE (values(num))
j = 1
test = -1
DO WHILE (test .NE. 0)
  retlog = DLGGET (dlg, IDC_LISTBOX1, values(j), j)
  test = values(j)
  j = j + 1
END DO
```

In this example, j is the selection index and values(j) holds the list numbers, of the items selected by the user, if any.

To read a single selection, or the first selected item in a set, you can use `DLG_STATE`, since for a List Box `DLG_STATE` holds the character string of the first selected item (if any). For example:

```
! Get the string for the first selected item.
retlog = DLGGET (dlg, IDC_LISTBOX1, str, DLG_STATE)
```

Alternatively, you can first retrieve the list number of the selected item, and then get the string associated with that item:

```
INTEGER value
CHARACTER(256) str
 ! Get the list number of the first selected item.
 retlog = DLGGET (dlg, IDC_LISTBOX1, value, 1)
 ! Get the string for that item.
 retlog = DLGGET (dlg, IDC_LISTBOX1, str, value)
```

In these examples, if no selection has been made by the user, str will be a blank string.

In the **Properties** Window in the Resource Editor, List boxes can be specified as sorted or unsorted. The default is sorted, which causes List box items to be sorted alphabetically starting with A. If a List box is specified as sorted, the items in the list are sorted whenever they are updated on the screen. This occurs when the dialog box is first displayed and when the items are changed in a callback.

The alphabetical sorting follows the ASCII collating sequence, and uppercase letters come before lowercase letters. For example, if the List box in the example above with the list "Moe," "Larry," "Curly," and "Shemp" were sorted, before a callback or after `DLGMODAL` returned, index 1 would refer to "Curly," index 2 to "Larry," index 3 to "Moe," and index 4 to "Shemp." For this reason, when using sorted List boxes, indexes should not be counted on to be the same once the dialog is displayed and any change is made to the list items.

You can also call `DLGSETCHAR` with the DLG_ADDSTRING index to add items to a List box or Combo box. For example:

```
retlog = DlgSet(dlgtab, IDC_LIST, "Item 1", DLG_ADDSTRING)
```

When you use DLG_ADDSTRING, the DLG_NUMITEMS control index of the List or Combo box is automatically incremented.

When adding items to a sorted list or Combo box, using DLG_ADDSTRING can be much easier than the alternative (setting DLG_NUMITEMS and then setting items using an index value), because you need not worry about the list being sorted and the index values changing between calls.

Using Combo Boxes

A Combo box is a combination of a List box and an Edit box. The user can make a selection from the list that is then displayed in the Edit box part of the control, or enter text directly into the Edit box.

All dialog values a user enters are character strings, and your application must interpret these strings as the data they represent. For example, numbers entered by the user are returned to your application as character strings.

Because user input can be given in two ways, selection from the List box portion or typing into the Edit box portion directly, you need to register two callback types with `DLGSETSUB` for a Combo box. These callback types are `dlg_selchange` to handle a new list selection by the user, and `dlg_update` to handle text entered by the user directly into the Edit box portion. For example:

```
retlog = DlgSetSub( dlg, IDC_COMBO1, UpdateCombo, dlg_selchange )
retlog = DlgSetSub( dlg, IDC_COMBO1, UpdateCombo, dlg_update )
```

A Combo box list is created the same way a List box list is created, as described in the previous section, but the user can select only one item from a Combo box at a time. When the user selects an item from the list, Windows automatically puts the item into the Edit box portion of the Combo box. Thus, there is no need, and no mechanism, to retrieve the item list number of a selected item.

If the user is typing an entry directly into the Edit box part of the Combo box, again Windows automatically displays it and you do not need to. You can retrieve the character string of the selected item or Edit box entry with the following statement:

```
! Returns the character string of the selected item or Edit box entry as str.  retlog = DLGGET (dlg,
IDC_COMBO1, str)
```

Like List boxes, Combo boxes can be specified as sorted or unsorted. The notes about sorted List boxes also apply to sorted Combo boxes.

You have three choices for Combo box Type option in the Properties Window:

- Simple
- Drop list
- Drop-down

Simple and Drop-down are the same, except that a simple Combo box always displays the Combo box choices in a list, while a Drop-down list Combo box has a Drop-down button and displays the choices in a Drop-down list, conserving screen space. The Drop list type is halfway between a Combo box and a List box and is described below.

Using Drop-Down List Boxes

To create a Drop-down list box, do the following:

1. Choose a Combo box from the control toolbar and place it in your dialog.
2. Select the **Combo box**, then select **View>Properties** to open the Properties Window.
3. Choose Drop List as the control type.

A Drop-down list box has a drop-down arrow to display the list. Like a Combo box, only one selection can be made at a time in the list, but like a List Box, the selected value cannot be edited. A Drop-down list box serves the same function as a List box except for the disadvantage that the user can choose only a single selection, and the advantage that it takes up less dialog screen space.

A Drop-down list box has the same control indexes as a Combo box with the addition of another INTEGER index to set or return the list number of the item selected in the list. For example:

```
INTEGER num
 ! Returns index of the selected item.
 retlog = DLGGET (dlg, IDC_DROPDOWN1, num, DLG_STATE)
```

Using Scroll Bars

With a Scroll bar, the user determines input by manipulating the slide up and down or right and left. Your application sets the range for the Scroll bar, and thus can interpret a position of the slide as a number. If you want to display this number to the user, you need to send the number (as a character string) to a Static text or Edit Box control.

You set the lower and upper limits of the Scroll bar range by setting the control index `DLG_RANGEMIN` and `DLG_RANGEMAX` with `DLGSET` or `DLGSETINT`. The default values are 1 and 100. For example:

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 212, DLG_RANGEMAX)
```

You get the slide position by retrieving the control index `DLG_POSITION` with `DLGGET` or `DLGGETINT`. For example:

```
INTEGER slide_position
retlog = DLGGET (dlg, IDC_SCROLLBAR1, slide_position, DLG_POSITION)
```

You can also set the increment taken when the user clicks in the blank area above or below the slide in a vertical Scroll bar, or to the left or right of the slide in a horizontal Scroll bar, by setting the control index `DLG_BIGSTEP`. For example:

```
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 20, DLG_BIGSTEP)
```

When the user clicks on the arrow buttons of the Scroll bar, the position is always incremented or decremented by 1.

The maximum value (DLG_POSITION) that a scroll bar can report (that is, the maximum scrolling position) depends on the page size (DLG_BIGSTEP). If the scroll bar has a page size greater than one, the maximum scrolling position is less than the maximum range value (DLG_RANGEMAX or DLG_RANGE). You can use the following formula to calculate the maximum scrolling position:

```
MaxScrollPos = MaxRangeValue - (PageSize - 1)
```

For example, if a scroll bar has DLG_RANGEMAX = 100 (100 is the default value of DLG_RANGEMAX) and DLG_BIGSTEP = 10 (10 is the default value of DLG_BIGSTEP), then the maximum DLG_POSITION is 91 (100 - (10 - 1)).

This allows your application to implement a "proportional" scroll bar. The size of the scroll box (or thumb) is determined by the value of DLG_BIGSTEP and should represent a "page" of data (that is, the amount of data visible in the window).

When the user clicks in the "shaft" of the scroll bar, the next (or previous) page is displayed. The top of the thumb (for a vertical scroll bar) or the left edge of the thumb (for a horizontal scroll bar) represents the position of the scroll bar (DLG_POSITION).

The size of the thumb represents the amount of data currently visible. There is a minimum thumb size so as not to affect usability. When the scroll bar is at its maximum position, the position will represent the position in the data such that the last "page" of data is visible in the window. When the top (or left edge) of the scroll bar is at the mid-point of the shaft, DLG_POSITION will be the mid-point of the range and the mid-point of the data should be displayed at the top of the window.

Using Pictures

The Picture control is an area of your dialog box in which your application displays a picture.

The user cannot change it, since it is an output-only window. It does not respond to user input and therefore does not support any callbacks.

The picture displayed can be set using the Properties dialog box in the Resource Editor. The options that can be fully defined using the Resource Editor include a:

- Icon
- Bitmap
- Frame
- Rectangle

When using the Resource editor, you need to click on the border of the picture to select it, then select **View>Properties** to display the Properties window. The **Properties Window** allows you to select the type of picture as well as specify the image for certain picture types.

For example, to add an existing BMP (Bitmap) file to a dialog:

1. Open your project, double-click on the .rc file, and select **Edit>Add Resource**.
2. Select the Resource type (such as Icon or Bitmap) and then select whether it is a new resource, an Import (existing) resource, or Custom resource. In our example, we will specify an existing Bitmap file we have previously copied into our project directory, so will click Import.
3. After we specify the resource, its name appears in the **Resource Editor** under the appropriate resource type (such as bitmap).
4. Add a picture to the dialog box, by dragging the picture icon from the Control toolbar to the appropriate place in the dialog box. You can resize the picture as needed. The default picture type is frame.
5. Carefully click on the border of the picture area to display, then select **View>Properties**.
6. In the **Properties** Window, select the type of picture. If you select a type of Bitmap or Icon, for example, you can select the image from the list of available project resources using the Image property.
7. Move the picture as needed.

Using Progress Bars

The Progress bar is a window that can be used to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled as an operation progresses.

Your application sets the range of the Progress bar, using `DLG_RANGEMIN` and `DLG_RANGEMAX`, and the current position, using `DLG_POSITION`. Both the minimum and maximum range values must be between 0 and 65535.

A Progress bar is an output-only window. It does not respond to user input and therefore does not support any callbacks.

Using Spin Controls

The Spin control contains up and down arrows that allow the user to step through values. Your application sets or gets the range of the Spin control's values, using `DLG_RANGEMIN` and `DLG_RANGEMAX`, and the current value, using `DLG_POSITION`.

The Spin control is usually associated with a companion control that is called a "buddy window." To the user, the Spin control and its buddy window often look like a single control. You can specify that the Spin control automatically position itself next to its buddy window and that it automatically set the title of its buddy window to its current value. This is accomplished by setting the "Auto buddy" and "Set buddy integer" properties on the Spin control.

The buddy window is usually an Edit Box or Static Text control. When the "Auto buddy" style is set, the Spin control automatically uses the previous control in the dialog box tab order as its buddy window.

The Spin Control calls the `DLG_CHANGE` callback whenever the user changes the current value of the control.

The Spin control is named the "Up-down" control in Windows programming documentation.

Using Sliders

The Slider Control is a window that contains a slider and optional tick marks. Your application sets or gets the range of the Slider control's values, using `DLG_RANGEMIN` and `DLG_RANGEMAX`, and the current value, using `DLG_POSITION`. Your application can also set:

- The number of logical positions the slider moves in response to keyboard input from the arrow keys using `DLG_SMALLSTEP`.
- The number of logical positions the slider moves in response to keyboard input, such as the PAGE UP or PAGE DOWN keys, or mouse input, such as clicks in the slider's channel, using `DLG_BIGSTEP`.
- The interval frequency for tick marks on the slider using `DLG_TICKFREQ`.

The Slider Control calls the `DLG_CHANGE` callback whenever the user changes the current value of the control.

The Slider control is named the "Trackbar" control in Windows programming documentation.

Using Tab Controls

The Tab control is like the dividers in a notebook or the labels on a file cabinet. By using a Tab control, an application can define multiple pages for the same area of a dialog box. Each page is associated with a particular Tab and only one page is displayed at a time.

The control index `DLG_NUMITEMS` determines how many Tabs are contained in the Tab control. For each Tab, you specify the label of the Tab using `DLGSETCHAR` and an index value from 1 to the number of Tabs set with `DLG_NUMITEMS`. Each Tab has an associated dialog box that is displayed when the Tab is selected. You specify the dialog box using `DLGSETINT` with the dialog name and an index value corresponding to the the Tab. For example, the code below defines three Tabs in a Tab control. The Tab with the label "Family" is associated with the dialog box named IDD_TAB_DIALOG1, and so on.

```
!  Set initial Tabs
lret = DlgSet(gdlg, IDC_TAB, 3)
lret = DlgSet(gdlg, IDC_TAB, "Family", 1)
lret = DlgSet(gdlg, IDC_TAB, "Style", 2)
lret = DlgSet(gdlg, IDC_TAB, "Size", 3)
lret = DlgSet(gdlg, IDC_TAB, IDD_TAB_DIALOG1, 1)
lret = DlgSet(gdlg, IDC_TAB, IDD_TAB_DIALOG2, 2)
lret = DlgSet(gdlg, IDC_TAB, IDD_TAB_DIALOG3, 3)
```

You define each of the Tab dialogs using the resource editor just as you do for the dialog box that contains the Tab control. In the Properties Window, you must make the following style settings for each Tab dialog:

1. Set the "Style" to "Child"
2. Set "Border" to "None"
3. Set "Title Bar" to "False."

Before displaying the dialog box that contains the Tab control (using `DLGMODAL` or `DLGMODELESS`):

1. Call `DLGSETSUB` to define a DLG_INIT callback for the dialog box

2. Call DLGINIT for each Tab dialog

In the DLG_INIT callback of the dialog box that contains the Tab control, if the callbacktype is DLG_INIT, call DLGMODELESS for each of the Tab dialog boxes. Specify SW_HIDE as the second parameter, and the window handle of the Tab control as the third parameter. After calling DLGMODELESS, call DLGSET with the DLG_STATE index to set the initial Tab. For example:

```
  ! When the Main dialog box is first displayed, call DlgModeless to
  ! display the Tab dialog boxes. Note the use of SW_HIDE. The
  ! Dialog Functions will "show" the proper Tab dialog box.
  if (callbacktype == dlg_init) then
hwnd = GetDlgItem(dlg % hwnd, IDC_TAB)
lret = DlgModeless(gdlg_tab1, SW_HIDE, hwnd)
lret = DlgModeless(gdlg_tab2, SW_HIDE, hwnd)
lret = DlgModeless(gdlg_tab3, SW_HIDE, hwnd)
 ! Note that we must set the default Tab after the calls to
 ! DlgModeless. Otherwise, no Tab dialog box will be displayed
 ! initially.
lret = DlgSet(dlg, IDC_TAB, 1, dlg_state)
```

Call DLGUNINIT for each Tab dialog when you are done with it.

Setting Return Values and Exiting

When the user selects the dialog's OK or CANCEL button, your dialog procedure is exited and the dialog box is closed. DLGMODAL returns the control name (associated with an integer identifier in your include (.FD) file) of the control that caused it to exit; for example, IDOK or IDCANCEL.

If you want to exit your dialog box on a condition other than the user selecting the OK or CANCEL button, you need to include a call to the dialog subroutine DLGEXIT from within your callback routine. For example:

```
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE IFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
...
  CALL DLGEXIT (dlg)
```

The only argument for DLGEXIT is the dialog derived type. The dialog box is exited after DLGEXIT returns control back to the dialog manager, not immediately after calling DLGEXIT. That is, if there are other statements following DLGEXIT within the callback routine that contains it, those statements are executed and the callback routine returns before the dialog box is exited.

If you want DLGMODAL to return with a value other than the control name of the control that caused the exit, (or -1 if DLGMODAL fails to open the dialog box), you can specify your own return value with the subroutine DLGSETRETURN. For example:

```
TYPE (DIALOG) dlg
INTEGER altreturn
...
altreturn = 485
CALL DLGSETRETURN (dlg, altreturn)
CALL DLGEXIT(dlg)
```

To avoid confusion with the default failure condition, use return values other than -1.

It is not possible to return a value when a modeless dialog box exits. However, you can call DLGSETSUB to set the DLG_INIT callback routine to have a procedure called immediately before the dialog box is destroyed.

If you want the user to be able to close the dialog from the system menu or by pressing the ESC key, you need a control that has the ID of `IDCANCEL`. When a system escape or close is performed, it simulates pressing the dialog button with the ID `IDCANCEL`. If no control in the dialog has the ID `IDCANCEL`, then the close command will be ignored (and the dialog can not be closed in this way).

If you want to enable system close or ESC to close a dialog, but do not want a cancel button, you can add a button with the ID `IDCANCEL` to your dialog and then remove the visible property in the button's Properties Window . Pressing ESC will then activate the default click callback of the cancel button and close the dialog.

Using ActiveX* Controls

Using ActiveX* Controls Overview

The dialog routines support the use of ActiveX* controls. This means that the dialog routines can act as an ActiveX control container.

Using an ActiveX control in a dialog box requires four steps discussed in the following sections:

1. Using the Resource Editor to Insert an ActiveX Control discusses using the resource editor to insert the ActiveX control into the dialog box.
2. Using the Intel® Fortran Module Wizard to Generate a Module discusses using the Intel® Fortran Module Wizard to generate a module that allows you to use the methods, properties, and events of the ActiveX control from Fortran.
3. Adding Code to Your Application discusses adding code to your application to manipulate the ActiveX control and respond to control events.
4. Registering an ActiveX Control describes how to ensure that the ActiveX control and the dialog procedure ActiveX container DLL are present and registered on the target system.

Using the Resource Editor to Insert an ActiveX Control

To add ActiveX controls to a dialog box:

- In the Dialog Editor window, right click and hold to display the pop-up (shortcut) menu. Input focus must be in the dialog box or in the window.

NOTE. To display the Dialog Editor window, follow the steps in Designing a Dialog Box Overview.

- Select Insert ActiveX Control... in the shortcut (right-click) menu. A dialog box displays the list of ActiveX controls that are registered on your system.
- Select one of the listed controls and click OK to add it to your dialog box.

Once you insert the ActiveX control, you can modify it, just like other controls using the resource editor. You can move, resize, and delete the ActiveX control. You can also modify its properties. ActiveX controls often support a large set of properties.

Using the Intel® Fortran Module Wizard to Generate a Module

An ActiveX control is an Automation object. ActiveX controls typically support methods, properties, and events. ActiveX controls use events to notify an application that something has happened to the control. Common examples of events include clicks on the control, data entered using the keyboard, and changes in the control's state. When these actions occur, the control issues an event to alert the application.

The application, in return, uses methods and properties to communicate with the control. Methods are functions that perform an action on the ActiveX control. For example, you would use a method to tell an Internet Explorer ActiveX control to load a particular URL. Properties hold information about the state of an object, for example, the font being used by a control to draw text.

The Intel® Fortran Module Wizard generates Fortran 90 modules that simplify calling COM and Automation objects from Fortran programs.

To run the wizard:

1. Select Tools>Intel ® Fortran Module Wizard in the main menu bar. Select the component of interest in the .COM tab. Select Generate code that uses Automation interfaces and click Next.
2. Select individual components of the ActiveX control to process using the module wizard or click Select All to process all components.
3. Select either of the code generation options as desired.
4. Enter a different module name than the one displayed, if desired.
5. Click Finish.

The module wizard now asks you for the name of the source file to be generated. Supply the name and make sure the Add file to Project checkbox is selected. COM will now open the type library and generate a file containing Fortran modules.

Adding Code to Your Application

The structure of your application remains the same as when using a dialog box that does not contain an ActiveX control. See Writing a Dialog Application for details. This section discusses programming specific to ActiveX controls.

Your application must call `COMINITIALIZE` before calling `DLGINIT` with a dialog box that contains an ActiveX control. Your application must include the statement `USE IFCOM` to access COMINITIALIZE. Your application must call `COMUNINITIALIZE` when you are done using ActiveX controls, but not before calling `DLGUNINIT` for the dialog box that contains the ActiveX control.

You can call the methods of an ActiveX control and set and retrieve its property values using the interfaces generated by the Intel Fortran Module Wizard or by using the IFAUTO routines. To do this, you must have the object's IDispatch interface pointer. Use the `DLGGET` function with the ActiveX control's name, the DLG_IDISPATCH control index, and an integer variable to receive the IDispatch pointer. For example:

```
retlog = DlgGet( dlg, IDC_ACTIVEX, idispatch, DLG_IDISPATCH )
```

You do not need to specify the index DLG_IDISPATCH because it is the default integer index for an ActiveX control.

However, the control's IDispatch pointer is not available until after the control has been created and is only valid until the dialog box is closed. The control is created during the call to `DLGMODALDLGMODELESS` or . If you call `DLGGET` to retrieve the IDispatch pointer before calling `DLGMODAL` or `DLGMODELESS`, the value returned will be 0.

Do not call COMRELEASEOBJECT with the iDispatch pointer returned by DLGGET. The dialog procedures use a reference counting optimization since the lifetime of the control is guaranteed to be less than the lifetime of the dialog box.

If you want to use a method or property of a control before the dialog box is displayed to your application's user, you can use a DLG_INIT callback. Call DLGSETSUB using the dialog box name and the DLG_INIT index to define the callback. For example:

```
retlog = DlgSetSub( dlg, IDD_DIALOG, DlgSub, DLG_INIT )
```

The **DLG_INIT** callback is called after the dialog box is created but before it is displayed (with callbacktype=DLG_INIT) and immediately before the dialog box is destroyed (with callbacktype=DLG_DESTROY). The **DLG_INIT** callback is the soonest that the control's IDispatch pointer is available. The **DLG_DESTROY** callback is the latest that this pointer is valid. After the **DLG_DESTROY** callback, the ActiveX control is destroyed.

The following example shows using a DLG_INIT callback to reset the state of a control property before it is destroyed:

```
SUBROUTINE mmplayerSub( dlg, id, callbacktype )
 !DEC$ ATTRIBUTES DEFAULT :: mmplayerSub
 use iflogm
 use ifcom
 use ifauto
 implicit none
 type (dialog) dlg
 integer id, callbacktype
 include 'resource.fd'
 integer obj, iret
 logical lret
 if (callbacktype == dlg_init) then
     lret = DlgGet(dlg, IDC_ACTIVEMOVIECONTROL1, obj)
     !  Add any method or property calls here before the
     !  dialog box is displayed
 else if (callbacktype == dlg_destroy) then
     !  Reset the filename to "" to "close" the current file
     lret = DlgGet(dlg, IDC_ACTIVEMOVIECONTROL1, obj)
     iret = AUTOSETPROPERTY(obj, "FileName", "")
 endif
END SUBROUTINE mmplayerSub
```

The module generated by the Fortran Module Wizard for an ActiveX control contains a number of sections:

- ! CLSIDs

  Parameters of derived type GUID which identify the ActiveX control class. Your application typically doesn't need to use this parameter.

- ! IIDs

  Parameters of derived type GUID which identify source (event) interfaces of the ActiveX control. Your application can use these values in calls to DLGSETCTRLEVENTHANDLER (see below).

- ! Enums

  Parameters of type integer that identify constants used in the ActiveX control's interfaces.

- ! Interfaces

  Interfaces for the source (event) interfaces that are defined by the ActiveX control. There may be 0, 1, or more source interfaces implemented by the control. A control does not have to support events.

- ! Module Procedures

  Wrapper routines that make it easy to call the control's methods and get or retrieve the control's properties.

See the *Language Reference* for more information on using the method and property interfaces generated by the Intel Fortran Module Wizard.

In addition to methods and properties, ActiveX controls also define events to notify your application that something has happened to the control. The dialog procedures provide a routine, `DLGSETCTRLEVENTHANDLER`, that allows you to define a routine to be executed when an event occurs.

The `DLGSETCTRLEVENTHANDLER` function has the following interface:

```
integer DlgSetCtrlEventHandler( dlg, controlid, handler, dispid, iid )
```

The arguments are as follows:

`dlg`	(Input) Derived type DIALOG. Contains dialog box parameters.
`controlid`	(Input) Integer. Specifies the identifier of a control within the dialog box (from the .FD file).
`handler`	(Input) EXTERNAL. Name of the routine to be called when the event occurs.
`dispid`	(Input) Integer. Specifies the member id of the method in the event interface that identifies the event
`iid`	(Input, Optional) Derived type (GUID). Specifies the Interface identifier of the source (event) interface. If not supplied, the default source interface of the ActiveX control is used.

Consider the following function call:

```
ret = DlgSetCtrlEventHandler( dlg, IDC_ACTIVEMOVIECONTROL1, &
    ActiveMovie_ReadyStateChange, -609, IID_DActiveMovieEvents2 )
```

In this function call:

- `IDC_ACTIVEMOVIECONTROL1` identifies an ActiveMovie control in the dialog box.
- `ActiveMovie_ReadyStateChange` is the name of the event handling routine.
- `-609` is the member id of the ActiveMovie's control ReadyStateChange event. You can get this number from:
  - The module that the Fortran Module Wizard generated. There is a "MEMBERID = nn" comment generated for each method in a source interface (see the example below).
  - The documentation of the ActiveX control.
  - A tool that allows you to examine the type information of the ActiveX control, for example, the OLE/COM Object Viewer in the Microsoft Visual Studio* IDE.
- `IID_DActiveMovieEvents2` is the identifier of the source (event) interface.

The interface generated by the Intel Fortran Module Wizard for the ReadyStateChange event follows:

```
INTERFACE
  !Reports that the ReadyState property of the ActiveMovie Control
  !has changed
  ! MEMBERID = -609
  SUBROUTINE DActiveMovieEvents2_ReadyStateChange($OBJECT, ReadyState)
      INTEGER(4), INTENT(IN) :: $OBJECT  ! Object Pointer
      !DEC$ ATTRIBUTES VALUE :: $OBJECT
      INTEGER(4) :: ReadyState
      !DEC$ ATTRIBUTES VALUE :: ReadyState
      !DEC$ ATTRIBUTES STDCALL :: DActiveMovieEvents2_ReadyStateChange
   END SUBROUTINE DActiveMovieEvents2_ReadyStateChange
 END INTERFACE
```

The handler that you define in your application must have the same interface. Otherwise, your application will likely crash in unexpected ways because of the application's stack getting corrupted.

Note that an object is always the first parameter in an event handler. This object value is a pointer to the control's source (event) interface, not the IDispatch pointer of the control. You can use `DLGGET` as described above to retrieve the control's IDispatch pointer.

Registering an ActiveX Control

Any ActiveX control that you use must be properly installed and registered on your machine and any machine that you distribute your application to. See the documentation for the ActiveX control for information on how to redistribute it.

The dialog routine ActiveX control container support is implemented in the files `IFDLG`*nnn*`.DLL`. This DLL must be present and registered on any machine that will run your application.

To register a DLL, use REGSVR32.EXE, located in the Windows system directory. REGSVR32 takes a single argument: the path of the DLL.

Index

.FD dialog file 84
.RC files
    dialog 84
/dll compiler option 20
/libs
    qwin compiler option 61

A

About box
    routine defining 71
ABOUTBOXQQ
    using 71
accessing window properties 30
activating a graphics mode 53
activating the dialog box 85
ActiveX* controls
    inserting 113
    programming practices 114
    properties, methods, and events 114
    system requirements 117
    using (Fortran) 113
adding color 49
adding controls to dialogs 78
adding shapes 56
APPENDMENUQQ
    using to append menu items 67
Application Wizard 89, 91
applications
    creating Windows* 9
    sample windows 15
AppWizard, see Application Wizard 9
ATTRIBUTES
    DLLIMPORT option 18
available indexes for dialog controls 98
available typefaces 61

B

bitmap images 65
blocking procedures
    effect in QuickWin 75
building dynamic-link library projects 16, 20
building excutables that use DLLs 21
buttons 105

C

C data types
    OpenGL translated to Fortran types 47
callback routines 72, 75, 86
    effect on mouse events 72
callback routines *(continued)*
    example of dialog 86
changing status bar and state messages 70
character-based text
    displaying in QuickWin 59
character-font routines
    using in QuickWin 35
characters
    displaying font-based in QuickWin 60
    routine to check input of 76
check boxes 104
checking the current graphics mode 36
child window
    creating 32
    keeping open 34
CLEARSCREEN
    example of 43
CLICKMENUQQ
    using to simulate menu selections 67
clip region 39
closing dialogs 112
color
    adding to graphics 49
    mixing 49
    QuickWin routines for 37
    text 52
color indexes 49, 52
    setting and getting 52
color mixing 49
color values
    setting and getting 52
combo boxes 105
common blocks
    in Fortran DLLs 18
comparing QuickWin with Windows-based applications 25
control indexes 97, 101
    specifying 101
controlling
    the size and position of windows 35
conventions
    in the documentation 7
coordinate graphics
    sample program 43
coordinate systems
    understanding 38
coordinates
    setting in QuickWin 42
    text 38
creating
    a menu list of available child windows 67
    child windows 32
    Fortran DLLs 16
    QuickWin windows 29

custom icons
  using in QuickWin 71
customizing QuickWin applications 66

D

data types
  OpenGL C translated to Fortran 47
default quickwin menus 28
defining an About box 71
DELETEMENUQQ
  using to delete menu items 67
deleting menu items 67
dialog applications
  using Windows APIs 9
  writing 84
dialog boxes 77, 78, 85, 96
  adding controls 78
  controls in 96
  designing for Fortran applications 78
  initializing and activating 85
  saving as a resource file 78
dialog callback routines 86
dialog controls 82, 96, 97, 98, 102, 104
  grouping 104
  indexes 97, 98
  setting properties of 82
  using 102
Dialog Editor
  opening 78
  using to insert an ActiveX* control 113
dialog routines 95
dialogs 77, 78, 82, 84, 86, 88, 89, 91, 93, 95, 97, 103, 104, 105, 109, 110, 111, 112, 113
  buttons in 105
  callback routines 86
  check boxes in 104
  combo boxes in 105
  control index in 97
  control properties 82
  designing for Fortran applications 78
  edit boxes in 103
  exiting 112
  group boxes in 104
  include files 84
  list boxes in 105
  modeless 88
  pictures in 109
  progress bars in 110
  routines 95
  scroll bars in 109
  setting return values 112
  sliders in 111
  spin controls in 110
  static text in 103
  tab controls in 111
  using ActiveX* controls (Fortran) 113
  using Fortran AppWizards 89
  using Fortran Windows Project AppWizards 91
  using in a DLL 93
  writing application using 84
display options
  selecting in QuickWin 36
displaying character-based text 59
displaying font-based characters 60
displaying graphics output 57
displaying message boxes 71
DLG_ index names 97, 98
DLGEXIT
  using 112
DLGGET
  example of 101
  using to check the state of dialog controls 104
  using with Edit boxes 103
  using with scroll bars 109
DLGGETCHAR
  using with Edit boxes 103
DLGGETINT
  using with scroll bars 109
DLGGETLOG
  using to check the state of controls 104
DLGINIT
  using to initialize dialog box 85
DLGISDLGMESSAGE
  using with a modeless dialog box 88
DLGMODAL
  using to indicate a dialog type 85
  using to return a dialog control name 112
  using with callback routines 86
DLGMODELESS
  using to display a modeless dialog box 88
DLGSET
  using to change the button state 104
  using to disable dialog controls 102
  using to set the control index 101
  using to set the scroll bar range 109
  using to specify value for dialog control index 97
  using to write to an Edit box 103
DLGSETINT
  using to set the scroll bar range 109
DLGSETLOG
  using to change the button state 104
  using to disable dialog controls 102
DLGSETRETURN
  using to specify a return value 112
DLGUNINIT
  using to free resources 85
DLL
  See dynamic-link libraries (DLLs) 16
DLLEXPORT
  option for ATTRIBUTES directive 16, 18
  using for common blocks 18
  using in modules 18
DLLIMPORT
  option for ATTRIBUTES directive 16, 18
documentation
  notational conventions 7
documentation, related 8
drawing a sine curve 55
drawing graphics 52, 57
drawing graphs 43
drawing lines on the screen 54

dynamic-link libraries (DLLs) 16, 18, 20, 21, 93
  and executables 21
  behavior of 16, 20
  building 20
  building executables using 21
  checking the export table 20
  creating 16
  organization of 20
  overview of 16
  sharing data 18
  using dialogs in 93

E

Edit boxes 103
editing graphics
  in QuickWin 65
editing screen images from the QuickWin edit menu 64
editing text
  in QuickWin 65
editing text and graphics from the QuickWin Edit menu 65
enhancing QuickWin applications 66
executables using DLLs 21

F

figure properties
  setting in QuickWin 37
fill mask
  QuickWin routines for 37
FLOODFILL
  using for figure properties 37
FLOODFILLRGB
  using for figure properties 37
focus
  using QuickWin to give 33
FOCUSQQ
  using to set focus 33
font
  definition of 61
font-based characters
  displaying in QuickWin 60
fonts
  example program 63
  initializing 62
  setting 62
  using from graphics library 61
Fortran Windowing applications
  coding requirements 9
  creating 9
  using menus 13
frame window
  controlling 67
functions
  exported to other applications 16
  imported from a DLL 16

G

GDI (Graphic Device Interface) program
  writing 9
GETACTIVEQQ
  using with active child window 33
GETBKCOLOR
  using for figure properties 37
  using for text colors 52
GETBKCOLORRGB
  using for figure properties 37
  using for text colors 52
GETCOLOR
  using for figure properties 37
GETCOLORRGB
  using for color mixing 49
  using for figure properties 37
GETCURRENTPOSITION
  using to locate graphics output position 42
GETCURRENTPOSITION_W
  using to locate graphics output position 42
GETFILLMASK
  using for figure properties 37
GETFONTINFO
  example of 62
GETGTEXTROTATION
  overview of graphic display routines 36
GETHWNDQQ
  overview of windows focus routines 33
  using with WIndows API routines 25
GETIMAGE
  using to transfer images in memory 64
GETIMAGE_W
  using to transfer images in memory 64
GETLINESTYLE
  using for figure properties 37
GETPHYSCOORD
  overview of graphics coordinate routines 42
GETPIXELRGB
  using to return color 49
GETPIXELSRGB
  using to return color 49
GETTEXTCOLOR
  using to get color values 52
GETTEXTCOLORRGB
  using to get color values 52
GETTEXTWINDOW
  using to get text window boundaries 59
GETVIEWCOORD
  overview of graphics coordinate routines 42
GETVIEWCOORD_W
  overview of graphics coordinate routines 42
GETWINDOWCONFIG
  using to get child window settings 36
  using to get virtual window properties 30
  using to get windows coordinates 42
GETWINDOWCOORD
  overview of graphics coordinate routines 42
GETWRITEMODE
  using to get logical write mode 37

GETWSIZEQQ
    using to get size or position of window 35
giving a window focus 33
Graphic Device Interface (GDI) calls 9
Graphical User Interface (GUI) 9
graphics
    adding color to 49
    adding shapes to 56
    coordinate systems in 38
    displaying in QuickWin 57
    drawing in QuickWin 57
    drawing lines in 54
    drawing sine curves 55
    editing in QuickWin 65
    mixing colors 49
    OpenGL 47
    physical coordinates in 39
    QuickWin routines that draw 57
    QuickWin routines to display 36
    setting highest possible resolution 36
    setting the mode of 36
    text colors in 52
    text coordinates in 38
    using fonts from library 61
    VGA color palette 51
    viewport coordinates in 39
    window coordinates in 39
    writing programs 52
graphics applications
    QuickWin 27
graphics coordinates
    setting in QuickWin 42
graphics fonts
    available typefaces 61
    example program 63
    initializing 62
    setting and displaying 62
graphics mode
    activating 53
    checking the current 36
group boxes
    using 104
GRSTATUS
    using to fix font problems 62

I

I/O
    simulating nonblocking 76
icons
    using custom in QuickWin 71
IFOPNGL library module 47
images
    loading and saving in QuickWin 65
    transferring in memory 64
    working with screen in QuickWin 64
IMAGESIZE
    using to transfer images in memory 64
IMAGESIZE_W
    using to transfer images in memory 64
import library (.lib)
    for DLL 16, 20
importing and exporting data with dlls 16, 20
Include (.FD) file
    for dialog boxes 84
initial menu
    controlling 67
INITIALIZEFONTS
    example of 62
initializing fonts 62
initializing the dialog box 85
INITIALSETTINGS
    using to define initial settings of window 67
INQFOCUSQQ
    example of 33
inserting menu items 67
Intel® Fortran
    creating Fortran Windowing applications 9
    creating QuickWin applications 23
    using dialogs 77
Intel® Fortran Module Wizard
    using to generate modules 114
IOFOCUS
    specifier for OPEN 33

K

keeping child windows open 34

L

labels
    platform 7
language extensions
    notational conventions 7
line style
    QuickWin routines for 37
LINETO
    example of 54
    in physical coordinates 39
LINETOAR
    example of 54
LINETOAREX
    example of 54
list boxes 105
LOADIMAGE
    using to transfer images 65
LOADIMAGE_W
    using to transfer images 65
loading and saving images to files 65

M

MDI Fortran Windowing applications 9
MDI menu bar 28
menu items
    use with Fortran Windowing applications 13
    use with Fortran Windows applications 13

menus
default QuickWin 28
modifying in QuickWin 67
message box
displaying in QuickWin 71
MESSAGEBOXQQ
example of 71
MIDI Mapper 9
modal dialogs 77, 89
using Fortran project AppWizards 89
modeless dialogs 77, 91
using Fortran project AppWizards 91
modeless dialogs (Fortran) 88
MODIFYMENUFLAGSQQ
using to modify menu items 67
MODIFYMENUROUTINEQQ
using to modify a menu callback routine 67
MODIFYMENUSTRINGQQ
using to customize text 59
using to modify a menu state 67
modules
DLLEXPORT and DLLIMPORT in 18
in Fortran DLLs 18
mouse
using in QuickWin 72
mouse events
effect of callback routines on 72
in QuickWin 72
MOVETO
example of 54, 63
overview of library routines 61
MSFWIN$ prefix for graphics routines 24
multithread applications
simulating nonblocking I/O 76

N

non-blocking I/O 76

O

OPEN
FILE specifier
in QuickWin 32
OpenGL applications 47
OpenGL graphics 47
OpenGL library 47
OUTGTEXT
overview of setting display options 36
overview of setting figure properties 37
OUTTEXT
and text color routines 52
overview of character display routines 59

P

PEEKCHARQQ
simulating in QuickWin 76
pictures
using 109
program control of menus 67
programs
Quickwin and Windows* 25
Windows* GDI 9
Windows* GUI 9
writing graphics 52
progress bars 110
projects
building dynamic-link library 16, 20

Q

QuickWin
about boxes in 71
callback routines in 75
capabilities of 23
changing messages 70
character-based text in 59
child windows in 32, 34
coding guidelines 29
compared to Windows-based applications 25
controlling menus in 67
creating windows 29
custom icons in 71
customizing 66
customizing applications 66
default menus 28
displaying graphics using 57
displaying message boxes 71
drawing graphics in 57
editing text and graphics 65
enhancing applications 66
font-based characters 60
giving focus to a window 33
loading and saving images 65
overview 23
position of windows 35
precautions when programming 75
restrictions to 25
screen images in 64
selecting display options in 36
setting active window in 33
setting figure properties in 37
setting graphics coordinates in 42
size of windows 35
standard graphics applications in 26
transferring images in memory 64
user interface 27
using character-font routines in 35
using graphics routines in 35
using mouse in 72
window properties 30
QuickWin functions 23
QuickWin graphics applications
characteristics of 27
example 27
how to build 25
QuickWin graphics library 23
QuickWin procedures 23

QuickWin programming precautions 75
QuickWin programs
  overview 23
QuickWin routines
  naming conventions 24
QuickWin subroutines 23

R

real-coordinate graphics
  sample program 43
RECTANGLE
  example of 54
RECTANGLE_W
  using for windows coordinates with floating-point values 39
related documentation 8
REMAPPALETTERGB
  using to set VGA color 51
Resource Editor
  dialog controls in 82, 96, 102
  include files 84
  include files for dialog boxes 84
  list boxes and combo boxes in 105
  using to design Fortran dialog boxes 78
  using to insert an ActiveX* control 113
resource files
  using multiple 83
resources
  using multiple RC files 83
RGB color values 51
routines ending in _w 64

S

samples
  available on the kit 15
SAVEIMAGE
  overview of routines to transfer images 65
SAVEIMAGE_W
  overview of routines to transfer images 65
screen images
  retrieving 64
  storing 64
SCROLLTEXTWINDOW
  coordinates for 38
  overview of routines affecting text display 59
SDI Fortran Windowing applications 9
selecting display options 36
SETACTIVEQQ
  using with active child window 33
SETBKCOLOR
  using for color mixing 49
  using for figure properties 37
  using for text colors 52
SETBKCOLORRGB
  using for color mixing 49
  using for figure properties 37
  using for text colors 52
SETCLIPRGN
  using for graphics coordinates 39, 42
SETCOLOR
  using for color mixing 49
  using to set a color index 51
  using to set figure properties 37
SETCOLORRGB
  using for color mixing 49
  using to set figure properties 37
SETFILLMASK
  example of 56
  using for figure properties 37
SETFONT
  example of 63
  using 62
SETGTEXTROTATION
  overview of graphic display routines 36
SETLINESTYLE
  example of 54, 56
  using for figure properties 37
SETMESSAGEQQ
  using to change QuickWin strings 59, 70
SETPIXELRGB
  example of 55
  using for color mixing 49
SETPIXELSRGB
  using for color mixing 49
SETTEXTCOLOR
  using for color mixing 49
  using for text colors 52
SETTEXTCOLORRGB
  using for color mixing 49
  using for text colors 52
SETTEXTPOSITION
  coordinates for 38
  using for text colors 52
  using to customize text 59
SETTEXTWINDOW
  using to customize text 59
setting control properties 82
setting figure properties 37
setting graphics coordinates 42
setting return values and exiting 112
setting the active window 33
setting the font and displaying text 62
setting the graphics mode 36
SETVIEWORG
  example of 54
  using for graphics coordinates 39, 42
SETVIEWPORT
  using for graphics coordinates 39, 42
  using to customize text 59
SETWINDOW
  using for graphics coordinates 39, 42
SETWINDOWCONFIG
  overview of graphics fonts 61
  using for graphics coordinates 39
  using for VGA palette 51
  using to configure Windows properties 36
  using to display child settings 32
  using to set virtual window properties 30

SETWINDOWCONFIG *(continued)*
    using to set windows coordinates 42
SETWINDOWMENUQQ
    using to create a list of child windows 67
SETWRITEMODE
    using to set logical write mode 37
SETWSIZEQQ
    using to get size or position of window 35
sharing data using Fortran DLLs 18
SHOWFONT.F90 example 63
simulating nonblocking I/O 76
sliders
    using 111
specifying control indexes 101
spin controls
    using 110
standard graphics applications 25, 26, 27
    characteristics of 27
    how to build 25
state messages
    QuickWin routine changing 70
static text
    using 103
STATUS
    specifier in CLOSE 34
status bar
    QuickWin routine changing 70

T

tab controls
    using 111
text
    displaying 62
    editing in QuickWin 65
    QuickWin routines to check or modify 59
    QuickWin routines to customize 59
    QuickWin routines to define 36
text coordinates 38
text output
    setting foreground and background colors for 52
trackbar control 111
transferring images
    in memory 64
type size
    definition of 61
type style
    definition of 61
typefaces 61
types of QuickWin programs 25

U

understanding coordinate systems 38
unit numbers
    converting to Windows handles 25
up-down control 110
user interface in QuickWin 27
using a mouse 72
using Activex* controls 113
using buttons 105
using check boxes and radio buttons 104
using custom icons 71
using dialog controls 102
using dialogs 77
using edit boxes 103
using fonts 62
using fonts from the graphics library 61
using graphics and character-font routines 35
using group boxes 104
using list boxes and combo boxes 105
using pictures 109
using progress bars 110
using QuickWin 23
using scroll bars 109
using sliders 111
using spin controls 110
using static text 103
using tab controls 111
using text colors 52
using the Microsoft* integrated development environment to build dlls 16
using the Resource Editor to design a Fortran dialog 78
using Windows API Routines with QuickWin 25

V

VGA
    color palette 51
    display and resolution in QuickWin 51

W

WAITONMOUSEEVENT
    using 75
Windowing applications
    coding requirements (Fortran) 9
    creating (Fortran) 9
    linking (Fortran) 9
    WinMain function (Fortran) 9
windows properties 30, 33
    routines to set or get 30
Windows*
    creating and controlling QuickWin 29
    creating graphics programs for 66
    creating simple applications for 66
    giving focus in QuickWin 33
    OpenGL 47
    QuickWin routines for size and position of 35
    setting active in QuickWin 33
Windows* APIs 25
    using with QuickWin 25
Windows* applications
    compared to QuickWin 25
Windows* graphics routines
    naming conventions 24
Windows* handles 25
WinMain function for Fortran Windowing applications 9
working with screen images 64

write mode
    QuickWin routines for 37
writing a dialog application 84
writing a graphics program 52
writing a Windows* GDI program 9