S82G-3743-00

IBM C/C++ Tools: User Interface Class Library

User's Guide

Version 2.01

IBM

IBM C/C++ Tools: User Interface Class Library

S82G-3743-00

User's Guide

Version 2.01

IBM C/C++ Tools: User Interface Class Library User's Guide Version 2.01

Document Number S82G-3743-00

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page iii.

Second Edition (November 1993)

This edition applies to Version 2.01 of IBM* C/C++ Tools (Programs 82G3733, 82G3735, 82G3736) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order documentation by phone in the United States or Canada. Call your regular IBM ordering channel, or call:

1-800-342-6672 (United States)
1-800-465-7999 (Canada)

You can also order documentation through your IBM representative or the IBM branch office serving your locality. Documents are not stocked at the address given below.

A form for readers' comments is provided at the back of this document. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1V7

You can also send your comments by facsimile to (416) 448-6057 to the attention of the RCF Coordinator. Or, if you have access to Internet, you can send your comments electronically to IBM: See "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this document.

Internet:	**torrcf@vnet.ibm.com**
IBMLink*:	**toribm(torrcf)**
IBM/PROFS*:	**torolab4(torrcf)**
IBMMAIL:	**ibmmail(caibmwt9)**

If you choose to respond through Internet, please include either your entire Internet network address or a postal address.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Notices

Trademarks

Summary of Changes

The User Interface Class Library Version 2.01 contains the following changes and enhancements:

- Direct manipulation (drag and drop) support
- Interface changes
- New and enhanced classes
- New, enhanced, and deleted member functions
- New, enhanced, and deleted styles

For details on these updates, late-breaking news, and improvements or fixes to the previous version, refer to the READ.ME file shipped with the product.

Direct Manipulation Support

Version 2.01 provides classes and default rendering mechanisms and formats (RMFs) or types so you can add drag and drop support to your applications.

- The direct manipulation classes provide built-in support for entry field and MLE controls. Container controls allow you to move or copy objects between source and target containers.
- Source for default RMFs is provided.
- Drag image support for three styles of drag images is provided. You can construct these images from an icon or bit-map resource or from a pointer or bit-map handle. Samples shipped with the product show you how to do this.

Interface Changes

The following classes have been renamed:

Old Class Name	New Class Name
IHelpHyperTextEvent	IHelpHypertextEvent
IHelpSubItemNotFoundEvent	IHelpSubitemNotFoundEvent
ISubMenu	ISubmenu
ISubMenu::Cursor	ISubmenu::Cursor

The following member function names have been renamed:

Class Name	Old Name	New Name
I3StateCheckBox	selectHalftoned selectHalftone	isHalftoned isHalftone
ICanvas	defaultPushbutton	defaultPushButton
IContainerControl	detailsViewportOnWindow detailsViewportOnWorkspace viewportOnWindow viewportOnWorkspace	detailsViewPortOnWindow detailsViewPortOnWorkspace viewPortOnWindow viewPortOnWorkspace
IHelpHandler	hyperTextSelect showCoverpage subItemNotFound	hypertextSelect showCoverPage subitemNotFound
IHelpWindow	coverpageWindow	coverPageWindow
IMenu	addSubMenu removeSubMenu removeSubMenuAt setSubMenu	addSubmenu removeSubmenu removeSubmenuAt setSubmenu
IMenuItem	setSubMenuHandle subMenuHandle	setSubmenuHandle submenuHandle
ITitle ColorArea enumeration	active inactive	activeFill inactiveFill
IWindow	defaultPushbutton startHandlingEvent	defaultPushButton startHandlingEventsFor

Some duplicate icon styles for IMessageBox were removed. Update these styles in your application with the appropriate style from the following chart.

Retained	Removed
noIcon	notificationIcon
informationIcon	asteriskIcon
queryIcon	questionIcon
warningIcon	exclamationIcon
errorIcon	handIcon, criticalIcon

New and Enhanced Classes

The following class has a new nested class:

Class Name	Member Function
IThread	IThread::Cursor

The IWindowHandle argument has been changed to const for classes that use this as a constructor. The following classes are affected:

I3StateCheckBox	IListBox
IBitmapControl	IMultiLineEdit
ICheckBox	IOutlineBox
IComboBox	IPushButton
IEntryField	IRadioButton
IGroupBox	IStaticText
IIconControl	IWindow

New, Enhanced, and Deleted Member Functions

The following sections list the new, enhanced, and deleted member functions.

New Member Functions

The following classes have new member functions:

Class Name	Member Function
IBitmapControl	constructor setBitmap
IComboBox	first notFound SearchType enumeration
IContainerColumn	dataAsDate dataAsIcon dataAsNumber dataAsString dataAsTime hasHorizontalSeparator hasVerticalSeparator isDate isHeadingIconHandle isHeadingReadOnly isHeadingString isIconHandle isNumber isReadOnly isString isTime
IContainerColumn::HorizontalAlignment	horizontalDataAlignment horizontalHeadingAlignment
IContainerColumn::VerticalAlignment	verticalDataAlignment verticalHeadingAlignment
IContainerControl	isMoveValid splitBarOffset
IDate	asCDATE constructor
IFileDialogHandler	filterName validateName
IFrameExtension	baseRectFor totalRectFor
IFrameHandler	calcRect

Class Name	Member Function
IFrameWindow	borderHeight borderSize borderWidth clientHandle clientRectFor defaultOrdering frameRectFor moveSizeToClient setBorderHeight setBorderSize setBorderWidth setDefaultOrdering usesDialogBackground
IGUIErrorInfo	throwError
IIconControl	constructor setIcon
IListBox	first notFound SearchType enumeration
IMenu	removeConditionalCascade setConditionalCascade
IMultiCellCanvas	defaultStyle disableDragLines disableGridLines enableDragLines enableGridLines hasDragLines hasGridLines setDefaultStyle
INotebook	notebookSize pageSettings
IProgressIndicator	initialize
IReference	operator T*
IStaticText	disableHalftone disableStrikeout disableUnderscore enableHalftone enableStrikeout enableUnderscore isHalftone isStrikeout isUnderscore
ISystemPointerHandle	Identifier enumeration
ITime	asCTIME constructor

Class Name	Member Function
IWindow	constructor create defaultOrdering disableUpdate enableUpdate handleWithId itemProvider positionBehindSibling positionBehindSiblings positionOnSiblings postEvent sendEvent setDefaultOrdering setItemProvider SiblingOrder enumeration

Enhanced Member Functions

The following classes have changed member functions:

Class Name	Member Function
IComboBox	locateText
ICommandHandler	dispatchHandlerEvent
IControl	setFont
IDDEClientConversation	dispatchHandlerEvent
IEditHandler	dispatchHandlerEvent
IFileDialogHandler	dispatchHandlerEvent
IFocusHandler	dispatchHandlerEvent
IFont	constructor
IFontDialogHandler	dispatchHandlerEvent
IFrameExtension	attachTo attachToId drawSeparator fixedSize relativeSize separatorWidth setSize sizeTo
IFrameWindow	client ColorArea enumeration create setClient
IGUIErrorInfo	throwGUIError
IKeyboardHandler	dispatchHandlerEvent
IListBox	locateText
IListBoxDrawItemHandler	dispatchHandlerEvent
IMenuDrawItemHandler	dispatchHandlerEvent
IMenuHandler	menuShowing
IMouseClickHandler	dispatchHandlerEvent

Class Name	Member Function
IMultiCellCanvas	constructor
IMultiLineEdit	setFont setText
INotebook	pageSettings
IPageHandler	dispatchHandlerEvent
IPaintHandler	dispatchHandlerEvent
IResizeHandler	dispatchHandlerEvent
IScrollHandler	dispatchHandlerEvent
ISelectHandler	dispatchHandlerEvent
IShowListHandler	dispatchHandlerEvent
ISpinHandler	dispatchHandlerEvent
IStaticText	disableHalftone disableStrikeout disableUnderscore enableHalftone enableStrikeout enableUnderscore isHalftone isStrikeout isUnderscore
ISystemErrorInfo	throwSystemError
ISystemPointerHandle	Identifier enumeration
ITitle	ColorArea enumeration
IWindow	Layout enumeration

Deleted Member Functions

The following classes have deleted member functions:

Class Name	Member Function
IFrameWindow	locateClient
IObjectWindow	handle
IProgressIndicator	create

New, Enhanced, and Deleted Styles

The following sections list new, enhanced and deleted styles.

New Styles

The following classes have new styles:

Class Name	New Style
IFrameWindow	dialogBackground
IMulticellCanvas	classDefaultStyle dragLines gridLines
ISpinButton	padWithZeros
IStaticText	halftone mnemonic strikeout underscore
IViewPort	noViewWindowFill

Enhanced Styles

For consistency with the rest of the library, the currentDefaultStyle style has been changed from protected to private for the following classes:

IMultiCellCanvas
INotebook
ISplitCanvas

Deleted Style

The noDrawWhenMinimized style has been deleted from IFrameWindow.

Contents

Part 1. Introduction

Chapter 1. About This Book

The *IBM C/C++ Tools: User Interface Class Library User's Guide* (hereafter called *User's Guide*) enables you to start using the IBM C/C++ Tools: User Interface Class Library (User Interface Class Library) classes and helps you learn some of the basic features that the class library provides to help you develop your own applications. This book assumes that you have OS/2 Presentation Manager* (PM) and C++ programming knowledge and experience. Refer to the *C++ Programmers Guide* to review C++ programming concepts and principles.

This *User's Guide* is divided into four parts. Read the "Introduction," which includes the chapter you are reading now through Chapter 3, "Overview of the Classes" on page 15, an overview of the class libraries and definitions for terms used throughout this book. Do this even if you have used class libraries before.

If you are new to C++ class libraries,

read Part 2, "Getting Started" on page 23 first to gain a general understanding of the key concepts used in the User Interface Class Library.

If you have previously used class libraries,

browse Part 2, "Getting Started" on page 23, and then read Part 3, "Programming Advanced Features" on page 83, to learn how you can use them in complex applications.

If you want to use the Hello World sample application,

read Part 4, "Learning from the Sample Application" on page 143. The sample application, "Hello World," uses the User Interface Class Library classes in a variety of ways to give you some examples that you can use as you develop your own code. The Hello World application is divided into several versions. Each version shows you a different aspect of the User Interface Class Library.

If you want to use the User Interface Class Library sample applications,

look in the \ibmcpp\samples\iclui directory. The directory includes the Hello World samples and others that are shipped with the User Interface Class Library.

About the User Interface Class Library

The User Interface Class Library is an object-oriented (OO) C++ class library that simplifies your development of Operating System/2* (OS/2*) applications with graphical user interfaces (GUI). You can use the library to build applications that simulate Common User Access* (CUA*) workplace look and feel and take advantage of PM features.

How to Use This Book

This section contains the following information to help you start using the User Interface Class Library:

- Detailed overview of the *User's Guide*
- Introduction to the contextual help and online documentation
- Hardware and software requirements
- User Interface Class Library conventions

Part 2. Getting Started

This part introduces some of the key concepts about class libraries.

Chapter 3, "Overview of the Classes" on page 15, provides a high-level description of the User Interface Class Library. The classes in the library are grouped into categories based on the tasks you perform when developing applications. This chapter also describes how you can provide double-byte character set (DBCS) and multiple language support.

Chapter 4, "Creating User Interface Class Library Applications" on page 25, describes the classes that make up a typical application and the classes you use to develop basic application components.

Chapter 5, "Creating Windows" on page 35, describes the classes that enable you to create frame extensions, basic controls, and canvas controls.

Chapter 6, "Adding Events and Event Handlers" on page 67, describes the classes that you use to create event handlers and events, as well as explains how you can write your own handler class.

Chapter 7, "Managing Character Data" on page 75, describes the IString and IFont classes.

Part 3. Programming Advanced Features

This part explains some of the more advanced features of the User Interface Class Library.

Chapter 8, "Creating Additional Controls" on page 85, describes classes used to create multiple-line entry field (MLE) controls, containers, and notebook controls.

In Chapter 9, "Covering Advanced Topics" on page 105, you learn about some of the advanced features of the User Interface Class Library classes (including ways to extend event handling, extend exception handling, simplify tracing, and ways to create threads) that enable you to create more complex applications.

Chapter 10, "Creating Dialogs" on page 137, describes the classes that enable you to create standard file dialogs, standard font dialogs, and message boxes.

Part 4. Learning from the Sample Application

This part of the book contains the "Hello World" sample application that helps you apply what you learned in Parts 2 and 3.

Chapter 11, "Introducing the Sample Applications" on page 145, through Chapter 17, "Enabling National Language Support and Advanced Functions" on page 203, take you step-by-step through the Hello World application that is designed to illustrate many of the features of the User Interface Class Library classes and member functions. Each version of the Hello World application builds on concepts covered in the previous versions and shows you a different aspect of the class library. You can find sample code for each of the examples in the User Interface Class Library samples directory, \ibmcpp\samples\iclui, so you can follow along and create your own examples as you read this book.

The Appendix, "Class Hierarchy by Category" on page 207, lists all User Interface Class Library classes.

While reading this *User's Guide*, use the *IBM C/C++ Tools: User Interface Class Library Reference* for complete reference details on the classes. It is available online as part of the product, or you can order a separate hardcopy version.

The Contextual Help Feature

The User Interface Class Library provides contextual help for each class and member function. The requirements for accessing contextual help are:

To install the DDE4UIL.INF and DDE4UIL.NDX files
To use the Enhanced System Editor provided by OS/2 Version 2.0 or 2.1

If you meet these requirements, access contextual help by positioning the cursor over the name of a class or member function in the text you are editing and pressing Ctrl-H. This opens the online version of the *IBM C/C++ Tools: User Interface Class Library Reference* and displays information about that class or member function.

Refer to the product installation instructions for complete details on setting the environment variables needed to use the contextual help feature.

Specified Operating Environment

Hardware Requirements

Disk space: A minimum of 65MB for all the tools in IBM C/C++ Tools.

System units: All system units supported by IBM OS/2 Version 2.0.

If you have an 80386 processor, we recommend an 80387 math co-processor because it greatly increases the speed of floating-point operations. If you have an 80486SX processor, we recommend an 80487 math co-processor for the same reason.

Display: We recommend an IBM 8514 or 8515 color display with IBM 8514A Adapter Card or IBM XGA* displaying 1024 x 768 pels. The minimum requirement is VGA.

Memory: A minimum of 12MB of RAM. We recommend 16MB.

OS/2 swap file: A minimum of 30MB of disk space.

Note: Increasing RAM does not necessarily reduce swap file requirements.

Software Requirements

IBM OS/2 Version 2.0 or 2.1

Note: Generated object programs run under OS/2 Version 2.0 and 2.1.

IBM C Set ++ Version 2.1, or the following products:

- IBM C/C++ Tools Version 2.01
- IBM Developer's Toolkit for OS/2 Version 2.0 with updates or Version 2.1

Conventions Used in This Book

The User Interface Class Library has conventions for the following:

- File names
- Class, member function, and data member names
- Enumerations
- Function return types
- Function arguments
- Additional standards

File Names

All files provided by the User Interface Class Library begin with the letter "I" for IBM, for example, IAPP.HPP. IBM C/C++ Tools product files begin with the letters "DDE4." File names have a maximum of eight characters, including the "I" or "DDE4." The following table lists file names, files extensions, and a brief description.

File Name and Extension	Description
I*xxxxxxx*.H	Constant definitions file
I*xxxxxxx*.HPP	Header file
I*xxxxxxx*.INL	Inline functions
DDE4MUII.LIB	Import library
DDE4MUI.DLL	Multi-threaded dynamic-link library
DDE4MUI.DEF	Import module-definition file used to rebuild the DDE4MUI.DLL file
DDE4MUIB.LIB	Static object library contains the base classes
DDE4MUIC.LIB	Static object library contains the controls
DDE4MUID.LIB	Static object library contains the dynamic data exchange (DDE) and direct manipulation classes
DDE4NIL.NDX	Index for online help
DDE4UIL.INF	Online help
DDE4UILE.MSG	Exception messages

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for an appendix that contains cross-reference tables for the header files and the classes they contain.

Class Names, Member Function Names, and Data Member Names

Class names

are mixed case, with the first letter of each word capitalized, as in ICurrentApplication. All classes in the User Interface Class Library that are global begin with the letter "I."

Member function names and data member names

are also mixed case, except the first letter is always lower case, as in the autoSize data member. In the *User's Guide*, single-word member functions have ClassName:: added to them, for example, IWindow::show. Here are more rules about class and member function names:

Acronyms are uppercase, as in IDBCSBuffer. DBCS is the acronym for double-byte character set. Among other acronyms you will see are GUI (graphical user interface) and DDE (dynamic data exchange).

Abbreviations are mixed case, such as IPresSpaceHandle, which is the class for presentation space handles.

Member functions that query begin with a prefix that implies a query is being conducted, such as "is" or "has." The IDragItem class, for example, has the isCopyable member function, which queries whether an object can be copied.

Member functions that render an object as a different type begin with the "as" prefix, such as asUnsignedLong, which renders an object as an unsigned long.

Member functions that provide enabling or disabling capabilities begin with the "enable" or "disable" prefix. The IEntryField class, for example, provides the enableAutoScroll member function, which enables automatic scrolling.

Member functions that set something begin with the "set" prefix. The setDefaultStyle member function sets the default style for a class.

Member functions that get something, however, do not have a "get" prefix. Instead, many classes use the defaultStyle member function to get the default style for that class.

Member functions that act on objects are verbs, such as copy and move.

Member function names and arguments tend to be self-explanatory. The following example would move the IWindow object, aWindow, to the position specified by the IPoint object, aPoint.

```
aWindow.moveTo( aPoint );
```

Many member functions that toggle the state of an object come with an optional Boolean argument that allows the opposite action of the function. This allows you to use the result of a prior query function as an input argument, such as:

```
Boolean initialVisibility = isVisible();
hide();
/  Do some hidden work  /
show(initialVisibility);
```

Enumerations and enumeration types

conventions are:

The first character of each enumeration name is uppercase. If two words are joined, each begins with an uppercase letter.

Enumerators use the same naming conventions as functions; they begin with lowercase letters, but if two words are joined, the second begins with an uppercase letter.

Function Return Types

for the various types of functions are:

A Boolean (true or false). A testing function typically returns a Boolean, such as:

```
Boolean isValid() const;
```

Note: The User Interface Class Library returns a zero if false and a nonzero if true, so do not test for == true.

An object. Other accessor functions typically return an object, for example:

```
ISize size() const;                    //Returns an object
IWindow  owner();                          //Returns a pointer to an object
static IWindow  desktopWindow();      //Returns a pointer that points to an
                                           //object
```

An object reference. Functions that act on an object return an object reference, such as:

```
IWindow& hide();
```

This allows the chaining of function calls, such as:

```
aWindow.moveTo(IPoint(1 ,1 )).show();
```

Function Arguments

are passed by:

Built-in types (ints, doubles) and enumerations are passed in by value.

Objects are passed by reference (a const reference if the argument is not modified by the function).

Optional objects are passed by pointer. This allows a null pointer to signify that no object is being passed.

IWindow objects are usually passed by pointer.

IContainerObjects are usually passed by pointer.

Strings are passed as const char *. This enables you to pass either an IString or a literal character array.

Additional Conventions

These additional conventions are followed by the User Interface Class Library:

Header files are wrapped to ensure that they are not included more than once.

All functions that can be placed inline are placed in separate INL files with a user option (I_NO_INLINES) to determine whether they should be placed inline into the application code. If you do not want to place these functions inline, then define I_NO_INLINE.

ISYNONYM.HPP declares synonyms for the types and values nested within the IBase class. Including this file places those names in the global name space, which allows consistent use of classes and code derived from IBase. If these global names conflict with other names in your application, modify ISYNONYM.HPP.

Chapter 2. Related Documents

This section lists the related documents comprising the C Set ++ library referenced in this book. The list of related IBM product documents and other related publications is not exhaustive, but should be adequate for most C Set ++ users.

C Set ++ Library

Title	Order Number
C Set ++ Library — Group 1	61G1439
IBM C/C++ Tools: Browser Introduction	61G1397
IBM C/C++ Tools: Debugger Introduction	61G1184
IBM C/C++ Tools: Developer's Toolkit for OS/2: Getting Started	60G9284
IBM C/C++ Tools: Execution Trace Analyzer Introduction	61G1398
IBM C/C++ Tools: Programming Guide	61G1181
IBM C/C++ Tools: Reference Summary	61G1441
IBM C/C++ Tools: User Interface Class Library User's Guide	82G3743
IBM C/C++ Tools: WorkFrame/2 Introduction Version 1.1*	61G1428
IBM C/C++ Tools: WorkFrame/2 Introduction Version 2.1	82G3740

Title	Order Number
C Set ++ Library — Group 2	61G1440
IBM C/C++ Tools: C Language Reference	61G1399
IBM C/C++ Tools: C Library Reference	61G1183
IBM C/C++ Tools: Collection Class Library Reference	61G1178
IBM C/C++ Tools: C++ Language Reference	61G1185
IBM C/C++ Tools: Standard Class Library Reference	61G1180
IBM C/C++ Tools: User Interface Class Library Reference	82G3738

C and C++ Related Publications and Standards

Title	Order Number
Portability Guide for IBM C	SC09-1405

You may also want to refer to the following standards:

American National Standard for Information Systems — Programming Language C (X3.159-1989)
(Draft Proposed) American National Standard for Information Systems — Programming Language C++ (X3J16/92-0060)
International Standard C ISO/IEC 9899:1990(E)

IBM OS/2 2.0 Publications

The following books are available only as part of OS/2 2.0 or the Developer's Toolkit products.

Title	**Order Number**
Developer's Toolkit for OS/2 2.0 Getting Started	10G6199
OS/2 2.0 Getting Started	42G0237
OS/2 2.0 Installation Guide	84F8464
OS/2 2.0 Quick Reference card	42G0231
OS/2 2.0 Using the Operating System	42G0238

IBM OS/2 2.0 Technical Library

The following books make up the OS/2 2.0 Technical Library (10G3356).

Title	**Order Number**
Application Design Guide	S10G-6260
Control Program Programming Reference	S10G-6263
Information Presentation Facility Guide and Reference	S10G-6262
Physical Device Driver Reference	S10G-6266
Presentation Driver Reference	S10G-6267
Presentation Manager Programming Reference Volume 1	S10G-6264
Presentation Manager Programming Reference Volume 2	S10G-6265
Presentation Manager Programming Reference Volume 3	S10G-6272
Procedures Language 2/REXX Reference	S10G-6268
Procedures Language 2/REXX User's Guide	S10G-6269
Programming Guide, Volume I	S10G-6261
Programming Guide, Volume II	S10G-6494
Programming Guide, Volume III	S10G-6495
System Object Model Guide and Reference	S10G-6309
Virtual Device Driver Reference	S10G-6310

IBM BookManager* READ/2 Publications

Title	Order Number
IBM BookManager READ/2: Displaying Online Books	SB35-0801
IBM BookManager READ/2: General Information	GB35-0800
IBM BookManager READ/2: Getting Started and Quick Reference	SX76-0146
IBM BookManager READ/2: Installation	GX76-0147

IBM Common User Access (CUA) Publications

Title	Order Number
Object-Oriented Interface Design: IBM Common User Access (CUA) Guidelines	SC34-4399

Chapter 3. Overview of the Classes

The User Interface Class Library contains over 260 classes and over 2600 member functions. To assist you in learning about the classes and to guide you as you start developing applications, we have categorized the classes into the following basic categories:

Application
Data Types
Event
Exceptions
Handler
Settings and Styles
Support
Window

While most applications use classes from all of these categories, the *User's Guide* focuses on the window category and its relationship with the event, event handler, and the other categories.

Application Classes: The application classes provide support for the application, threads, profiles, and the resources used by the application.

Data Types: The data type classes model basic data types, such as strings, points, and rectangles. These classes hide the structure of the data, while providing the capability to access and alter the data. In addition, a set of handler classes are provided to access window or application-specific handlers.

Event and Handler Classes: The event and event handler classes encapsulate the user's interaction with application windows. The library creates event objects as a result of some action by the user or by other applications. These event objects contain information about what occurred; they are passed to handler objects for processing. Each window has some default event processing; however, the application can create instances of the handler classes to process certain event objects to override the default behavior.

Exception Classes: The exception classes inform the application that the library cannot complete a request. Instances of these classes capture the type of exception and other information about the exception.

Settings and Style Classes: The settings and style classes change the appearance or behavior of window classes. For example, the IFontDialog class allows you to specify the default font and window title to be displayed through its settings class. Setting the text alignment, specifying a minimize button on a frame, or specifying a tab stop are three examples of styles that you can set.

Support Classes: The support classes work with other classes. An example of these classes are cursors.

Window Classes: The window classes encapsulate the basic graphical building blocks that are used to construct application windows. These range from the simple graphical objects like title bars, which display the title of the window, to complex objects like containers, which can contain other objects and provide different views on those objects. Window classes support both parent and owner windows. This allows window position and appearance (parent windows) to be separated from event handling (owner windows).

To learn more about a specific category, refer to the sections listed with each category:

Categories	**References**
Application	"Using the Application Classes" on page 31 "Controlling Threads and Protecting Data" on page 113 "Application Classes" on page 207
Data types and attributes	"Data Types and Attributes Classes" on page 208
Event	Chapter 6, "Adding Events and Event Handlers" on page 67 "Extending Event Handling" on page 105 "Event Classes" on page 210
Exception and error handling	"Handling Exceptions" on page 110 "Simplifying Tracing" on page 108 "Error Handling and Exception Classes" on page 209
Handler	Chapter 6, "Adding Events and Event Handlers" on page 67 "Handler Classes" on page 212
Settings and styles	"Adding Styles" on page 63 "Settings and Styles Classes" on page 213
Supporting classes	"Creating Cursor Classes" on page 65 "Support Classes" on page 214
Window	Chapter 5, "Creating Windows" on page 35 Chapter 8, "Creating Additional Controls" on page 85 Chapter 10, "Creating Dialogs" on page 137 "Window Classes" on page 215

Rebuilding DDE4MUI.DLL

Why You May Want to Rebuild

If you ship a renamed version of DDE4MUI.DLL with your application, you can reduce the size of this DLL by rebuilding DDE4MUI.DLL and leaving out the classes that your application does not use.

A smaller DLL takes up less space on your installation media and can also result in faster load time for the applications that use the DLL.

How to Rebuild

Use the following steps to rebuild DDE4MUI.DLL:

1. Make \ibmcpp\icluidll your current directory. (This is the directory where this text file resides.)

   If C Set ++ is installed in \ibmcpp, type:

   ```
   cd \ibmcpp\icluidll
   ```

   Note: These instructions assume C Set ++ is installed in \ibmcpp.

2. Extract the needed .OBJ files from the User Interface Class Library static libraries.

   To get the best performance for your rebuilt DLL, you must link the object files in the order specified in REBUILD.RSP. To do this, extract the .OBJ files from the User Interface Class Library static libraries instead of re-linking the DLL by using the static libraries directly.

   To extract the needed .OBJ files and put them in the \ibmcpp\icluidll directory, type:

   ```
   GETJOBS ..\LIB\DDE4MUIB.LIB
   GETJOBS ..\LIB\DDE4MUIC.LIB
   GETJOBS ..\LIB\DDE4MUID.LIB
   ```

   Note: If your application does not use the dynamic data exchange (IDDExxx) classes or the direct manipulation (IDMxxx) classes, you do not need to run GETOBJS against DDE4MUID.LIB.

3. Modify the DDE4MUI.DEF and REBUILD.RSP files.

   To remove a class from your rebuilt DLL, you must first determine the name of the .OBJ file in which the class implementation resides. However, be aware that some .OBJ files contain more than one class implementation. If your application uses *any* of the classes that an .OBJ file implements, you cannot remove it.

   The cross-reference tables in Appendix A of the *IBM C/C++ Tools: User Interface Class Library Reference* can help you determine the .OBJ file that implements a given class. Although this table lists the .HPP file, you can generally substitute .OBJ for .HPP to determine the right name for the .OBJ file.

   For some classes, such as IString and IResourceLibrary, a single .HPP file declares the class, but multiple .OBJ files contain the implementation. In these cases, a number is appended to the file name to make the .OBJ file names unique. For example, the implementations of the classes declared in IRESLIB.HPP are in IRESLIB.OBJ, IRESLIB1.OBJ, IRESLIB2.OBJ, IRESLIB3.OBJ, and IRESLIB4.OBJ. For sequentially numbered .OBJ files such as these, either remove all of the .OBJ files or do not remove any.

   Once you determine the .OBJ files that you do not need for your rebuilt DLL, do the following:

   Edit REBUILD.RSP and remove the lines that list the unneeded .OBJ files.

   Edit DDE4MUI.DEF and remove all lines that export the functions contained in the .OBJ files whose lines you removed from REBUILD.RSP.

   REBUILD.RSP and DDE4MUI.DEF are both found in the \ibmcpp\icluidll directory.

   For example, if you do not want to use the ITime class, delete the two lines that export Ordinals 358 and 359. These lines are below the comment lines that identify the exports from ITIME.OBJ:

```
;
; --> Object: C:\DRVRGM3\IBASE\OBJ\ITIME.obj
;
__ls__FR7ostreamRC5ITime   @358  noname          ----> DELETE
asString__5ITimeCFPCc   @359  noname             ----> DELETE
;
```

   The following table can help you identify groups of files that you can delete. This table provides the wildcard .OBJ file name and the conditions under which you can delete all files that match the pattern:

File Name Pattern	Can Be Deleted if Your Application
N*.OBJ	Never uses I_NO_INLINES when compiling
ICNR*.OBJ	Does not use container controls
IDDE*.OBJ	Does not use dynamic data exchange
IDM*.OBJ	Does not use direct manipulation

4. Build the new DLL.

   Once you modify DDE4MUI.DEF and REBUILD.RSP to remove what you do not want to include in your DLL, you are ready to link your .OBJ files. To do this, enter the following commands:

   ```
   ICC @REBUILD.RSP
   IMPLIB DDE4MUII.LIB DDE4MUI.DEF
   ```

   These commands create the DLL (DDE4MUI.DLL) and its corresponding import library (DDE4MUII.LIB) in the current directory, which is \ibmcpp\icluidll. You can now use these two files to link your application.

5. Delete unneeded .OBJ files from the \ibmcpp\icluidll directory

   You can delete the .OBJ files in the current directory once you are sure you no longer need them for rebuilding. If you do this, do *not* delete the DDE4UDLL.OBJ file.

   We ship this file in the \ibmcpp\icluidll directory because it is not available from any of the static libraries. You need this file if you ever attempt to rebuild again. (It is the DLL Init/Term routine.) You may want to put a backup copy of this file in another directory.

   A safe way to delete the unneeded files is to type the following commands:

   ```
   DELETE I .OBJ
   DELETE N .OBJ
   ```

Creating Your Own Classes

Most applications require new classes. You can derive most new classes from an existing class. You derive new classes for two reasons:

- To inherit implementation details from a base class
- To substitute for a base class

The following table provides a starting point to determine the base class to use:

Added New Function	Base Class
Attribute	IBase or IVBase
Canvas class	ICanvas
Control	IControl or ITextControl
Cursor	IVBase
Data type	IBase or IVBase
Dialog window	IFrameWindow
Event	IEvent
Exception	IException
Primary or secondary window	IFrameWindow
Settings	IBase

Added New Function	Base Class
Style	IBitFlag
Window behavior	IHandler

Supporting DBCS and National Languages

You can use one source file for your application code and provide double-byte character set (DBCS) and national language support (NLS) by using separate resource files for the languages you support. The User Interface Class Library approach includes either of the following:

- Use a single executable file with a separate dynamic link library (DLL) for each language.
- Use separate executable files for each language (each with a separate resource file bound to it).

Enabling National Language Support

Because you define message strings in resource files, you can translate them easily to another language without changing the source code. You can bind these resource files to resource DLLs. Your application determines which resource DLLs to dynamically load at run time. For example, in Version 6 of the Hello World application, if the user specifies the parameter "/P" on the command line, the application dynamically loads the Portuguese resource DLL file and generates the text in Portuguese.

The canvas classes allow you to change the window text without having to change any code. Typically, a window's size and position is dependent on the text it displays. As a result, changing the text adversely affects the window's appearance. The canvas classes automatically size and position their child windows at run time, taking into account the current window's text size and font.

Creating DBCS Applications

The following suggestions will assist you in creating DBCS-enabled applications:

Use the IKeyboardEvent class, which provides all the keyboard action event information, for DBCS-enabled applications. To process both single- and double-byte character key events, use the mixedCharacter member function. To process only single-byte characters, use the character member function.

Use the IString class, which is DBCS-enabled and supports mixed strings that contain both single-byte character set (SBCS) and DBCS characters. Objects of the IString class are essentially arrays of characters. The IString class provides functions to test the characters that make up the string. These functions help users determine whether a character is single-byte or double-byte, and whether it is a valid DBCS first byte.

Use the IDBCSBuffer class, which ensures that the search functions do not inadvertently match the second byte of a DBCS character. The IDBCSBuffer class is derived from the IBuffer class, which holds the IString contents. The two bytes of a DBCS character will not be split.

Use the following member functions in a DBCS-enabled application:

Member Function	**Returns True If and Only If**
isCharValid	The character at the given index is in the set of valid characters
isDBCS1	The byte at the given offset is the first byte of DBCS
isPrevDBCS	The character preceding the one at the given offset is a DBCS character

Specify one of the following data type styles when you create and manage the IEntryField and IComboBox classes:

Data Type Styles	**Allows the Following Input**
anyData	A mixture of SBCS and DBCS characters.
dbcsData	DBCS-only data.
mixedData	A mixture of SBCS and DBCS characters. Use this style if you plan to convert data to an EBCDIC code page.
sbcsData	SBCS-only data.

Specify the appDBCSStatus style when constructing an IFrameWindow to include a DBCS status area when the frame appears in a DBCS environment. The User Interface Class Library automatically shares DBCS status control between a parent and child frame window.

Part 2. Getting Started

Chapter 4. Creating User Interface Class Library Applications

To create a User Interface Class Library application, you need to know which files to create and what goes into them. The following list describes the minimum files required for an application. Typically, the name of each file is the same; only the extensions differ.

filename.CPP	Contains the primary C++ code for your application.
filename.HPP	Contains the declaration of any class or classes that you create. You can put each class in a separate .HPP file or all classes in one file. If your classes are used in only one .CPP file, they can be declared in that .CPP file instead.

Optionally, you can create the following files:

filename.RC	Contains the application resource file and associated resources used when the application requires data, such as text strings or bit maps, from an external source. Examples of external sources include .BMP, .ICO, and .DLG files.
filename.H	Contains the header file, which defines constants used in a resource (.RC) file.
filename.DEF	Contains the module definition file, which holds information that defines your application for the linker.
filename.IPF	Contains the text and tags to produce the help information for your application.
filename.MAK	Contains the make file, which holds information to compile and link your application.

Constructing Applications

Write User Interface Class Library applications using the C++ programming language. These files have the following structure:

#include statements

Insert #include statements at the beginning of the file to specify other files that contain information that your application requires. The following order is recommended for #include statements in an application:

1. Standard C library headers
2. OS/2 Toolkit headers
3. User Interface Class Library headers
4. Your class headers

Note: In certain cases, the User Interface Class Library headers can detect whether the OS/2 Toolkit headers are included and can define some Toolkit-specific functions in such cases.

Typical #include statements are:

– #include <Ixxxxx.HPP>

  Includes the header file that contains information about a User Interface Class Library class that your application uses. You must include the header file for each class you use. All User Interface Class Library header files begin with the letter "I."

  Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for an appendix that contains cross-reference tables for header files and the classes they contain.

  For faster compiles, surround #include statements with #define and #endif statements, as follows:

```
#ifndef _IXXXX_
  #include  <IXXXX.HPP>
#endif
```

  where IXXXX is the name of the class library header file being included (without the .HPP extension).

– #include "xxxxx.HPP"

  Represents the inclusion of a header file for a class that you created. Include header files for classes that you create if your source file uses those classes. See "Creating Your Own Classes" on page 19 for more information.

– #include "xxxxx.H"

  Includes the file that defines your constants.

Main procedure which defines the application entry point

Create the primary application window in the main procedure. Then the windows display and event processing starts for the application. Use the run member function within the ICurrentApplication class to do this. See "Starting Event Processing" on page 32 for more information.

Constructor for the application window

Use the IFrameWindow class to construct the application window.

Once the application window is constructed, your application can call other classes to insert controls and dialogs into the window and to handle mouse and keyboard events.

Understanding a Simple Application

An easy way to understand how the classes and objects work together is to look at a simple application. This application has three basic user interface components:

- A standard frame window with a title bar, system menu, border, and minimize and maximize buttons. The window title is set to “Simple Application.”
- A menu bar that contains a single menu item called **Close**. When the user selects this menu item, the application closes the window and ends.
- The rest of the window, or client area, that contains the phrase “Simple Example.”

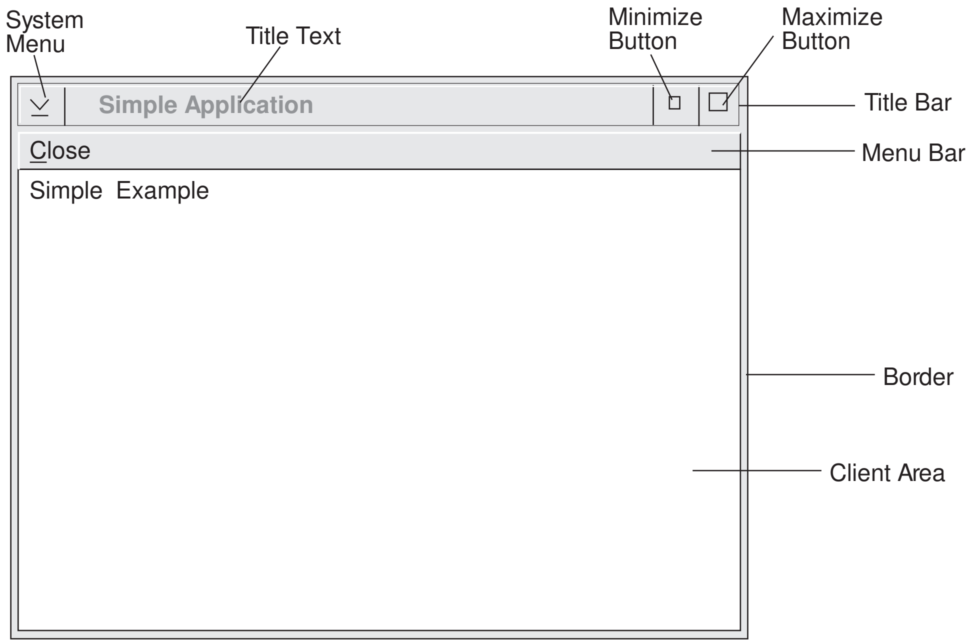


Figure 1. Simple Application Main Window

Two source files are required for this application:

- C++ source file (.CPP file)
- Resource file (.RC file)

Creating a Sample C++ Source File

The first file is the C++ source file used by the C++ compiler to generate the executable part of this application. A copy of the “Simple” application is in the \ibmcpp\samples\iclui\simple directory.

Listing of the C++ source file for the simple application.:

```
#include <iapp.hpp>                              //IApplication Class
#include <iframe.hpp>                            //IFrameWindow Class
#include <icmdhdr.hpp>                           //ICommandHandler & ICommandEvent
#include <istattxt.hpp>                          //IStaticText Class
#include <istring.hpp>                           //IString Class

#define WND_MAIN          5                      //Main Window Id
#define MI_CLOSE          5  1                   //Command Id

class AWindow : public IFrameWindow,          //Define AWindow Class from
                        public  ICommandHandler   //   IFrameWindow  &  ICommandHandler
{
public:
  AWindow(unsigned long windowId)              //Define AWindow Constructor
    : IFrameWindow (                           //Call IFrameWindow constructor
      IFrameWindow::defaultStyle()             //   Use default styles plus
      | IFrameWindow::menuBar,                 //   Get Menu Bar from Resource File
      windowId)                                //   Main Window Id
  {
    IString aString("Simple Example");         //Create text string for static text
    IStaticText   staticText=new               //Create Static Text Control
      IStaticText (5  2, this, this);          //   Pass in myself as parent & owner
    staticText->setText(aString);              //Set text in Static Text Control
    handleEventsFor(this);                     //Set self as command event handler
    setClient(staticText);                     //Set button control in Client Area
    setFocus();                                //Set focus to main window
    show();                                    //Set to show main window
  } /  end AWindow :: AWindow(...)  /

  Boolean command(ICommandEvent& cEvent)//Define command member function
  {
    if (cEvent.commandId() == MI_CLOSE) //Is Command Event Id = Close Id
    {                                          //   Yes, the command is close
      close();                                 //   Let's close the main window
      return true;                             //   Normally, you would return true
    };                                         //   to indicate command processed
    return false;                              //Return Command not Processed
  } /  end AWindow :: command(...)  /
};                                             // End of AWindow class definition

void main()                                    //Main Procedure with no parameters
{
  AWindow mainWindow(WND_MAIN);                //Create main window on the desktop
  IApplication::current().run();               //Get current application & start
} /  end main  /
```

Lines 1-5 include the class header files needed from the class library for the application. WND_MAIN (line 7) is used as the window ID for the main window. MI_CLOSE (line 8) is used as the command ID for the **Close** menu item.

A class called AWindow is defined in lines 10-39. This class is derived from the IFrameWindow and ICommandHandler classes (lines 10-11). The AWindow class has a single constructor (lines 14-28) and a single member function called AWindow::command (lines 30-39).

This application creates the following objects:

mainWindow	This AWindow object is the main window for the application. It is constructed in line 43.
staticText	This is the static text control (IStaticText) object that contains the phrase “Simple Application.” This object is constructed on line 21.
aString	This IString object is constructed on line 20. It contains the phrase “Simple Application” that is set in the staticText object on line 23.
title bar	This window is created because the code specifies the default styles on the IFrameWindow constructor on lines 15-16.
menu bar	This object is constructed by the User Interface Class Library as a result of specifying the menuBar style on the IFrameWindow constructor on lines 15-17. Several handlers are also created supporting the menu bar. In this application, the menu bar sends a command event to the frame window when the user selects the **Close** menu item. Line 24 specifies that we are a command handler for ourselves. Lines 30-38 define the processing for command events sent to the frame window.
cEvent	This ICommandEvent object is created by the User Interface Class Library, which is a parameter on the command member function on line 30. This object returns the command ID on line 32 and is compared against MI_CLOSE. If it is an MI_CLOSE command, the application closes the window and ends using close on line 34; otherwise, the member function returns false indicating that the command has not been processed.

Defining Application Resources

Resources are defined in a resource script file, an ASCII text file that you manipulate using a standard text editor. The menu, string table, and dialog template are examples of the resources defined in the resource file. Each static string used in a window has a corresponding string ID that you define in the resource file. The resource compiler produces a compiled version of the resources, which is then incorporated into the application’s executable code or stored in a dynamic link library (DLL) for use by one or more applications.

A major benefit of defining such resources externally to the application is that changes can be made to resource definitions without affecting the application code itself.

You can also provide national language versions by storing the resources for each language in a separate resource file. You can then build your application as separate executable versions for each language (each with a different resource file bound to it) or as a single executable with a separate DLL for each language.

Creating a Sample Resource File

The second file contains the resource definitions used by the resource compiler to generate the resources for this application.

```
 1 #define WND_MAIN             5                  //Main window Id
 2 #define MI_CLOSE             5 1                //Command Id
 3
 4 STRINGTABLE
 5    BEGIN
 6       WND_MAIN,     "Simple Application"      //Title bar text (main ID)
 7    END
 8
 9 MENU WND_MAIN                                   //Main Window Menu (WND_MAIN)
1    BEGIN
11       MENUITEM "~Close",    MI_CLOSE           //Close Menu Item
12    END
```

Line 1 defines WND_MAIN for this resource file. This number, 5000, must match the definition on line 7 of the C++ source file. Line 2 defines MI_CLOSE for this resource file. This number, 5001, must match the definition on line 8 of the C++ source file.

Line 6 defines a string resource containing the phrase “Simple Application” with a string ID of WND_MAIN. Because this matches the main window ID used on line 43 of the C++ source program, the User Interface Class Library uses this string as the default window title. You can change the “Simple Application” title without changing the application code.

Lines 9-12 define the menu bar used by this application. Because we specified the menuBar style on line 17, the User Interface Class Library attempts to load the menu with an ID equal to the window ID. In this example, the main window ID is WND_MAIN (5000) and line 9 defines the menu with a menu ID of WND_MAIN; therefore, the User Interface Class Library uses this menu bar for the main window.

The close menu item is defined on line 11. This menu item appears to the user as **Close** with a command ID of MI_CLOSE. When the user selects this menu item, lines 30-39 in the C++ source code are executed. You can change the “Close” phrase (for example, it could be “Quit”) without changing the application code. This is important if you translate your application into other languages or if you make required changes for end users. It is also possible to reorganize complex menus with many menu items and submenus without changing the application code.

Using the Application Classes

To develop a User Interface Class Library application, you always use IApplication and the single instance of its derived class, ICurrentApplication.

Objects of the ICurrentApplication class represent the application that is currently running.

There is a single instance of this class. Obtain a reference to it by using the static member function IApplication::current. The instance of this class contains information about the application that is accessible to the objects executing in the process.

Use the ICurrentApplication class member functions to:

- Record and query command line arguments to the application
- Start event processing
- Identify the primary resource library used by the application
- Exit from an application

Recording and Querying Command Line Arguments

With ICurrentApplication, you can record and query the command line arguments of your application. Set the arguments by calling setArgs, passing in the arguments that were passed to the main procedure.

To query the number of arguments, use the member function ICurrentApplication::argc. This member function returns a nonzero value because it always has at least the name of the application as a parameter.

To get the nth parameter, use the member function ICurrentApplication::argv, where the argv(0) component is always the name of the application. Because argv is returned as an IString, you can use all the overloaded operators for this class.

For example, the following code records the command line parameters.

```
void main(int argc____, char   argv____)             //Main procedure with arguments
{
  IApplication::current().setArgs(argc      ____,  argv__);

⋮
```

Where

argc is the number of arguments received

argv represents the actual arguments.

See Chapter 7, "Managing Character Data" on page 75, and refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for more information about the ICurrentApplication and IApplication classes.

Starting Event Processing

To start event processing for a C++ application using the User Interface Class Library, use the run member function of the ICurrentApplication class. For example:

```
void main()                                   //Main procedure with no arguments
{
  AHelloWindow mainWindow (WND_MAIN);     //Create our main window on the desktop
  IApplication::current().run();          //Get current & run the application
} /  end main  /
```

Exiting from an Application

To exit from an application, use the exit member function, highlighted in this example:

```
if (IApplication::current().argv(1)=="") { //If no command line arguments
     IApplication::current().exit();          //Get current & exit the application
  } /  endif  /
} /  end AHelloWindow :: AHelloWindow(...)  /
```

Loading Resources into an Application

Using the User Interface Class Library, you can load a resource from a DLL. Use the ICurrentApplication member function setUserResourceLibrary to identify which resource library will be used if none is specified on a call that loads a resource. The following highlighted code shows an example.

```
void main(int argc, char   argv)              //Main procedure with no arguments
 {
  IApplication::current().                      //Get  current
     setArgs(argc, argv);                         //    and set command line arguments

  IString  Dllname(IApplication::current().argv(1));

  IApplication::current().                      //Get current application
    setUserResourceLibrary(Dllname.asString()); //    Set the name of resource DLL        .

  AHelloWindow mainWindow (WND_MAIN);          //Create our main window on the desktop

  IApplication::current().run();               //Get current & run the application

} /  end main  /
```

First the library tries to create a frame window by loading a dialog with the WND_MAIN ID from the default user resource library.

You can also determine the default user resource library by calling the userResourceLibrary member function. The following highlighted code shows an example.

```
AHelloWindow :: AHelloWindow(unsigned long windowId)
   : IFrameWindow(windowId)                     //Call IFrameWindow constructor
{
   hello = new IStaticText(WND_HELLO,      //Create static text control
      this, this);                               //    Pass in myself as owner & parent
  hello->setText(
   IApplication::current().
   userResourceLibrary().asString());

                                                  //Set text in static text control
   hello->setAlignment(                         //Set alignment to center in both
    IStaticText::centerCenter);              //    directions
   setClient(hello);                            //Set hello control as client window

   setFocus();                                    //Set focus to main window
   show();                                         //Set to show main window
} /  end AHelloWindow :: AHelloWindow(...)  /
```

Linking Your Application to the User Interface Class Library

Link your application to the User Interface Class Library in one of the following ways, depending on your needs.

1. If you are developing an application, use the DDE4MUII.LIB import library.

   Your application links dynamically to DDE4MUI.DLL at run time. Using this DLL during development reduces the time it takes to link your application and the amount of swap space used. You must also link with the following libraries:

   DDE4CCI.LIB, which resolves Collection Class library references and uses DDE4CC.DLL.

   DDE4MBSI.LIB, which resolves C++ runtime external references and uses DDE4MBS.DLL.

2. If you are shipping an application, use the DDE4MUIB.LIB and DDE4MUIC.LIB static object libraries. Also use the DDE4MUID.LIB static object library when your application includes dynamic data exchange (DDE) or direct manipulation.

   Using these libraries does not create dependencies on DDE4MUI.DLL.

   You must also link your application to the following libraries:

   DDE4CC.LIB, which resolves Collection Class Library references.

   DDE4SBS.LIB or DDE4MBS.LIB, which resolve C++ runtime external references.

   Note: If you use the IThread class to start multiple threads, you must use DDE4MBS.LIB.

The following additional rules apply when you build your application with the dynamic libraries, instead of the static object libraries:

1. A DLL using the User Interface Class Library must link dynamically to the User Interface Class Library code (that is, it must link with DDE4MUII.LIB).
2. An EXE using the User Interface Class Library and calling a DLL that also uses the class library must link dynamically to the User Interface Class Library (that is, it must link with DDE4MUII.LIB).
3. An EXE or DLL file should not link both dynamically and statically to the User Interface Class Library code.

Stack Size Requirements for Your Application

This only applies if you are using OS/2 Version 2.0. You should build your applications with a minimum stack size of 4K. If you encounter an insufficient stack size error at run time, increase the size of your stack. You may also encounter false out-of-stack errors.

Note: You may increase or decrease the size of your stack by some number other than a multiple of 4K to find a value that allows you to run your application.

#pragma Priority Values

The User Interface Class Library reserves the use of #pragma priority values in the range of -2147482624 through -2147481600. The C++ compiler reserves the range below that. As a result, avoid using a #pragma priority value less than -2147481599 (this is equivalent to INT_MIN + 2048) to control the order of static object construction in your User Interface Class Library application.

See the *C++ Language Reference* for more information on #pragma priority values.

Chapter 5. Creating Windows

When you develop an application, you usually start with a window that is a composite of the frame window, several frame-control windows, and a client window. The frame window coordinates the actions of the frame controls and client window, enabling the composite window to act as a single unit.

The User Interface Class Library provides classes that construct the frame window and that allow you to add a variety of styles and controls.

For a discussion and description of frame windows, refer to the *OS/2 2.0 Technical Library Programming Guide Volume II*.

Creating a Frame Window

A *frame window* is a window that an application uses as the base when constructing a main window or other composite window, such as a dialog window or message box. A frame window provides basic features, such as borders and a menu bar. It can also have a set of resources associated with it, such as icons, menus, and accelerators.

Use the IFrameWindow class to create a frame window. The default style of the IFrameWindow class has a title bar, system menu, minimize button, maximize button, and border. The default style adds an entry for the frame window to the system window list.

The IFrameWindow class also provides several other styles. You can, for example, associate an accelerator key table to the frame window or an icon to be used when the window is minimized.

When you construct an IFrameWindow with a style of minimizedIcon, accelerator, or menuBar, resources corresponding to the style must be in the resource library you use to construct the frame. This library is usually the default user library, which you use by entering:

```
IApplication::current().userResourceLibrary()
```

However, you can also specify the resource library on the IFrameWindow constructor by using the const IResourceId argument.

If a required resource is not found, an exception is thrown and the frame window is not constructed.

See “Adding Styles” on page 63 for more information on setting styles. For a list of the styles provided with IFrameWindow, refer to the *IBM C/C++ Tools: User Interface Class Library Reference*.

Figure 2 shows the components of a frame window created using the IFrameWindow class with the default style and some added controls.

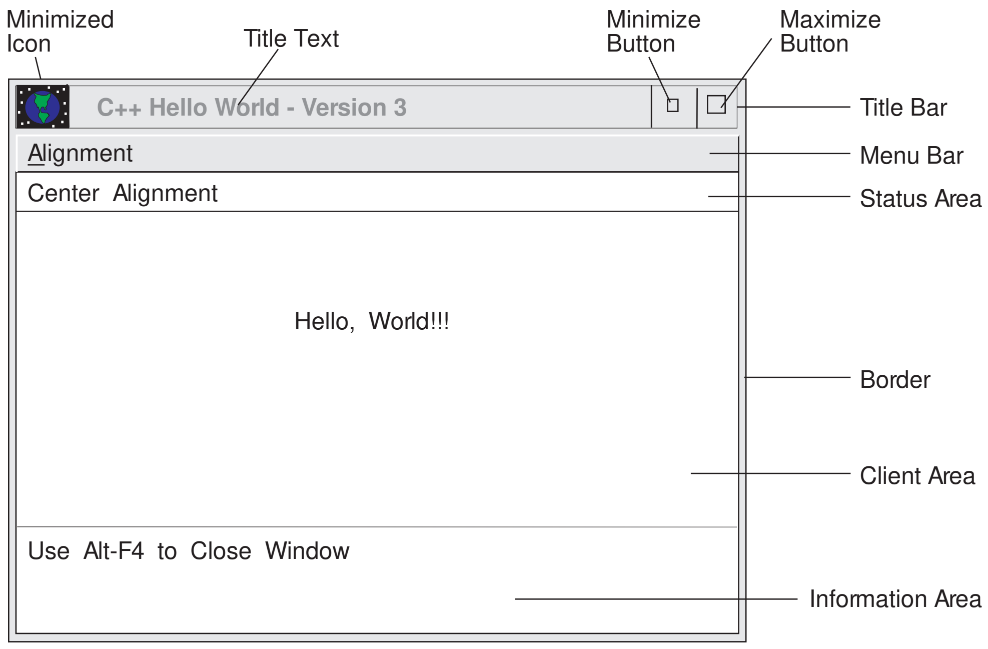


Figure 2. Frame Window Components

The following example defines a frame window:

```
AHelloWindow :: AHelloWindow(unsigned long windowId)
  :IFrameWindow(IFrameWindow::defaultStyle())
```

When a frame window is minimized, the frame window hides and draws its minimized icon. Sometimes other windows associated with the frame window are drawn on top of its icon. This occurs when the windows are children of the frame window but not the client window. This can also happen when frame extensions are added to the client area, for example, instances of the IInfoArea class.

To suppress the drawing of these windows when they are supposed to be minimized, add a handler to the frame window, detect when the frame is minimized, and hide these windows. The windows should be visible when the frame is restored.

The User Interface Class Library minimizes the amount of work to construct an IFrameWindow because much of the frame control and extension layout is deferred until the frame window shows. As a result, if you query the size and position of the frame window's client window or frame extensions, an accurate value will not be returned until the frame window updates.

The IWindow::show or showModally member functions automatically update the frame window. You can force the frame window to update by calling the update member function.

Changing the Title Bar

The *title bar* is the area at the top of each frame window that contains a minimized icon, a window title, and the maximize and minimize buttons.

You can specify the minimized icon and the window title when you create the frame window.

If you do not provide a window title, your application sets the title to a string loaded from the application's resource library. If your application cannot find a string, the title defaults to the system-generated title (typically, the name of the executable file).

The following example shows how to specify a minimized icon and the window title when you create the frame window.

1. The icon and title text are defined in the resource file:

   ```
   ICON WND_MAIN HELLO.ICO
   STRINGTABLE
     BEGIN
       WND_MAIN,     "Title Bar Sample"
     END
   ```

   WND_MAIN is the frame window identifier. The frame window uses the window identifier (windowId) passed on the constructor to load its icon, title, menu bar, or accelerator table resources if these components are specified in the frame window style.

2. The following code comes from the AHELLOW3.CPP file:

   ```
   ⋮
   void main()                                        //Main Procedure with no parameters
   {
     AHelloWindow mainWindow (WND_MAIN);     //Create our main window on the
                                                         //  desktop
     IApplication::current().run();             //Get the current application and
                                                          // run it
    } /  end main  /
   ⋮
   AHelloWindow :: AHelloWindow(unsigned long windowId)
     : IFrameWindow (                              //Call IFrameWindow constructor            v2
       IFrameWindow::defaultStyle()              //    Use default plus                          v2
       | IFrameWindow::minimizedIcon,          //    Get Minimized Icon from    RC file    v2
       windowId)                                  //    Main Window ID
   ⋮
     sizeTo(ISize(4  ,3  ));                      //Set the size of main window              v2
     setFocus();                                  //Set focus to main window
     show();                                       //Set to show main window
   ⋮
   ```

   When the application creates the AHelloWindow object, it constructs the IFrameWindow base class using the default style with a minimized icon, HELLO.ICO, and "Title Bar Sample" as the title text.

Adding a Menu Bar

The *menu bar* is the area near the top of a window, below the title bar and above the client area of the window. A menu bar contains a list of choices. When a user selects a choice on a menu bar, a pull-down menu associated with that choice is displayed.

The following example uses the IMenuBar class to add a menu bar with only one submenu named **Alignment**. When you run the example and select **Alignment**, the pull-down menu is displayed. The choices in the pull-down menu are **Left**, **Center**, and **Right**. When you select one of the choices, the text string in the client window aligns to the selected position.

1. The following code from the AHELLOW3.RC file defines the text for the menu bar and its associated pull-down menu.

```
⋮
MENU WND_MAIN                                      //Main Window Menu (WND_MAIN) v3
  BEGIN
    SUBMENU "~Alignment", MI_ALIGNMENT             //Alignment Submenu          v3
      BEGIN
        MENUITEM "~Left",    MI_LEFT               //Left Menu Item             v3
        MENUITEM "~Center", MI_CENTER              //Center Menu Item           v3
        MENUITEM "~Right",   MI_RIGHT              //Right Menu Item            v3
      END
  END
```

2. The following code from the AHELLOW3.HPP file shows the addition of the IMenuBar object to the AHelloWindow class:

```
⋮
class AHelloWindow : public IFrameWindow,
                     public  ICommandHandler                        //v3
{
  public:                              //Define the Public Information
    AHelloWindow(unsigned long windowId); //Constructor for this class

  protected:                           //Define Protected Member          v3
    Boolean command(ICommandEvent& cmdEvent);                       //v3

  private:                             //Define Private Information
    IStaticText     hello;             //Hello contains "Hello, World" text
    IInfoArea       infoArea;          //Define  an  Information  Area    v2
                                       //   Control to create an information  .
                                       //   area beneath the client area  v2
    IStaticText     statusLine;        //Status Line at top of client window v3
    IMenuBar        menuBar;           //Define Menu Bar                   v3
};
#endif
```

3. This code is from the AHELLOW3.CPP file:

```
⋮
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow (                              //Call IFrameWindow constructor        v2
    IFrameWindow::defaultStyle()              //   Use default plus                    v2
    | IFrameWindow::minimizedIcon,            //   Get Minimized Icon from   RC file   v2
    windowId)                                 //   Main Window ID
{
  hello=new IStaticText(WND_HELLO,            //Create Static Text Control
    this, this);                              //   Pass in myself as parent & owner
  hello->setText(STR_HELLO);                  //Set text in Static Text Control       v2
  hello->setAlignment(                        //Set Alignment to Center in both
   IStaticText::centerCenter);                //   directions
  setClient(hello);                           //Set hello control as Client Window
⋮
  handleEventsFor(this);                      //Set self as event handler (commands)v3
  menuBar=new IMenuBar(WND_MAIN, this); //Create Menu Bar for main window             .
 menuBar->checkItem(MI_CENTER);               //Place Check on Center Menu Item        v3
⋮
Boolean AHelloWindow :: command(ICommandEvent & cmdEvent)                           // .
{                                                                                    //v3
  switch (cmdEvent.commandId()) {             //Get command id                         v3

    case MI_CENTER:                           //Code to Process Center Command Item v3
      hello->setAlignment(                    //Set alignment of hello text to          .
        IStaticText::centerCenter);           //   center-vertical, center-horizontal .
      statusLine->setText(STR_CENTER);        //Set Status Text to "Center" from Res .
     menuBar->checkItem(MI_CENTER);           //Place Check on Center Menu Item         .
      menuBar->uncheckItem(MI_LEFT);          //Uncheck Left Menu Item                  .
      menuBar->uncheckItem(MI_RIGHT);         //Uncheck Right Menu Item                 .
      return(true);                           //Return command processed                .
     break;                                   //                                        v3

    case MI_LEFT:                             //Code to Process Left Command Item     v3
      hello->setAlignment(                    //Set alignment of hello text to          .
       IStaticText::centerLeft);              //   center-vertical,  left-horizontal    .
      statusLine->setText(STR_LEFT);          //Set Status Text to "Left" from Res      .
     menuBar->uncheckItem(MI_CENTER);         //Uncheck Center Menu Item                .
      menuBar->checkItem(MI_LEFT);            //Place Check on Left Menu Item           .
      menuBar->uncheckItem(MI_RIGHT);         //Uncheck Right Menu Item                 .
      return(true);                           //Return command processed                .
     break;                                   //                                        v3

    case MI_RIGHT:                            //Code to Process Right Command Item   v3
      hello->setAlignment(                    //Set alignment of hello text to          .
       IStaticText::centerRight);             //   center-vertical,  right-horizontal   .
      statusLine->setText(STR_RIGHT);         //Set Status Text to "Right" from Res     .
     menuBar->uncheckItem(MI_CENTER);         //Uncheck Center Menu Item                .
      menuBar->uncheckItem(MI_LEFT);          //Uncheck Left Menu Item                  .
      menuBar->checkItem(MI_RIGHT);           //Place Check on Right Menu Item          .
      return(true);                           //Return command processed                .
     break;                                   //                                        v3

  } /  end switch  /                          //                                        v3
⋮
```

Lines 25 through 29 create the AHelloWindow object.

Lines 31 through 36 create an IStaticText control and sets it as the client window.

Line 48 adds an event handler via handleEventsFor(this) to handle events that originate from the menu bar.

Line 49 creates an IMenuBar object.

Line 50 sets the default submenu item as **Center**.

Lines 62 through 96 implement a command handler to handle the selection of menu items.

Creating a Status Area

The *status area* is a small rectangular area that is usually located at the top of a window, below the menu bar. The status area displays information about the state of an object or the state of a particular view of an object.

To create a status area:

1. In the .HPP file, define an IStaticText object in the AHelloWindow class. This example defines an object called statusLine.

```
class AHelloWindow : public IFrameWindow,
                              public  ICommandHandler
{
  public:
     AHelloWindow(unsigned long windowId);
    virtual  ~AHelloWindow();
  protected:
     Boolean command(ICommandEvent& cmdEvent);
  private:
    IMenuBar            menuBar;
    IStaticText      hello;
    IInfoArea          infoArea;
    IStaticText      statusLine;
};
```

2. In the .CPP file, modify the constructor of AHelloWindow to include a status area when AHelloWindow is created.

   In this example, WND_STATUS is a control ID defined in the header file. STR_STATUS, defined in the resource file, is a string resource ID that specifies a string to be displayed in the status area.

```
AHelloWindow :: AHelloWindow(unsigned long windowId)
   : IFrameWindow ( IFrameWindow::defaultStyle()
                              | IFrameWindow::minimizedIcon, windowId),
    menuBar(WND_MAIN,this),
    hello(WND_HELLO,this,this),
    infoArea(this),
    statusLine(WND_STATUS,this,this)
{
  statusLine.setText(STR_STATUS);
  addExtension(&statusLine,
    IFrameWindow::aboveClient,
    IFont(statusLine).maxCharHeight());
⋮
}
```

Creating an Information Area

The *information area* is a small rectangular area that is usually located at the bottom of a window. You can use the information area to display:

- A brief explanation of the state of an object
- Information about the completion of a process

Use the IInfoArea class to create and manage the information area. Objects of IInfoArea class provide a frame extension to show information about the menu item where the cursor is positioned. The string displayed in the information area is defined in a string table in the resource file.

The following example uses the IInfoArea class to create the information area and the text to display in it.

1. The menu bar and string table are defined in the resource file. The string table contains strings of text and each string is associated with a menu item. When you choose the menu item, the string related to that item is displayed in the information area.

```
MENU WND_MENU
  BEGIN
     SUBMENU "~Alignment", MI_ALIGNMENT
       BEGIN
          MENUITEM  "~Left",     MI_LEFT
           MENUITEM "~Center", MI_CENTER
          MENUITEM  "~Right",    MI_RIGHT
       END
  END
STRINGTABLE
  BEGIN
     MI_ALIGNMENT "Select Alignment Menu"
     MI_LEFT          "Select Left Alignment Menu Item"
    MI_CENTER      "Select  Center  Alignment  Menu  Item"
     MI_RIGHT        "Select Right Alignment Menu Item"
  END
```

2. This code is from the .HPP file. The highlighted line adds an object, infoArea, to the AHelloWindow class.

```
class AHelloWindow : public IFrameWindow,
                              public  ICommandHandler
{
  public:
     AHelloWindow(unsigned long windowId);
  protected:
     Boolean command(ICommandEvent& cmdEvent);
  private:
    IMenuBar          menuBar;
    IStaticText     hello;
    IInfoArea        infoArea;
};
```

3. In the .CPP file, the constructor of AHelloWindow is modified. The information area is constructed when AHelloWindow is created.

```
AHelloWindow :: AHelloWindow(unsigned long windowId)
   : IFrameWindow ( IFrameWindow::defaultStyle()
                          | IFrameWindow::minimizedIcon, windowId),
     menuBar(WND_MENU,this),
     hello(WND_HELLO,this,this),
     infoArea(this)
{
⋮
}
```

Creating Basic Window Controls

A window *control* is a part of the user interface that allows a user to interact with data. Controls are usually identified by text; for example, headings, labels in push buttons, field prompts, and titles in windows.

This section explains how to code the following controls:

- Static text
- Entry fields
- Push buttons
- Check boxes
- Radio buttons
- Sliders

Creating a Static Text Control

Static text controls are text fields, bit maps, icons, and boxes that you can use to label or box other controls. The IStaticText class creates and manages the static text control window.

You can set the text and control its color, size, and position in the static text window.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for a list of the public members provided with IStaticText.

The following example comes from the AHELLOW1.CPP sample file.

```
⋮
 9 void main()                                     //Main procedure with no parameters
1  {
11    IFrameWindow   mainWindow=new                 //Create our main window on the desktop
12      IFrameWindow( x1   );                      //    Pass in our Window ID
13
14    IStaticText   hello=new IStaticText(     //Create static text control with
15       x1 1 , mainWindow, mainWindow);       //    mainWindow as parent & owner
16    hello->setText("Hello, World!");         //Set text in Static Text Control
17    hello->setAlignment(                        //Set Alignment to Center in both
18      IStaticText::centerCenter);              //    directions
19
2    mainWindow->setClient(hello);              //Set hello control as Client Window
21    mainWindow->setFocus();                      //Set focus to main window
22    mainWindow->show();                            //Set to show main window
23
24    IApplication::current().run();            //Get the current application and
25                                                   // run it
26 } /  end main  /
```

Lines 14 and 15 use the window ID, the parent window, and the owner window to create the static text control and an object for it.

Line 16 sets a text string in the IStaticText class, using the setText member function, which is inherited from ITextControl.

Lines 17 and 18 use the setAlignment member function to position the text. Figure 3 shows the nine locations for positioning text within a static text control.

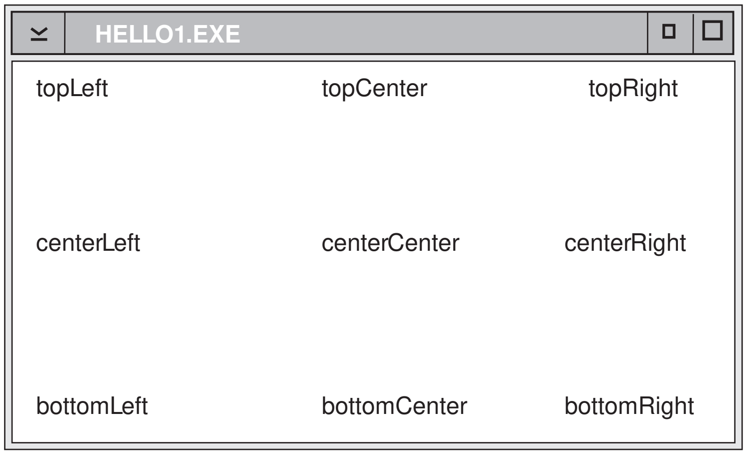


Figure 3. Aligning Text in a Window

Creating an Entry Field Control

An *entry field* is a control a user can put text into.

The following example, from Version 4 of the Hello World application, shows how to define and create an entry field.

1. This code from the ADIALOG4.DLG file defines the entry field.

```
 1 DLGINCLUDE 1 "AHELLOW4.H"
 2
 3 DLGTEMPLATE WND_TEXTDIALOG LOADONCALL MOVEABLE DISCARDABLE
 4 BEGIN
 5      DIALOG   "Hello World Edit Dialog", WND_TEXTDIALOG, 17, 22, 137, 84,
 6                 WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
 7      BEGIN
 8           DEFPUSHBUTTON     "OK", DID_OK, 6, 4, 4 , 14
 9           PUSHBUTTON        "Cancel", DID_CANCEL, 49, 4, 4 , 14
 1           LTEXT             "Edit Text:", DID_STATIC, 8, 62, 69, 8
 11          ENTRYFIELD        "", DID_ENTRY, 8, 44, 114, 8, ES_MARGIN
 12      END
 13 END
```

   Line 11 sets the field to be empty, supplies the entry field id (DID_ENTRY), describes the dimensions of the entry field window, and adds a border.

2. This code from the ADIALOG4.CPP file creates the entry field control.

```
 ⋮
 23   textField=new IEntryField(DID_ENTRY,   //Create entry field object using dialog
 24     this);                               //   entry  field
 25   textField->setText(textString);        //Set top current "Hello, World" text
 ⋮
 44        textValue=textField->text();      //Get Text from Dialog Entry Field
 ⋮
```

   Lines 23 through 25 create an entry field object and set the text using the member function setText.

   After an event occurs, line 44 queries the entry field for its contents using the member function ITextControl::text. (See “The Text Dialog Source Code File” on page 178 for complete listing of ADIALOG4.CPP.)

   Figure 4 on page 46 shows the window created using this sample.

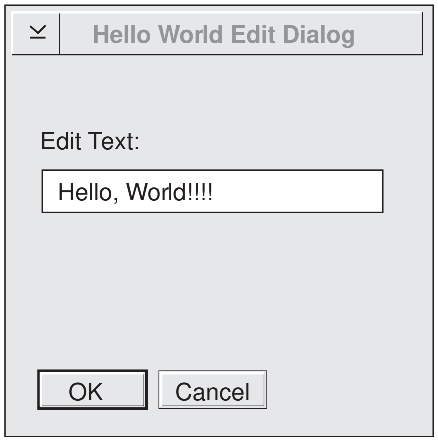


Figure 4. Example of an Entry Field Control

Creating a Push Button Control

A *push button* is a button that represents an action that is initiated when a user selects it. You can label a push button with text, graphics, or both. When a user selects a push button, the action occurs immediately if there is a handler for the generated command event.

Use the IPushButton class to create and maintain the push button control window. By default, a push button generates an application ICommandEvent. You can change the default style by changing the window style value to generate a help event or an ISysCommandEvent.

Refer to *IBM C/C++ Tools: User Interface Class Library Reference* for a list of the styles provided for IPushButton.

In Version 4 of the Hello World application, the member function setupButtons creates three push buttons (see lines 77 through 91). Adding the IControl::tabStop into the default style allows the user to use the Tab key to move the selection between the push buttons.

```
⋮
65 //                                             v4
66 // AHelloWindow :: setupButtons                                         .
67 //     Setup Buttons                                                     .
68 //                                             .
69 Boolean AHelloWindow :: setupButtons()   //Setup Buttons                 .
7  {                                         //                              .
71     ISetCanvas        buttons;              //Define canvas of buttons    .
72                                           //                              .
73     buttons=new ISetCanvas(WND_BUTTONS,     //Create a Set Canvas for Buttons .
74        this, this) ;                       //   Parent and Owner=me       .
75     buttons->setMargin(ISize());           //Set Canvas Margins to zero  .
76     buttons->setPad(ISize());              //Set Button Canvas Pad to zero .
77     leftButton=new IPushButton(MI_LEFT,     //Create Left Push Button      .
78        buttons, buttons, IRectangle(),     //   Parent, Owner=Button Canvas .
79        IPushButton::defaultStyle() |       //   Use Default Styles plus    .
8        IControl::tabStop);                 //   tabStop                     .
81     leftButton->setText(STR_LEFTB);         //Set Left Button Text         .
82     centerButton=new IPushButton(MI_CENTER,//Create center push button     .
83        buttons, buttons, IRectangle(),     //   Parent, Owner=Button Canvas .
84        IPushButton::defaultStyle() |       //   Use Default Styles plus    .
85        IControl::tabStop);                 //   tabStop                    .
 86    centerButton->setText(STR_CENTERB);    //Set  Center  Button  Text     .
87     rightButton=new IPushButton(MI_RIGHT, //Create Right Push Button       .
88        buttons, buttons, IRectangle(),     //   Parent, Owner=Button Canvas .
89        IPushButton::defaultStyle() |       //   Use Default Styles plus    .
9        IControl::tabStop);                 //   tabStop                     .
91     rightButton->setText(STR_RIGHTB);       //Set Right Button Text        .
92     addExtension(buttons,                   //Add Buttons Canvas           .
93        IFrameWindow::belowClient,          //   below client and           .
94        3 UL);                              //   specify height in pixels   .
95     return true;                            //Return                       .
96 } /  end AHelloWindow :: setupButtons()  /                          //v4
⋮
```

The command events generated by pressing the **Left**, **Center**, and **Right** push buttons are handled by the AHelloWindow::command member function. The help event generated by pressing the **Help** push button is handled by the keysHelpId member function.

Creating a Check Box Control

A *check box* is a square with text beside it that represents a choice. When a user selects the choice, a check mark symbol (√) appears in the check box to indicate that the choice is selected. By selecting the choice again, the user clears the check box. Use a check box to set a choice in a group of choices that are not mutually exclusive.

The ICheckBox class allows you to create and maintain a check box. The selection of a check box is processed by using the ISelectHandler class and adding the handler to either the check box or its owner window.

Refer to Chapter 6, "Adding Events and Event Handlers" on page 67 for information about event handlers.

The following example shows how to create a check box:

1. The text associated with the check box is defined in the resource file as string text:

```
STRINGTABLE
  BEGIN
    STR_CHECK1  ,  "check box one"
    STR_CHECK2  ,  "check box two"
    STR_CHECK3  ,  "check box three"
  END
```

2. Each check box has a corresponding control ID in the .H file, for example:

```
#define STR_CHECK1          x1 1
#define STR_CHECK2          x1 2
#define STR_CHECK3          x1 3
```

3. Three ICheckBox objects are defined and constructed in the canvas called **canvas1**.

```
⋮
ICheckBox checkBox1( WND_CHECK1, &canvas1, &mainwindow );
checkBox1.setText(STR_CHECK1);
ICheckBox checkBox2( WND_CHECK2, &canvas1, &mainwindow );
checkBox2.setText(STR_CHECK2);
ICheckBox checkBox3( WND_CHECK3, &canvas1, &mainwindow );
checkBox3.setText(STR_CHECK3);
⋮
```

Creating a Radio Button Control

A *radio button* is a circle with text beside it. Use radio buttons to display a set of choices from which the user can select one. A group of radio buttons contains at least two radio buttons.

The IRadioButton class allows you to create and manage the radio button control window. The ISelectHandler class processes the selection of a radio button and adds the handler to either the radio button or its owner window.

Refer to Chapter 6, “Adding Events and Event Handlers” on page 67 for information about event handlers.

The following example creates a group of radio buttons with two radio buttons, one labeled **Black**, the other **White**.

1. The text associated with each radio button is defined in the resource file as string text, as follows:

```
⋮
STRINGTABLE
  BEGIN
    STR_BLACK,   "Black"
    STR_WHITE,   "White"
  END
:
⋮
```

2. This code from the .HPP file defines a select handler class, which is inherited from the ISelectHandler class, to handle selecting the radio button.

```
⋮
class MySelectHandler : public ISelectHandler
{
  public:
    MySelectHandler()  ;
  protected:
     selected( IControlEvent& evt );
  private:
⋮
};
```

3. This code is from the .CPP file:

```
 1 IRadioButton radioBtBlack(WND_BLACKBT, &canvas1, &mainWindow);
 2 radioBtBlack.setText(STR_BLACK);
 3 IRadioButton radioBtWhite(WND_WHITEBT, &canvas1, &mainWindow );
 4 radioBtWhite.setText(STR_WHITE);
 5
 6 radioBtBlack.enableGroup().enableTabStop();
 7 radioBtBlack.select();
 8
 9 selectHdr.handleEventsFor(&radioBtBlack);
1  selectHdr.handleEventsFor(&radioBtWhite);
11
12 Boolean MySelectHandler::selected(IControlEvent& evt )
13 {
14 Boolean fProcess= false;
15    switch  (evt.controlId())
16    {
17    case  WND_BLACKBT:
18
:
19          fProcess=  true;
2       break;
21    case  WND_WHITEBT:
22
:
23          fProcess=  true;
24       break;
25    }
26    return  fProcess;
27 }
```

Lines 1 through 4 create two radio buttons in canvas1 using the IRadioButton constructor. WND_BLACKBT and WND_WHITEBT are control IDs defined in the .H file.

Line 6 sets the group styles of the controls using the enableGroup and enableTabStop member functions.

Line 7 uses the ISettingButton::select member function to make the black button the default selection.

Lines 9 and 10 set the select handler to handle events from the selection of the **Black** and the **White** radio buttons.

Lines 12 through 27 provide a selected member function for the select handler class.

Creating a Slider Control

A *slider* is a control that represents a quantity and its relationship to the range of possible values for that quantity. Users can set, display, or modify a value by moving the slider arm.

A slider consists of a slider arm, slider shaft and, optionally, detents, tick marks, tick text, and slider buttons. Figure 5 shows the components of a slider control.

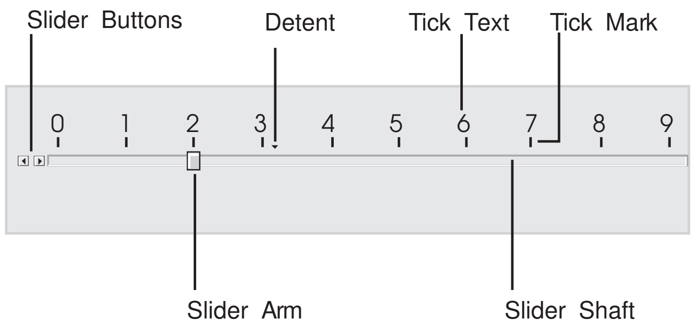


Figure 5. Slider Components

The ISlider class inherits from the IProgressIndicator class, which is a read-only version of the slider control. Typically, a progress indicator is used to display the percentage of a task that is complete by filling in its shaft as the task progresses. Because the user cannot change the value represented by a progress indicator, a slider arm and slider buttons are not provided in a progress indicator.

The following example shows how to create a slider in the constructor of a subclass of IFrameWindow.

```
1 AHelloWindow :: AHelloWindow(unsigned long windowId)
2     : IFrameWindow (
3        IFrameWindow::defaultStyle()
4        | IFrameWindow::minimizedIcon,
5        windowId)
6 {
7     clientCanvas = new IMultiCellCanvas( WND_MCCANVAS, this, this );
8     setClient( clientCanvas );
9
1     infoArea = new IStaticText( WND_INFO, clientCanvas, clientCanvas );
11     infoArea->setAlignment( IStaticText::centerCenter );
12     infoArea->setText( STR_INFO );
13
14     unsigned long numberOfTicks = 1 ;
15     unsigned long tickSpacing=  ;
16
17     mySlider = new ISlider( (unsigned long)ID_SLIDER,
18                             clientCanvas,  clientCanvas,
19                             IRectangle(),
2                              numberOfTicks,
21                             tickSpacing,
22                             ISlider::defaultStyle());
23
24      mySlider->setTickLength(5);
25      for (unsigned long z= ; z<1 ; z++)
26            mySlider->setTickText(z,  (char )(IString(z)));
27
28   unsigned long DetentId = mySlider->addDetent((unsigned long) 1 5) ;
29
3     clientCanvas->addToCell(infoArea, 1, 1);
31     clientCanvas->addToCell(mySlider, 1, 2);
32    clientCanvas->setColumnWidth(1,1  ,true);
33    clientCanvas->setRowHeight(2,1  ,true);
34
35    setFocus();
36    show();
37 }
```

Lines 7 and 8 create a multicell canvas and set it to be the client area of this frame window.

Lines 10, 11, and 12 create a static text control to show a message string.

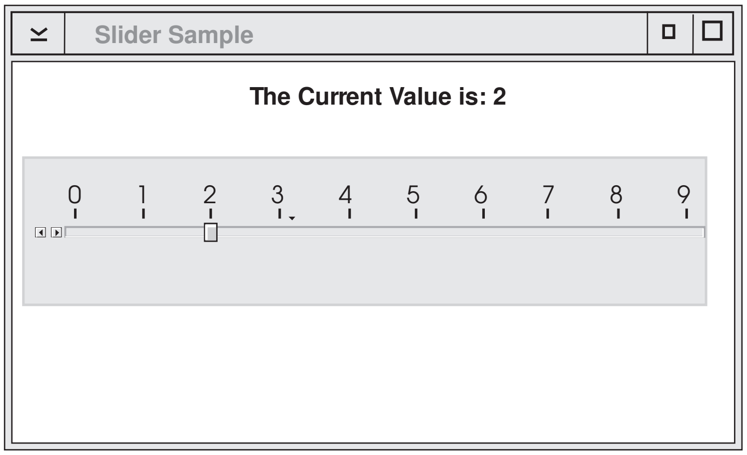
Lines 17 through 22 create the slider with the default style, tick marks, tick text, and detents.

Line 24 uses the setTickLength member function to set the length of all tick marks on the slider scale.

The setTickText member function on lines 25 and 26 sets the text associated with the tick at the specified index.

The addDetent member function on line 28 adds a detent to the slider. The detent is set at the pixel offset from the home position specified, and a unique ID is returned. Use this ID to remove a detent or query its position.

Figure 6 on page 53 shows the slider control generated from this example.



The Current Value is: 2

Figure 6. Slider Sample

The default style of ISlider positions the slider horizontally and centers it in the window with tick marks and text above it. To change the style to a vertical position, add IProgressIndicator::vertical to the default style, as follows:

```
  mySlider = new ISlider( (unsigned long)ID_SLIDER,
                                  clientCanvas,  clientCanvas,
                                  IRectangle(),
                                  numberOfTicks,
                                  tickSpacing,
                                  ISlider::defaultStyle()
                                  | IProgressIndicator::vertical);
```

You can also construct a slider with the tick marks and text below the shaft and place it in any position in the window.

Several member functions inherited from IProgressIndicator provide the slider arm operation. For example, you can use the following statements to return the current slider arm position measured by tick offset, and then modify its position to the result plus one.

```
unsigned long tickNumber= mySlider->armTickOffset ( );
tickNumber++ ;
mySlider->moveArmToTick ( tickNumber );
```

Refer to *IBM C/C++ Tools: User Interface Class Library Reference* for a list of the slider control member functions.

Creating Canvas Controls

A *canvas* is a control that divides the client area. With the canvas classes, you can build windows with multiple child controls that contain fixed-size areas, user-sizeable areas, and scrollable areas. In addition, canvas controls allow you to control tabbing and cursor movement between child controls, providing an alternative to using dialog boxes.

Generally, you build a complex window with a canvas control as the client area. This canvas can contain other canvas controls to provide the required layout.

The canvas classes are:

ISplitCanvas
ISetCanvas
IMultiCellCanvas
IViewPort

Set and multicell canvases automatically size themselves to contain their child windows.

Creating a Split Canvas

A *split canvas* contains two or more child controls. Each child control is placed in a pane. The panes are separated by moveable (default) or fixed split bars. A split canvas can have its split bars oriented vertically or horizontally.

Use a split canvas to contain controls that can be resized to display more information, such as list boxes, containers, MLEs, and notebooks.

Note: Use the noAdjustPosition style on a list box control in a split canvas.

The order in which you declare the child controls determines both their relative position on the split canvas and the order in which tab and cursor keys switch focus between them. For a canvas with vertical split bars, the child controls are arranged with the control that was declared first in the leftmost pane. For a canvas with horizontal split bars, the control that was declared first is placed in the top pane.

The following examples show how to create a window containing two split canvases. Each pane is occupied by a static text control.

1. This code from the header file declares the ASplitCanvas class as a subclass of IFrameWindow.

```
#include <iframe.hpp>                          // IFrameWindow
#include <istattxt.hpp>                        // IStaticText
#include <isplitcv.hpp>                        // ISplitCanvas
class ASplitCanvas : public IFrameWindow
{
  public:
     ASplitCanvas(unsigned long windowId);     // Constructor

  private:
     ISplitCanvas   horzCanvas,                // The canvases will be created
                    vertCanvas;                // in the same order they
     IStaticText    lText,                     // are declared.
                    rText,
                    bText;
};
```

2. This code creates the window.

```
ASplitCanvas :: ASplitCanvas( unsigned long windowId )
  : IFrameWindow( windowId )
  , horzCanvas( WND_CANVAS, this, this )
  , vertCanvas( WND_CANVAS2, &horzCanvas, &horzCanvas )
  , lText( WND_TXTL, &vertCanvas, &vertCanvas )
  , rText( WND_TXTR, &vertCanvas, &vertCanvas )
  , bText( WND_TXTB, &horzCanvas, &horzCanvas )
{

  horzCanvas.setOrientation( ISplitCanvas::horizontalSplit ); //Give the canvas
   setClient( &horzCanvas );              //a horizontal split bar
                                          //and make it the client area

   vertCanvas.setOrientation( ISplitCanvas::verticalSplit );//Give the canvas
                                          //a vertical split bar
   lText.setText(STR_TOPLEFT);            //Set top left static text
   lText.setAlignment( IStaticText::centerCenter );

   rText.setText(STR_TOPRIGHT);           //Set top right static text
  rText.setAlignment( IStaticText::centerCenter );

   bText.setText(STR_BOTTOM);             //Set bottom static text
   bText.setAlignment( IStaticText::centerCenter );

   setFocus().show();                     //Set focus and show window

} /  end ASplitCanvas :: ASplitCanvas(...)  /
```

Lines 1 through 7 create a canvas with horizontal and vertical split bars. The canvases are created in the same order they were declared in the header file. In line 4, the vertical canvas is defined as a child of the horizontal canvas.

Lines 10 and 11 make the horizontal canvas the client area.

Line 14 defines a canvas with vertical split bars.

Lines 16 through 23 set the text for each static control and position the text in each pane.

Figure 7 shows the completed split canvas.

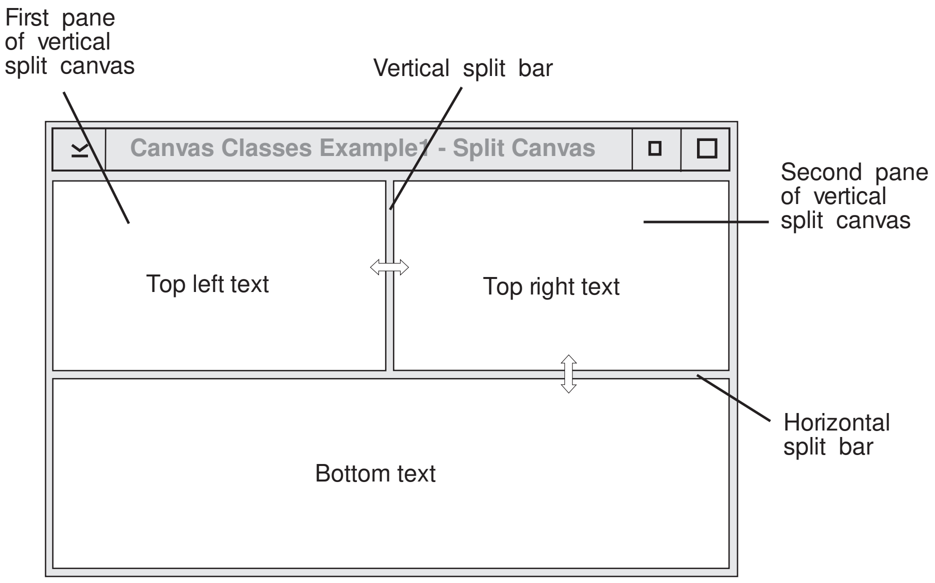

First pane of vertical split canvas
Vertical split bar
Canvas Classes Example1 - Split Canvas
Second pane of vertical split canvas
Top left text
Top right text
Horizontal split bar
Bottom text

Figure 7. Split Canvas Example

Creating a Set Canvas

A *set canvas* arranges its child controls in either rows or columns. The User Interface Class Library uses the term *deck* for either a row or column. You can arrange the decks of a set canvas either horizontally or vertically. The set canvas attempts to place the same number of controls in each deck.

Each deck is created large enough to contain the largest control in the deck. To do this, the canvas calls the minimumSize member function for each child control. For controls that have sizes defined by the text they contain, such as push buttons and radio buttons, this default processing is normally sufficient. However, for a control that does not have a fixed size, such as a notebook, you need to set its minimum size by overriding the calcMinimizeSize member function or by calling the setMinimumSize member function before adding it to the set canvas.

The order in which you create the child controls determines their positions on the set canvas and the order in which tab and cursor keys switch focus between the controls. Several styles are available to control the orientation of the decks and the placement of controls within the decks. You can also alter the spacing between controls and between the decks and the edge of the canvas.

The following examples use a split canvas as a client area. Two set canvases, each with seven radio buttons, are then added to the split canvas.

1. This code from the header file declares the ASetCanvas class as a subclass of IFrameWindow.

```
#include <iframe.hpp>                          // IFrameWindow
#include <istattxt.hpp>                        // IStaticText
#include <iradiobt.hpp>                        // IRadioButton
#include <isetcv.hpp>                          // ISetCanvas
#include <isplitcv.hpp>                        // ISplitCanvas
#define  NUMBER_OF_BUTTONS  14

class ASetCanvas : public IFrameWindow
{
  public:                                      //Define the public information
     ASetCanvas(unsigned long windowId); //Constructor for this class
    ~ASetCanvas();                             //Destructor for this class

  private:                                     //Define private information
    ISplitCanvas     clientCanvas;
    IStaticText      status;
    ISetCanvas         vSetCanvas,
                        hSetCanvas;
    IRadioButton      radiobut[NUMBER_OF_BUTTONS];
};
```

2. This code from the .CPP creates the window.

```
 1 ASetCanvas::ASetCanvas(unsigned long windowId)
 2     : IFrameWindow( windowId )
 3     , clientCanvas( WND_SPLITCANVAS, this, this, IRectangle(),
 4                              ISplitCanvas::defaultStyle() |
 5                              ISplitCanvas::horizontal)
 6     , status(WND_STATUS, &clientCanvas, &clientCanvas)
 7     , vSetCanvas(WND_VSETCANVAS, &clientCanvas, &clientCanvas)
 8     , hSetCanvas(WND_HSETCANVAS, &clientCanvas, &clientCanvas)
 9 {
1
11      setClient(&clientCanvas);                              //Make split canvas the client area
12
13      status.setAlignment(IStaticText::centerCenter);//Set alignment of status area text
14
15     vSetCanvas.setDeckOrientation(ISetCanvas::vertical);
16      vSetCanvas.setDeckCount(3);                              //Create 3 vertical decks in top canvas
17
18     hSetCanvas.setDeckOrientation(ISetCanvas::horizontal);
19      hSetCanvas.setDeckCount(3);                              //Create 3 horizontal decks in bottom canvas
2      hSetCanvas.setPad(ISize(1 ,1 ));                        //Set some space around buttons
 :
```

Lines 1 through 8 create the set canvas.

Line 11 makes the split canvas the client area.

Line 13 adds a static text control to the top pane of the split canvas.

Lines 15 through 20 add two set canvases to the bottom panes.

Figure 8 shows the set canvas created using this code.

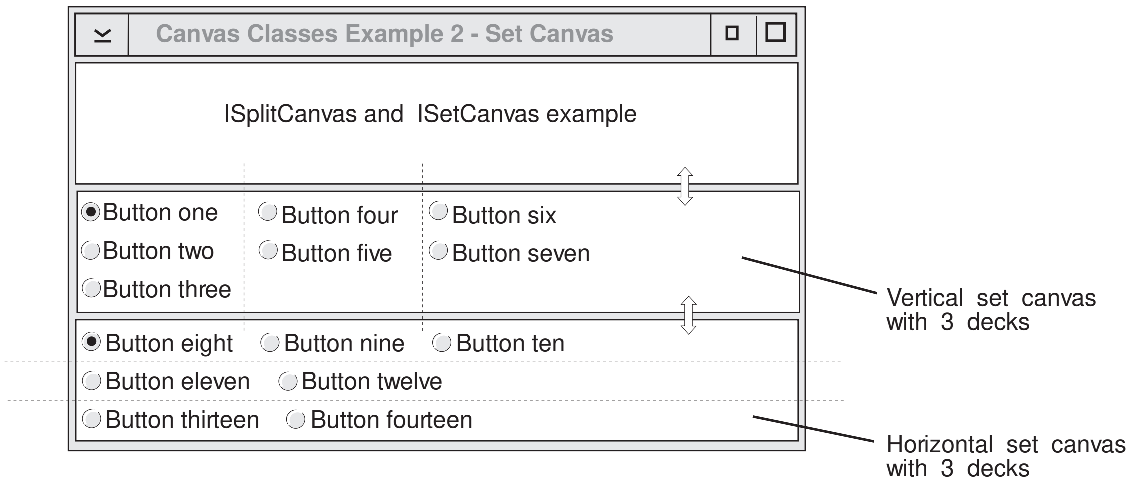


Figure 8. Set Canvas Example

Creating a Multicell Canvas

A *multicell canvas* consists of a grid of rows and columns. You place child controls on the canvas by specifying the starting cell and the number of contiguous rows and columns that they can span. You can refer to cells in the grid by the column and row value. The top left cell coordinate is (1,1).

The actual number of rows and columns in the canvas is the highest row and column value used. In the following example, a radio button is placed at (4,5) and a push button at (2,7). Therefore the canvas will have 4 columns and 7 rows.

The initial size of a row or column is determined by the size of the largest control in that row or column. By default, the row and column sizes are fixed. You can make the rows and columns expandable, in which case sizing the canvas also sizes them.

You can also leave rows and columns empty to provide spacing between child controls. If you do this, specify the size of the empty rows and columns.

The following examples show you how to create a window containing a multicell canvas. The canvas contains two check boxes, two radio buttons, three static text controls, and a one push button.

1. This code from the header file declares the AMultiCellCanvas class as a subclass of IFrameWindow class.

```
#include <iframe.hpp>                          // IFrameWindow
#include <istattxt.hpp>                        // IStaticText
#include <ipushbut.hpp>                        // IPushButton
#include <iradiobt.hpp>                        // IRadioButton
#include <icheckbx.hpp>                        // ICheckBox
#include <imcelcv.hpp>                         // IMultiCellCanvas
class AMultiCellCanvas : public IFrameWindow
{
  public:
    AMultiCellCanvas(unsigned long windowId);

  private:
    IMultiCellCanvas     clientCanvas;
    IStaticText          status,
                         title1,
                         title2;

    ICheckBox            check1,
                         check2;
    IRadioButton         radio1,
                         radio2;
    IPushButton          pushButton;
};
```

2. This code creates the window.

```
AMultiCellCanvas::AMultiCellCanvas(unsigned long windowId)
  : IFrameWindow(windowId)
  , clientCanvas( WND_MCCANVAS, this, this )
  , status( WND_STATUS, &clientCanvas, &clientCanvas )
  , title1( WND_TITLE1, &clientCanvas, &clientCanvas )
  , title2( WND_TITLE2, &clientCanvas, &clientCanvas )
  , check1( WND_CHECK1, &clientCanvas, &clientCanvas )
  , check2( WND_CHECK2, &clientCanvas, &clientCanvas )
  , radio1( WND_RADIO1, &clientCanvas, &clientCanvas )
  , radio2( WND_RADIO2, &clientCanvas, &clientCanvas )
  , pushButton( WND_PUSHBUT, &clientCanvas, &clientCanvas )
{

   setClient( &clientCanvas );           // make multicell canvas the client area
   status.setAlignment( IStaticText::centerCenter );// set status area text
   status.setText( STR_STATUS );

 title1.setAlignment( IStaticText::centerLeft );       // set text and alignment
   title1.setText( STR_TITLE1 );

 title2.setAlignment( IStaticText::centerLeft );       // set text and alignment
 title2.setText( STR_TITLE2 );

   check1.setText( STR_CHECK1 );                              // set check box text
   check2.setText( STR_CHECK2 );
   radio1.setText( STR_RADIO1 );                              // set radio button text
   radio2.setText( STR_RADIO2 );

   pushButton.setText( STR_PUSHBUT );

   radio2.select();                                    // preselect one radio button
   check1.enableGroup().enableTabStop();// set tabStop and group styles
  radio1.enableGroup().enableTabStop();
  pushButton.enableGroup().enableTabStop();

   clientCanvas.addToCell(&status    , 1, 1, 4, 1);         // add controls to canvas.
   clientCanvas.addToCell(&title1    , 1, 3, 2, 1);         // the canvas runs from
   clientCanvas.addToCell(&title2    , 3, 3, 2, 1);         // 1,1 to 4,7
   clientCanvas.addToCell(&check1    , 2, 4);                   // only one row and
  clientCanvas.addToCell(&check2    , 2, 5);                    // one column are
   clientCanvas.addToCell(&radio1    , 4, 4);                   // expandable, as this
   clientCanvas.addToCell(&radio2    , 4, 5);                   // allows the canvas to
   clientCanvas.addToCell(&pushButton , 2, 7);              // fill the client area.

   clientCanvas.setRowHeight(2, 2 , true); // make row 2 2  pixels high and expandable

   clientCanvas.setRowHeight(6, 4 ); // make row 6 4  pixels high

   clientCanvas.setColumnWidth(4, 4 , true);    // make column 4 4  pixels wide and expandable

   check1.setFocus();                                     // set focus to first check box
   show();                                                   // show main window

} /  end AMultiCellCanvas :: AMultiCellCanvas(...)  /
```

Lines 1 through 11 create a multicell canvas.

Line 14 makes it the client area.

Lines 36 through 43 place the other controls on the canvas using the addToCell member function.

Lines 45 through 49 set the sizes for rows 2 and 6 and column 4. Row 2 and column 4 are expandable.

Figure 9 shows the completed multicell canvas.

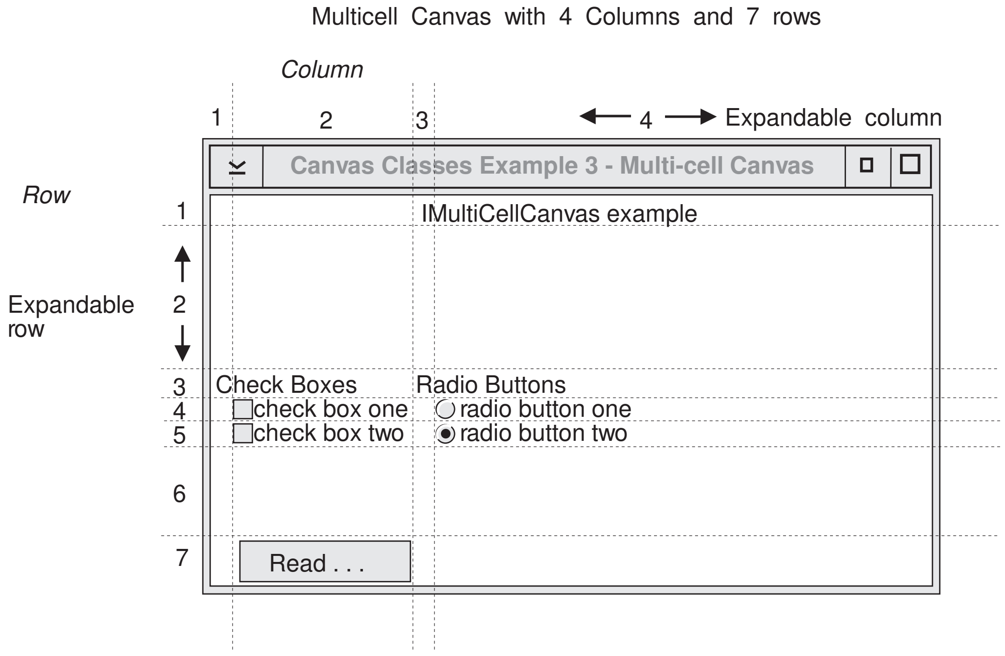


Figure 9. Multicell Canvas Example with 4 Columns and 7 Rows

Creating a Viewport

A *viewport* canvas has only one child control. The size of the child control is fixed. If the viewport is smaller than the child control, scroll bars are added to the viewport. The user can then use the scroll bars to view different parts of the child control.

If you need more than one control in a viewport, place the controls into another type of canvas, which you can then make the child of the viewport.

The following examples show how to create a window containing a viewport. The viewport has a single bit-map control inside it.

1. This code from the header file declares the AViewPort class as a subclass of IFrameWindow class.

```
#include <iframe.hpp>                          // IFrameWindow
#include <ivport.hpp>                          // IViewPort
#include <ibmpctl.hpp>                         // IBitmapControl
class AViewPort : public IFrameWindow
{
  public:                                      // define the public information
    AViewPort(unsigned long windowId);         // constructor for this class

  private:                                     // define private information
    IViewPort          clientViewPort;
    IBitmapControl   bitmap;
};
```

2. This code creates the viewport control and sets it as the client area. The bit-map control is then made a child of the viewport.

```
#include <ireslib.hpp>                         // IResourceId class
AViewPort :: AViewPort(unsigned long windowId)
   : IFrameWindow( windowId )
   , clientViewPort( WND_VIEWPORT, this, this )
   , bitmap( WND_BITMAP, &clientViewPort, &clientViewPort
                , IResourceId(BMP_ID) )
{
   setClient( &clientViewPort );               // make viewport the client
   setFocus().show();                          // set focus and show window
} /  end AViewPort :: AViewPort(...)  /
```

Figure 10 shows the viewport canvas created using this code.

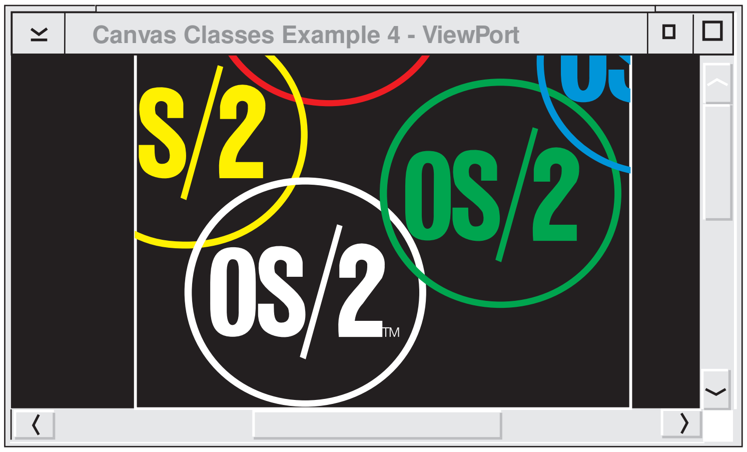


Figure 10. Viewport Canvas Example

Adding Styles

A *style* affects the appearance and behavior of a window. Each window class has styles that are encapsulated in style objects. Each style object operates within the scope of the window class that it affects.

Generic styles are defined in IWindow and IControl. Classes derived from IWindow and IControl can combine their own styles with those of IWindow and IControl.

Each window class maintains its own default style object. You can access this default style object using the static member function defaultStyle and then set it using the static member function setDefaultStyle. Each window class also maintains a style object called classDefaultStyle that corresponds to the initial setting of defaultStyle.

All window classes provide one or more constructors that accept a style object as one parameter. You can only construct a style object from existing style objects. These style object are only used by window constructors. The style of a window can subsequently be changed and queried using the window class member functions. Also some styles cannot change after a window has been created, in which case no member function is provided to change the style.

Combining Style Objects

You can combine style objects using the bitwise OR (|) operator.

The following example creates a list box style object that you can use to construct a multiple-selection list box:

```
IListBox::Style lbStyle = IListBox::defaultStyle()
                          | IListBox::multipleSelect;
```

Removing a Style

You can remove styles from a style object by creating a negated-style object using the negation (˜) bitwise operator and then using the bitwise AND (&) operator.

The following example creates a list box style object that you can use to construct a list box without a horizontal scroll bar:

```
IListBox::Style lbStyle = IListBox::defaultStyle()
                          & ˜IListBox::horizontalScroll;
```

Setting Window Styles

You can create a window with a specific style in the following ways:

- Create a window using a constructor that accepts the style as a parameter. The following three examples illustrate this method.

  This example shows how to create an entry field control with a style that is a combination of styles from IWindow, IControl, and IEntryField.

```
IEntryField  entryField( ID_EF1, parent, owner,
                         IRectangle(1 , 1 , 1  , 2 ),
                         IWindow::visible      |
                         IControl::tabStop     |
                         IControl::group       |
                         IEntryField::margin   |
                         IEntryField::autoScroll  );
```

Alternatively, you can explicitly construct the style object and pass it as a parameter:

```
IEntryField::Style efStyle = IWindow::visible      |
                             IControl::tabStop     |
                             IControl::group       |
                             IEntryField::margin   |
                             IEntryField::autoScroll  ;
IEntryField  entryField( ID_EF1, parent, owner,
                         IRectangle(1 , 1 , 1  , 2 ),
                         efStyle  );
```

The default style object can also be accessed using the static member function defaultStyle. This simplifies the preceding example to:

```
IEntryField  entryField( ID_EF1, parent, owner,
                         IRectangle(1 , 1 , 1  , 2 ),
                         IEntryField::defaultStyle()   |
                         IControl::tabStop             |
                         IControl::group  );
```

Use the static member function setDefaultStyle to set the default style and then construct the window. For example:

```
IEntryField::Style efStyle = IEntryField::defaultStyle()   |
                             IControl::tabStop             |
                             IControl::group               ;
IEntryField::setDefaultStyle(efStyle);
IEntryField  entryField( ID_EF1, parent, owner,
                         IRectangle(1 , 1 , 1  , 2 ) );
```

Create a window with the default style and change it using member functions of the window. The example now becomes:

```
IEntryField  entryField( ID_EF1, parent, owner,
                         IRectangle(1 , 1 , 1  , 2 ) );
entryField.enableGroup();            // member function of IControl
entryField.enableTabStop();          // member function of IControl
entryField.enableAutoScroll();       // member function of IEntryField
```

For a complete list of available styles, see the *IBM C/C++ Tools: User Interface Class Library Reference*.

Creating Cursor Classes

The *cursor classes* provide member functions to move through the items, either forward or backward, and to add items after the cursor position. Window classes that can contain one or more items generally provide a nested cursor class.

A cursor must be in a valid state to access the items in a list. A cursor is generally created in an invalid state. Any cursor function that causes the cursor to point to an item in the list validates the cursor. For example, the function setToFirst causes the cursor to be valid if there are items in the list. If the contents of the list that the cursor is iterating through changes by the addition or removal of items, the cursor becomes invalid and cannot be used to access items in the list until it is validated again by a function that points the cursor at a valid item.

In some cases, you may want to construct a cursor that iterates through items with a particular property only. For example, the constructor for a list box cursor can have a second parameter that determines whether the cursor returns all items in the list box or just the selected items with that property.

The following example shows how to iterate through all selected items in a multiple-selection list box:

```
  IListBox listbox( ID_LB, parent, owner, IRectangle(),
                             IListBox::defaultStyle() | IListBox::multipleSelect );
/   ...  add  items  to  listbox   ... /
 IListBox::Cursor  lbCursor(listbox);
 for (lbCursor.setToFirst(); lbCursor.isValid(); lbCursor.setToNext())
     {
      IString    str(listbox.elementAt(lbCursor));          //Return item at cursor
      unsigned long ul = lbCursor.asIndex();                //Return zero-based index
      / ... process string or index ...  /
     }
```

Chapter 6. Adding Events and Event Handlers

The User Interface Class Library uses events and event handlers to encapsulate the message architecture of OS/2 Presentation Manager (PM) in an object-oriented way. The User Interface Class Library reserves message IDs beginning at 0xFFE0. If you use the User Interface Class Library, define user messages only in the range of WM_USER (0x1000) through 0xFFDF. Figure 11 shows the relationships between window, event, and handler classes.

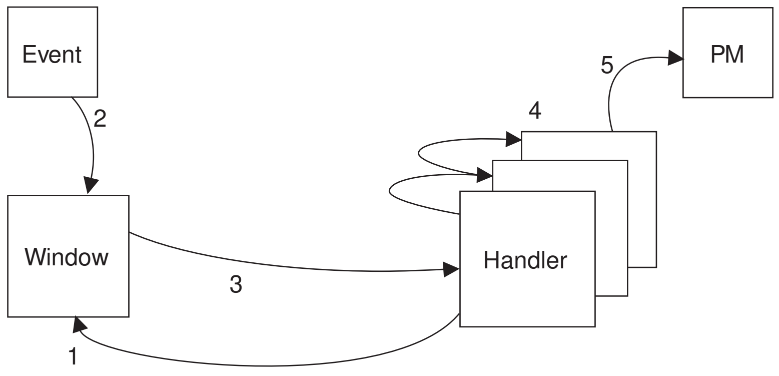


Figure 11. Relationship of Window, Event, and Handlers to Presentation Manager

1. Handlers are registered with the window.
2. PM messages are encapsulated in event objects, which are passed to the window or control that had the event.
3. The window then invokes the handlers attached to it, passing the event object as a parameter.
4. The handlers are called sequentially with the most recently added handler invoked first. A handler indicates when processing for the event is complete by returning a Boolean value of true.
5. If none of the handlers can process the event, it is passed to the default PM window procedure for the window.

The distinction between window classes and handler classes allows you to separate the event handling logic from the rest of the application. This enables reuse of this logic. For example, you can reuse a handler to verify telephone numbers wherever an entry field accepts telephone numbers.

Processing Events Using Handlers

Each handler class has one or more virtual functions that are called to process the event. When an application processes events, it normally subclasses a handler class and overrides the virtual function to provide its own application-specific logic.

Note: Ensure that handlers return from virtual functions within 1/10 second to avoid locking up the system by delaying the PM message processing.

Figure 12 shows how the ICommandHandler works. All handler classes contain a dispatchHandlerEvent function to determine whether the handler needs to process the event or return it. If the event needs processing, it creates the appropriate event object and calls the appropriate virtual function to process the event.

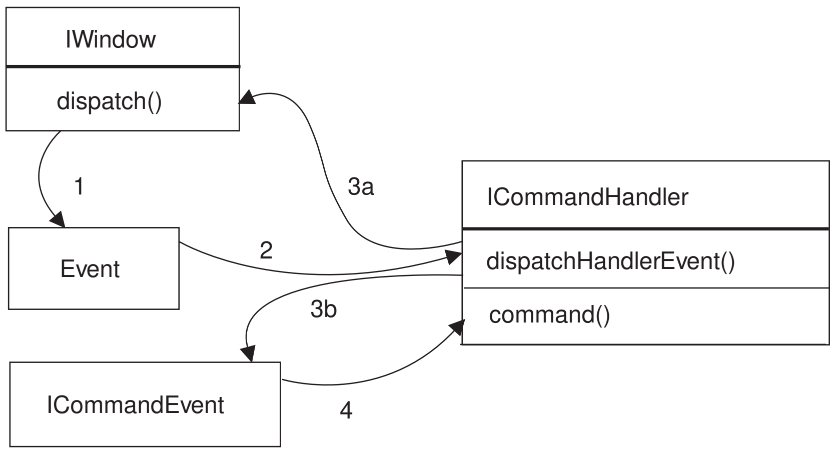


Figure 12. Processing within the ICommandHandler

Figure 13 on page 69 presents some common events for which you can provide handlers. It relates the type of event, the handler for that event, and the member function in the handler class that the application must override in order to provide its own logic.

The *IBM C/C++ Tools: User Interface Class Library Reference* contains descriptions of all handler classes and member functions.

Figure 13. Common Events and Their Handlers

Event Generated by	Event Class	Handler Class	Member Function	PM Message
Command event by menu selection, push button, or accelerator key	ICommandEvent	ICommandHandler	command	WM_COMMAND
System command event by menu selection, push button, or accelerator key	ICommandEvent	ICommandHandler	systemCommand	WM_SYSCOMMAND
Edit event by entry field, combination box, MLE, or slider	IControlEvent	IEditHandler	edit	WM_CONTROL
Gain focus or lose focus by entry field, combination box, MLE, slider, container, or spin button	IControlEvent	IFocusHandler	getFocus, lostFocus	WM_CONTROL
Keyboard entry by entry field, combination box, MLE, or other input focus control	IKeyboardEvent	IKeyboardHandler	keyPress, key, scanCodeKeyPress, virtualKeyPress, characterKeyPress	WM_CHAR
Paint area event by all controls	IPaintEvent	IPaintHandler	paintWindow	WM_PAINT
Resize event by all controls	IResizeEvent	IResizeHandler	windowResize	WM_WINDOWPOSCHANGED
Item selected by List box, combination box, container, check box, or radio button	IControlEvent	ISelectHandler	selected	WM_CONTROL
Enter pressed when item selected, or double-click on item by list box, combination box, or container	IControlEvent	ISelectHandler	enter	WM_CONTROL
Pop-up menu requested by mouse button or keyboard	IMenuEvent	IMenuHandler	makePopUpMenu	WM_CONTEXTMENU
Menu about to be shown by pull-down menu or pop-up menu	IMenuEvent	IMenuHandler	menuShowing	WM_INITMENU
Menu item highlighted and ready to be selected by mouse or keyboard	IMenuEvent	IMenuHandler	menuSelected	WM_MENUSELECT
Menu removed by mouse or Esc key	IMenuEvent	IMenuHandler	menuEnded	WM_MENUEND
Container item context menu requested by container	IMenuEvent	ICnrMenuHandler	makePopupMenu	WM_CONTROL

Extracting Information Using Events

The IEvent class acts as the base class for the more specialized event classes. It provides general member functions to extract the message ID and message parameters. The subclasses of IEvent generally add more specialized functions for extracting information specific to that type of event.

Figure 14 shows some common event classes and some of the functions they contain to extract event information.

Figure 14. Event Classes and Accessor Functions

Event Class	Accessor Function	Description of Return Value
IEvent	window	The IWindow object pointer
IEvent	handle	IWindowHandle of the window
IEvent	eventId	ID of the event
IEvent	parameter1	IEventData containing first event parameter
IEvent	parameter2	IEventData containing second event parameter
ICommandEvent	source	An enumeration type that gives the type of control
ICommandEvent	commandId	The ID of the command that caused the event
IControlEvent	controlId	The ID of the control that caused the event
IControlEvent	control	Pointer to the control that caused the event
IKeyboardEvent	character	Single-byte character code (exception thrown if DBCS)
IKeyboardEvent	mixedCharacter	IString containing character (can be DBCS)
IKeyboardEvent	virtualKey	An enumeration type that gives the virtual key event
IMenuEvent	menuItemId	The ID of the selected menu Item
IMenuEvent	mousePosition	Position of mouse at the time the event occurred
IPaintEvent	presSpaceHandle	The handle of the presentation space to use for any drawing
IPaintEvent	rect	The screen rectangle that needs updating

The IEvent class provides a member function, setResult, for those events that require a value to be returned.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for a complete list of event classes and member functions.

Writing an Event Handler

In general, writing an event handler can be divided into the following steps:

1. Determine which handler class processes the event.
2. Subclass the handler class and override the event handling functions.
3. Create an instance of your subclass.
4. Attach the instance to the window.

The Hello World application in Part 4, “Learning from the Sample Application” on page 143, has several event handlers. The following code is from Version 3 of the Hello World application and shows how to process user menu selection.

1. Determine which handler class processes the event.

   When the user selects a menu, a command message is sent to the frame window and the client window. The handler class for this type of event is ICommandHandler.

2. Subclass the handler class and override the event handling function.

   The Hello World application uses multiple inheritance to provide a class named AHelloWindow that inherits from both IFrameWindow and ICommandHandler. The class ICommandHandler has a virtual function command to process command events. The class AHelloWindow overrides this function to provide its own command event handling.

   The following example, taken from AHELLOW3.HPP, shows the class declaration of AHelloWindow.

```
⋮
//
class AHelloWindow : public IFrameWindow,
                     public  ICommandHandler                      //v3
{
  public:                              //Define the Public Information
    AHelloWindow(unsigned long windowId); //Constructor for this class

 protected:                          //Define Protected Member           v3
    Boolean command(ICommandEvent& cmdEvent);                       //v3
⋮
};
```

The definition of the command function is taken from AHELLOW3.CPP. The ID of the menu item is extracted from the command event object using the commandId member function.

```
//
// AHelloWindow :: command                                                  .
//     Handle menu commands                                                 .
//                                         .
Boolean AHelloWindow :: command(ICommandEvent & cmdEvent)                 // .
{                                                                          //v3
    switch (cmdEvent.commandId()) {          //Get command id             v3

      case MI_CENTER:                          //Code to Process Center Command Item v3
:
        return(true);                       //Return command processed       .
        break;                                //                              v3

      case MI_LEFT:                           //Code to Process Left Command Item   v3
:
        return(true);                       //Return command processed       .
        break;                                //                              v3

      case MI_RIGHT:                     //Code to Process Right Command Item   v3
:
        return(true);                       //Return command processed       .
        break;                                //                              v3

    } /  end switch  /                     //                               v3

    return(false);                         //Return command not processed   v3
} /  end HelloWindow :: command(...)  /                                   //v3
```

3. Create an instance of your subclass.

   Because the window is its own command event handler, creating the window creates an instance of the handler. If a separate handler class has been defined, you have to create an instance of it. Normally, do this in the constructor for the window.

4. Attach the instance to the window.

   The base class IHandler provides a member function handleEventsFor to attach a handler to a window. In the Hello World application, AHELLOW.CPP, the handler is attached on line 48.

```
⋮
22 //
23 // AHelloWindow :: AHelloWindow - Constructor for our main window
24 //
25 AHelloWindow :: AHelloWindow(unsigned long windowId)
26     : IFrameWindow (                              //Call IFrameWindow constructor        v2
27        IFrameWindow::defaultStyle()              //   Use default plus                  v2
28        | IFrameWindow::minimizedIcon,            //   Get Minimized Icon from   RC file  v2
29        windowId)                                 //   Main Window ID
3  {
31     hello=new IStaticText(WND_HELLO,             //Create Static Text Control
32        this, this);                              //   Pass in myself as parent & owner
⋮
47
48     handleEventsFor(this);                       //Set self as event handler (commands)v3
49     menuBar=new IMenuBar(WND_MAIN, this); //Create Menu Bar for main window          .
5     menuBar->checkItem(MI_CENTER);                //Place Check on Center Menu Item      v3
51
52     sizeTo(ISize(4  ,3  ));                      //Set the size of main window          v2
53     setFocus();                                  //Set focus to main window
54     show();                                      //Set to show main window
55
56 } /  end AHelloWindow :: AHelloWindow(...)  /
```

Chapter 7. Managing Character Data

The data type classes simulate basic data types, such as strings, points, and rectangles. These classes hide the structure of the data while providing the capability to access and alter the data. In addition, a set of handle classes provide access to window or application-specific handles.

This chapter discusses two data type classes, IString and IFont.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for more information on the data type classes.

Managing Data Using IString

The IString class contains member functions that enable you to manipulate and manage data.

With the IString class, you can:

- Perform stream I/O
- Query string characteristics
- Test the contents of the string
- Compare strings using overloaded operators
- Convert string
- Edit strings
- Manipulate strings using concatenation, copy, and alignment operators

Performing Stream I/O

You can read and write an IString instance using the operators << and >>. The highlighted lines from an application called **t1** show how to do this:

```
#include <istring.hpp>
#include <iostream.h>
void main()
{
IString
    s1="Enter a letter = ",
   s2;
  cout << s1 << endl;
  cin >> s2 ;
} /  end main  /
```

Here is an example of running application **t1**.

```
[C:\]t1
Enter a letter =
a
[C:\]
```

Querying String Characteristics

The IString class provides accessor functions that you can use to analyze various elements of a string. An *accessor* returns information about the elements of a data type.

The following table lists the accessors for IString:

Accessor	**Returns**
charType	Type of the character at the argument index.
length	Length of the string, not counting the terminating null character.
size	Size of the string.
subString	One part of the string. subString has three parameters: startPos, length, and padCharacter.
operator[]	Value of the *n*th position in the string.

In the following code from an application called **t2**, the highlighted text shows you how to use accessors:

```
#include <istring.hpp>
#include <iostream.h>
void main()
{
  IString
     s1("string"),
     s2;
   cout << " The size of s1 is " << s1.size() << endl;
   cout << "And the 5th element is " << s1[5] << endl;
   cout << "And the first three characters are " << s1.subString(1,3) << endl;
} /  end main  /
```

Here is an example of running application **t2**:

```
[C:\]t2
The size of s1 is 6
And the 5th element is n
And the first three characters are str
[C:\]
```

Testing the Contents of a String

Use the following IString member functions to test for the conditions or characteristics of a string.

Member Function	Returns True if
isASCII	All characters are 0x00–0x7F
isAlphabetic	All characters are A–Z or a–z
isAlphanumeric	All characters are A–Z, a–z, or 0–9
isBinaryDigits	All characters are either 0 or 1
isControl	All characters are 0x00–0x1F or 0x7F
isDBCS	All characters are double-byte characters
isDigits	All characters are 0–9 only
isGraphics	All characters are 0x21–0x7E
isHexDigits	All characters are 0–0, A–F, or a–f
isLowerCase	All characters are a–z only
isPrintable	All characters are 0x20–0x7E
isPunctuation	None of the characters are white space, control characters, or alphanumeric
isSBCS	All of the characters are single-byte characters
isUpperCase	All characters are A–Z only
isValidDBCS	No DBCS characters have a second byte of 0
isWhiteSpace	All characters are 0x09–0x0D or 0x20

In the following code from an application called **t3**, the highlighted text shows you how to test an input string and return the result.

```
#include <istring.hpp>
#include <iostream.h>
void main()
{
   IString  s,s1;

    cin >> s;
   if  (s.isDigits())
    {cout << "The string " << s << " contains only numbers." << endl;}
   else
   {
      if  (s.isAlphabetic())
      {
       if  (s.isLowerCase())
        { cout << "The string " << s << " contains only lowercase characters." << endl;}
       else
         if  (s.isUpperCase())
          {cout << "The string " << s << " contains only uppercase characters." << endl;}
         else
          {cout << "The string " << s << " contains only mixed alphabetic characters." << endl;}
      }

      else
      {
       if  (s.isAlphanumeric())
            { cout << "The string " << s << " contains only alphanumeric characters." << endl;}
      else
        { cout << "The string " << s << " contains some unusual characters." << endl; }
      }
   }
} /  end main  /
```

Here are examples of running the **t3** application:

```
[C:\]t3
ABC
The string ABC contains only uppercase characters.

[C:\]t3
abc
The string abc contains only lowercase characters.

[C:\]t3
Abc
The string Abc contains only mixed alphabetic characters.

[C:\]t3
12a
The string 12a contains only alphanumeric characters.

[C:\]t3
#@%
The string #@% contains some unusual characters.

[C:\]
```

Comparing Strings

Use the comparison operators included in the IString class to compare one IString either to another IString or to a literal character string. The following table lists the comparison operators for IString:

Operator	Returns
==	True if the strings are identical
!=	True if the strings are not identical
<	True if the first string is less than the second, applying the standard collating scheme (memcmp)
<=	Equivalent to (string1 < string2)||(string1 == string2)
>	Equivalent to !(string1 <= string2)
>=	Equivalent to !(string1 < string2)

In the following code from an application called **t4**, you see how to compare two strings to determine if they are equal:

```
#include <istring.hpp>
#include <iostream.h>
void main()
{

IString s("Name"), s1;

cin >> s1;

    if (s1 == s)
      cout << s1 << " is equal to " << s << endl;
    else
      if (s1 != "name")
        cout << s1 << " is not expected " << endl;
    else
      cout << " The first letter must be capitalized. " << endl;
} / end main /
```

Here are examples of running **t4**:

```
[C:\]t4
12
12 is not expected

[C:\]t4
name
 The first letter must be capitalized.

[C:\]t4
Name
Name is equal to Name

[C:\]
```

Converting Strings

You can use the member functions in the IString class to convert strings into other values.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for conversion member functions.

In the following code from an application called **t5**, you see how to convert a string into a long integer:

```
#include <istring.hpp>
#include <iostream.h>
void main()
{

IString s1("111 1");
int n1;

 n1=s1.asInt();
 n1+=1;
 s1=n1;
 cout << s1 << endl;

} /  end main  /
```

Here is an example of running **t5**:

```
[C:\]t5
111 2

[C:\]
```

Editing Strings

Use member functions in the IString class to modify and align text strings. The following table lists those member functions for IString:

Member Function	Result
IString::change	Changes occurrences of an argument to an argument replacement string
IString::center	Centers the receiver within a string of a specified length
leftJustify	Left justifies the receiver within a string of a specified length
rightJustify	Right justifies the receiver within a string of a specified length
upperCase	Changes all lowercase letters in the receiver to uppercase
lowerCase	Changes all uppercase letters in the receiver to lowercase

In the following example from an application called **t6**, you see how to replace one string with another, change the text alignment, and translate text from lowercase to uppercase:

```
#include <istring.hpp>
#include <iostream.h>
void main()
{


IString s("text"),s1,s2,s3,s4, s5("this is a test");
   s4 = s3 = s2 = s1 = s;
   cout << " | " << s1.center(1 ,'+') << " | " << endl;
   cout << " | " << s2.leftJustify(1 ,'<') << " | " << endl;
   cout << " | " << s3.rightJustify(1 ,'>') << " | " << endl;
   cout << " | " << s4.upperCase().center(1 ,' ') << " | " << endl;
   cout << " | " << s4.upperCase().center(1 ,' ') << " | " << endl;
   cout << " | " << s5.change("This","These").change("is","are")
   .change("test","tests").change("a ","",8,1) << " | " << endl;

} /  end main  /
```

Here is an example of running **t6**:

```
[C:\]t6
 | +++text+++ |
 | text<<<<<< |
 | >>>>>>text |
  |      TEXT      |
  |      TEXT      |
 | These are tests |

[C:\]
```

Manipulating Strings

Use operators in the IString class to manipulate text in a variety of ways.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for IString member functions.

In the following example from an application called **t7**, you see how how to concatenate two strings:

```
#include <istring.hpp>
#include <iostream.h>
void main()
{

IString s("1"), s1("2"), s2;

    s2 = s + s1;
    if (s2 != "12") cout << " Something is wrong " << endl;
    else cout << " I expected that " << endl;

} /  end main  /
```

Here is an example of running **t7**:

```
[C:\]t7
 I expected that

[C:\]
```

Setting and Changing Fonts

The IFont class contains member functions to set and change the characteristics of the fonts you use in your applications.

You can set the font of almost all objects using the member function setFont, which is defined in the IControl class.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for more information on the IFont class.

The highlighted lines in the following code show you how to create a font with a specific name and point size, and then how to change the point size of different text strings:

```
#include <ifont.hpp>
⋮
  IFont  Fonts("Helv",8);
⋮
   title1 = new IStaticText( WND_TITLE1, clientCanvas, clientCanvas );
   title1->setAlignment( IStaticText::centerLeft );
   title1->setText( STR_TITLE1 );
  Fonts.setPointSize(12);
  title1->setFont(Fonts);
⋮
   check1 = new ICheckBox( WND_CHECK1, clientCanvas, clientCanvas );
   check1->setText( STR_CHECK1 );
  Fonts.setPointSize(2 );
  check1->setFont(Fonts);
⋮
```

To test the font statements, include the highlighted lines in the AMCELCV.CPP file.

Part 3. Programming Advanced Features

Chapter 8. Creating Additional Controls

This chapter covers the following controls:

- Multiple-line edit fields
- Containers
- Notebooks

Viewing and Editing Multiple-Line Edit Fields

A *multiple-line edit field* (MLE) control enables users to view and edit multiple lines of text. Use the IMultiLineEdit class to create an MLE field control. The member functions of the IMultiLineEdit class enable you to display text files with horizontal and vertical scrolling, read a file into and save it from an MLE, or perform basic editing tasks, for example cut, copy, and clear.

Creating an MLE

To create an instance of the IMultiLineEdit class, include the ID of a specified MLE, the parent and owner windows, an IRectangle instance, and one or more styles.

Styles define such functions as scrolling text, wrapping words, adding a border, and making the field read-only.

Refer to *IBM C/C++ Tools: User Interface Class Library Reference* for further information about the IMultiLineEdit class and its styles.

Use the following example to create an instance of an MLE that uses the default style and includes horizontal scrolling. DID_MLE is the ID of an MLE defined in the resource file.

```
⋮
AEditorWindow :: AEditorWindow(unsigned long windowId)
   : IFrameWindow (                              //Call IFrameWindow constructor
      IFrameWindow::defaultStyle()             //Use default plus
      | IFrameWindow::minimizedIcon,           //Get minimized icon from RC file
      windowId)                                  //Main window ID
{
   mtextfield = new IMultiLineEdit( DID_MLE, this, this,IRectangle(),
                                    IMultiLineEdit::defaultStyle()  |
                                    IMultiLineEdit::horzScroll);
⋮
  setClient(mtextfield);
⋮
```

The highlighted lines create an instance of the IMultiLineEdit class. When the IMultiLineEdit object is set in a client window, it looks like Figure 15 on page 87.

Loading and Saving a File

The following member functions from the IMultiLineEdit class manage files and MLEs:

Member Function	Use To
importFromFile	Load a file into an MLE
exportToFile	Save from an MLE
exportSelectedTextToFile	Save marked text in an MLE into a file

Refer to *IBM C/C++ Tools: User Interface Class Library Reference* for descriptions of these member functions.

The highlighted line in following example illustrates how to load a file into an MLE:

```
⋮
  filename=fd->fileName();                 //
   if (filename.size())                    //If the name has been specified,
   {                                              //load the file into mtextfield.
 mtextfield->importFromFile(filename.asString());
 mtextfield->setCursorAtLine( );
   } / endif /                             //
} / endif /                                //
⋮
```

Positioning the Cursor

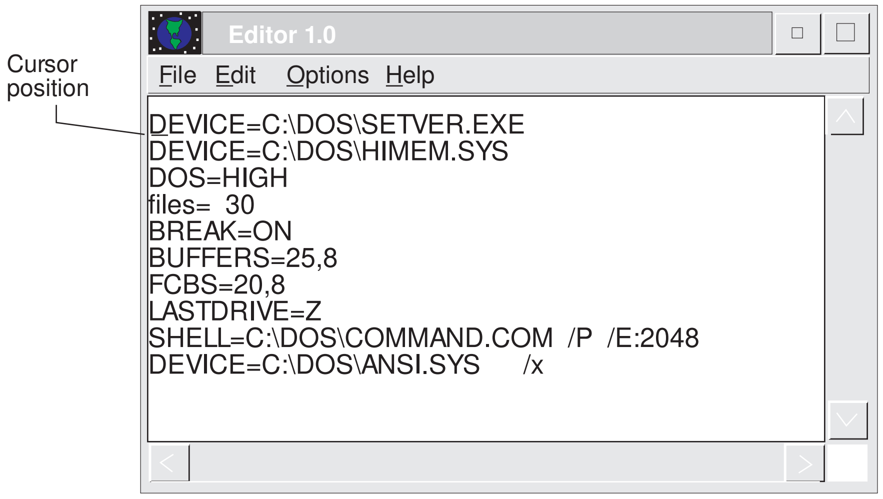
You can position a cursor on a specific line of an MLE or in a specific character position, add to or remove lines from an MLE, or ask for the number of lines in an MLE.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for descriptions of MLE member functions.

To position the cursor on the first line of an MLE after a file has been imported, use the setCursorAtLine member function. The statement is highlighted in the following code example:

```
⋮
  filename=fd->fileName();                 //
   if (filename.size())                    //If the name has been specified,
   {                                              //load the file into mtextfield.
 mtextfield->importFromFile(filename.asString());
 mtextfield->setCursorAtLine( );         //Put the cursor on the first line.
   } / endif /                             //
} / endif /                                //
⋮
```

Figure 15 on page 87 shows the cursor on the first line of the MLE.



```
DEVICE=C:\DOS\SETVER.EXE
DEVICE=C:\DOS\HIMEM.SYS
DOS=HIGH
files= 30
BREAK=ON
BUFFERS=25,8
FCBS=20,8
LASTDRIVE=Z
SHELL=C:\DOS\COMMAND.COM /P /E:2048
DEVICE=C:\DOS\ANSI.SYS      /x
```

Figure 15. Example of Positioning the Cursor

Performing Clipboard Operations

The IMultiLineEdit class has several member functions to perform clipboard operations, including copy, cut, paste, clear, and discard. After you define an MLE and import it into a client window, use these member functions to copy text to the clipboard, cut and put text into the clipboard, or paste only the marked lines from the clipboard.

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for descriptions of other member functions.

The following statements show you how to implement the MI_CUT member function:

```
⋮
case MI_CUT:
   mtextfield->cut();                  //   cut to clipboard
  break;
⋮
```

Developing Containers

A *container* control holds objects. OS/2 provides a variety of containers, such as folders, templates, and the Workplace Shell itself. Containers can display their objects from different views: tree, icon, text, name, and details views. Using the User Interface Class Library, you can also develop your own containers and change their views, behaviors, and layouts.

Figure 16 shows an example of a container.

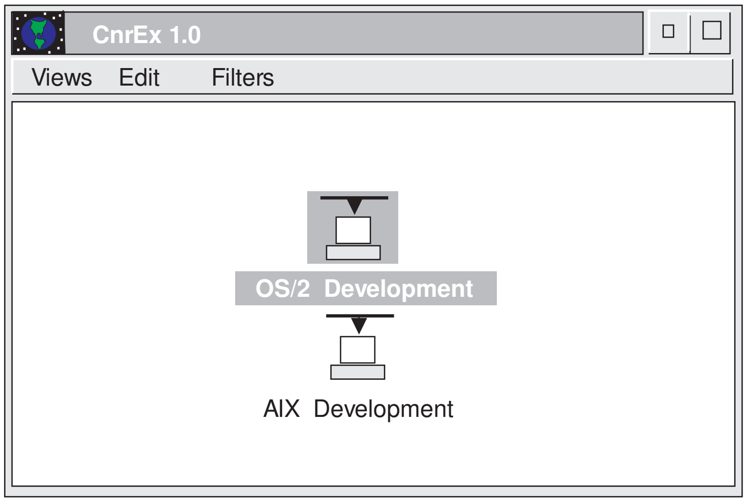


Figure 16. Example of a Container

Containers are defined by the Common User Access (CUA) architecture.

Creating a Container

Use the IContainerControl class to create an instance of a container object. With this class, you can control, for example, the view of the objects inside the container. To create a container, use the following statement:

```
IContainerControl cnrCtl(CNR_RESID, this, this);
```

Several styles are available for containers that you can use to manage such activities as multiple-selection and automatic positioning.

You can define the styles in the constructor, or you can use member functions to set the style required after you create an instance of the container object. An example of a style statement is highlighted in the following:

```
cnrCtl = new IContainerControl (CNR_RESID, this, this);
cnrCtl->setMultipleSelection();
```

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* to learn about other styles and related member functions.

Creating a Container Object

Because a container has no meaning without its contents or objects, use the IContainerObject class to create objects to put into it. At a minimum, an IContainerObject has an icon and a name.

The following statement is an example of an IContainerObject constructor:

```
IContainerObject   ( const IString&                string,
                           const IPointerHandle&     iconHandle =  );
```

Design your own objects for your applications, create a class that is derived from the IContainerObject class. For example, if you want to create a container object with department names, addresses, and ZIP codes for your company, define this class as follows:

```
class Department : public IContainerObject
{
   public:
          Department(const IString& Name,
                           const IPointerHandle& Icon,
                           const IString& Code,
                           const IString& Address,
                          ACnrexWindow   win);

         IString  Code()
          const {    return strCode; }

         IString  Address()
          const {    return strAddress; }

          void setCode (IString code)
          {strCode = code;}

          void setAddress (IString address)
          {strAddress = address;}

   private:

        IString  strAddress;
        IString  strCode;
};
```

The statements for a constructor definition are:

```
Department :: Department(const IString& Name, const IPointerHandle& Icon,
                const IString& Code, const IString& Address, ACnrexWindow  win):
         IContainerObject(Name,  Icon),
         strCode  (Code),
         strAddress  (Address),
         Mywin(win)
         {}
```

After you define the class, create an instance of an object using the following statements:

```
dept1 = new Department (
 "OS/2 Development",
 IApplication::current().userResourceLibrary().loadIcon(OSLOGO),
 "TWPD",
 "Building 71",
 this);
```

Adding and Removing Container Objects

After you create the objects and the container, add the objects into the container.

The following statements add objects to the container, cnrCtl. The first line simply adds an object, dept1. The next three lines add dept2, dept3, and dept4 in a hierachy under dept1. The last line adds dept5.

```
cnrCtl->addObject(dept1)              // OS/2 Development
cnrCtl->addObject(dept2,dept1); // C++ Development
cnrCtl->addObject(dept3,dept1); // Platform I
cnrCtl->addObject(dept4,dept1); // OS/2 Edit and Comm Services
cnrCtl->addObject(dept5);             // AIX Development
```

When you place the container in the client window and show the window and the container, you see a window like the one in Figure 17:

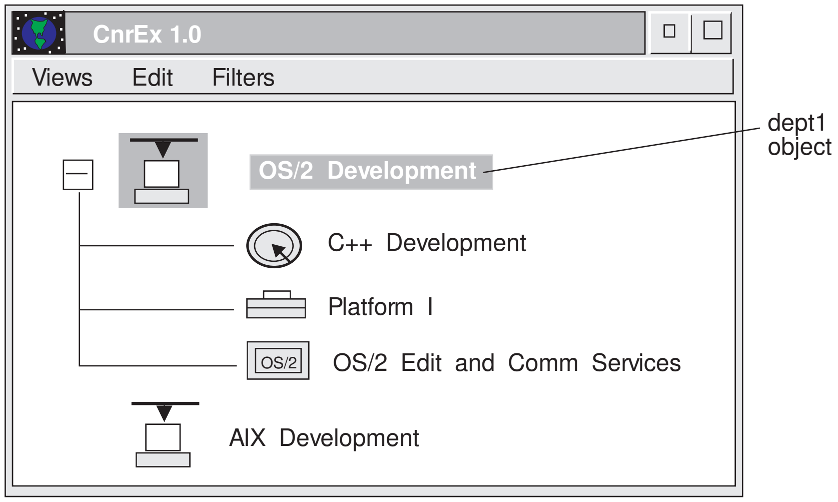


Figure 17. Example of a Container Showing Objects in a Tree View

The window in Figure 17 shows a tree view of the container's objects. This view is discussed later.

By default, the container only removes objects when the container is deleted. It does not delete them. You can delete all objects in the container when the container is deleted by using the following code statement.

```
cnrCtl->setDeleteObjectsOnClose();
```

Filtering Out Container Objects

With the User Interface Class Library, you can filter out objects in a container. The container uses the FilterFn nested class to show a subset of the existing objects by filtering out some of the objects.

To create a filter:

1. Define a class derived from the FilterFn class and override the member function isMemberOf to code the conditions of a valid object.

   The following example defines a FilterFn class:

   ```
   class OnlySelectedObjects : public IContainerControl:FilterFn
   {
   virtual Boolean
      isMemberOf( IContainerObject  object,
                        IContainerControl  container) const
     {
        return  isSelected(object);
     }
   };
   ```

   The member function isMemberOf receives the container and container objects and returns true or false. If true is returned, the container object remains displayed in the container; if false, the object is hidden.

   The isSelected member function returns true if the object has selection emphasis.

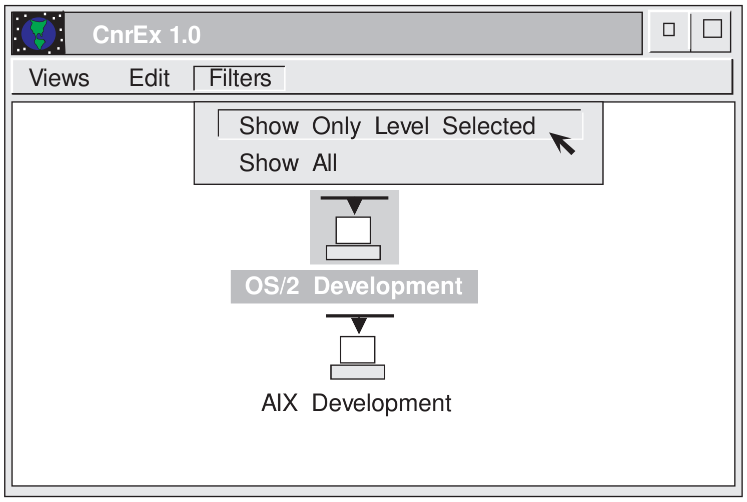
   Refer to *IBM C/C++ Tools: User Interface Class Library Reference* for information about the types of emphasis.

2. Apply the IContainerControl::filter member function. Use the following statements:

   ```
   OnlySelectedObjects onlySelectedObjects;
   cnrCtl->filter(onlySelectedObjects);
   ```

Figure 18 on page 92 shows how the container appears before and after you apply the filter.

Before



After

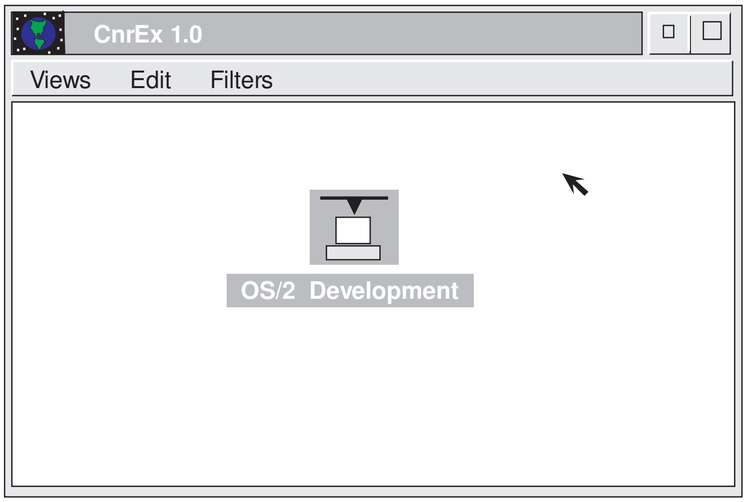


Figure 18. Example of Filtering Container Objects

Adding an Object Cursor

Use an object cursor to apply an action to a group of objects or to show which objects have a specific emphasis. Use the ObjectCursor nested class to iterate through a group of container objects.

The following example creates an ObjectCursor and uses it to set the emphasis selected to all container objects:

```
IContainerControl::ObjectCursor CO1 (cnrCtl);

for (CO1.setToFirst(); CO1.isValid(); CO1.setToNext())
  {
  cnrCtl->setSelected(cnrCtl.objectAt(CO1));
  }
```

Figure 19 shows the before and after result of setting the selection emphasis:

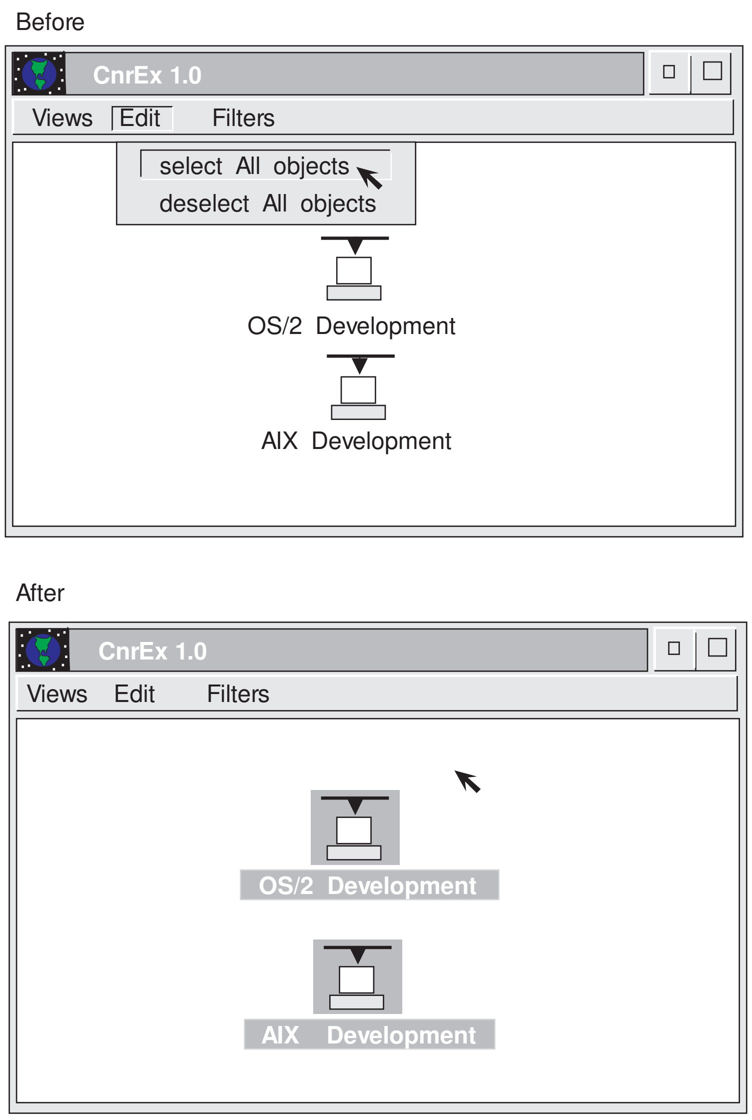


Figure 19. Example of Using an Object Cursor

Adding Views to a Container

You can use most of the User Interface Class Library views by using the corresponding member function. For example, the following statement uses the member function that causes a container to display the icon view:

```
cnrCtl->showIconView();
```

This statement provides the container view shown in Figure 20:

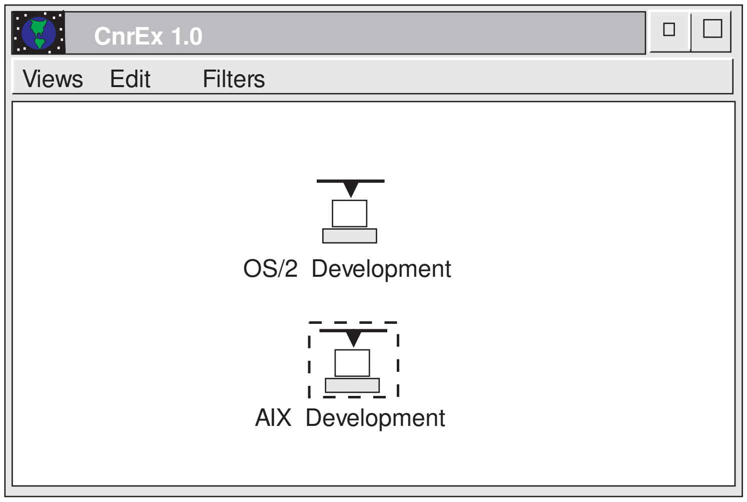


Figure 20. Example of the Icon View

The following statement provides the tree icon view:

```
cnrCtl->showTreeIconView();
```

Figure 21 shows a container with the expanded tree icon view:

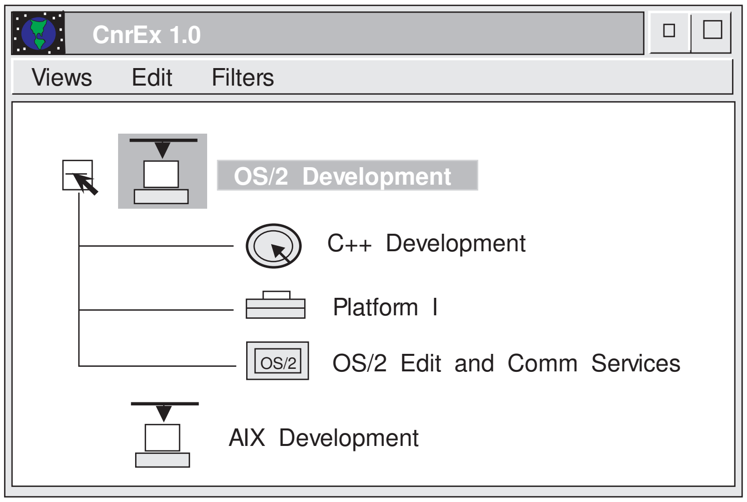


Figure 21. Example of an Expanded Tree Icon View

Showing the Details View Using Container Columns

Use the IContainerColumn class to show a details view from a container object in a container. You can use this class to set text in the heading of the columns, add horizontal and vertical separators by column, and align the column contents.

One way to create an instance of a IContainerColumn is for you to provide the offset of the object data to be displayed in the column and, optionally, the styles to be used for the heading and data.

The following is an example of the constructor for IContainerColumn:

```
IContainerColumn        ( unsigned long           dataOffset,
                              const HeadingStyle& title = defaultHeadingStyle(),
                              const DataStyle&       data = defaultDataStyle());
```

To create an instance of a container column, use the following statements:

```
colIcon = new IContainerColumn (IContainerObject :: iconOffset(),
                        IContainerColumn::defaultHeadingStyle  (),
                        IContainerColumn::icon  |
                        IContainerColumn::alignVerticallyCentered);

colName = new IContainerColumn (IContainerObject::iconTextOffset(),
                        IContainerColumn::defaultHeadingStyle  (),
                        IContainerColumn::string  |
                        IContainerColumn::alignVerticallyCentered  |
                        IContainerColumn::alignLeft  |
                        IContainerColumn::horizontalSeparator);

colCode = new IContainerColumn (offsetof(Department, strCode));

colAddress = new IContainerColumn (offsetof(Department, strAddress));
```

Use the IContainerObject member functions iconOffset and iconTextOffset with the C++ function offsetof to obtain the necessary offsets.

In the previous example, colIcon, colName, colCode, and colAddress are defined as members of an IFrameWindow. The statements look like this:

```
private:                                      //Define private information
   IContainerControl   cnrCtl;
   Department  dept1,  dept2,  dept3,  dept4,  dept5,  dept6,  dept7;
   IContainerColumn  colIcon,  colName,  colCode,  colAddress;
  IMenuBar             menuBar;
```

After creating the container columns, you can add heading text to them using the following statements:

```
colIcon->setHeadingText("Icon");
colName->setHeadingText("Department Name");
colCode->setHeadingText("Code");
colAddress->setHeadingText("Address");
```

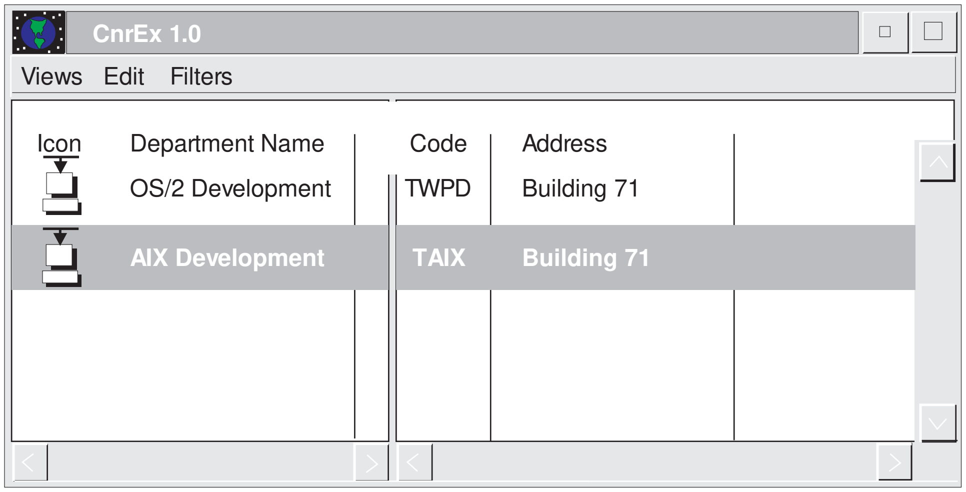
Use the member function showSeparators to add a vertical separator next to a column or a horizontal separator under the heading text. The default adds both. To create only one of the separators, specify it in the member function statement. The following statements show examples of how to create separators:

```
   //Only Horizontal Separator
colIcon->showSeparators(IContainerColumn::horizontalSeparator);
   //Only Vertical Separator
colName->showSeparators(IContainerColumn::verticalSeparator);
colCode->showSeparators(); //both separator by default
colAddress->showSeparators(); //both separator by default
```

After you create the container columns, add them into the container using the following statements:

```
cnrCtl->addColumn(colIcon);
cnrCtl->addColumn(colName);
cnrCtl->addColumn(colCode);
cnrCtl->addColumn(colAddress);
```

Figure 22 is an example of a details view of a container.



Icon	Department Name	Code	Address
	OS/2 Development	TWPD	Building 71
	AIX Development	TAIX	Building 71

Figure 22. Example of the Details View

Use the following code statement to put a split bar in the details view by specifying the last column to be viewed in the left window and the location of the split bar in pixels.

```
cnrCtl->setDetailsViewSplit(colName, 15 );
```

By default, the container only removes objects, but does not delete them, when the container is deleted. You can delete all objects in the container when the container is deleted by using the following code statement.

```
cnrCtl->setDeleteObjectsOnClose();
```

Displaying Pop-Up Menus

A *pop-up menu* is a menu that is displayed next to its associated object when a user presses the appropriate key or mouse button. A pop-up menu contains choices that can be applied to an object at the time the menu is displayed.

The User Interface Class Library provides the IPopUpMenu class, which is inherited from the IMenu class, to manipulate pop-up menus. Use the makePopUpMenu member function to construct a pop-up menu.

You can construct IPopUpMenu objects in the following ways:

- Create an empty menu bar given a window ID, frame window owner and styles.

```
IPopUpMenu ( IWindow  owner
             unsigned long menuWindowId,
             const Style &style = noStyle,
             Boolean autoDelete = true ),
```

- Create a menu bar with a menu resource ID and frame window owner.

```
IPopUpMenu ( const IResourceId &menuResId,
            IWindow   owner,
             Boolean autoDelete = true);
```

If autoDelete is true, the pop-up menu object is deleted by IMenuHandler when it is no longer visible. The pop-up menu is not visible until its IWindow::show member function is called. Typically, applications will override the makePopUpMenu member function in the IMenuHandler class and create a pop-up menu.

Version 6 of the Hello World application creates a pop-up menu object to apply to the "Hello, World" static text control area. The contents of the pop-up menu are defined in the AHELLOWE.RC resource file, as follows:

```
⋮
MENU WND_POPUP                                  //Popup Menu              v6
  BEGIN                                       //                          .
     MENUITEM "Left",     MI_LEFT               //Left Menu Item          .
     MENUITEM "Center", MI_CENTER               //Center Menu Item        .
     MENUITEM "Right",    MI_RIGHT              //Right Menu Item         .
  END                                         //                        v6
⋮
```

In the AHELLOW6.HPP file, an AMenuHandler class is defined to create the pop-up menu.

```
⋮
//                                        v6
// Class:     AMenuHandler                                          .
//                                                                  .
// Purpose: Subclass of IMenuHandler so that the a PopUp Menu can be          .
//              created.                                             .
//                                                                  .
//                                        .
class AMenuHandler: public IMenuHandler//                           .
{                                         //                          .
   protected:                             //Define Protected Member         .
     Boolean makePopUpMenu(IMenuEvent& menuEvent);                 //v6
};                                                                //v6
⋮
```

The makePopUpMenu member function creates an IPopUpMenu object with the default AutoDelete attribute. In the following example, from the AHELLOW6.CPP file, the WND_POPUP menu resource ID is used to create the pop-up menu.

```
⋮
//                                               v6
// AMenuHandler :: makePopUpMenu                                                    .
//                                               .
Boolean AMenuHandler :: makePopUpMenu(IMenuEvent& menuEvent)    //                  .
{                                                //                                 .
   IPopUpMenu   popUp;                          //Define popUp variable             .
   popUp=new IPopUpMenu(WND_POPUP,          //Create PopUp Menu with AutoDelete on .
                     menuEvent.window());        //                                 .
   popUp->show(menuEvent.mousePosition());//Show PopUp Menu                         .
  popUp->setAutoDeleteObject();               //                                    .
  return  true;                                  //Return                           .
} /  end AMenuHandler :: makePopUpMenu(...)  /                               //v6
⋮
```

The AMenuHandler is created in the setupClient member function in the AHELLOW6.CPP file. The menu handler is set for the hello static text control.

```
//                                               v5
// AHelloWindow :: setupClient()                                                    .
//    Setup  Client                                                                 .
//                                               .
Boolean AHelloWindow :: setupClient()    //Setup Client Window                      .
{                                                //                                 .
⋮
   AMenuHandler   mh=new AMenuHandler(); //Create Menu Handler                   v6
   mh->handleEventsFor(hello);                //Set Menu Handler for hello          .
   ICommandHandler::handleEventsFor(hello);//Set self as command event handler v6
⋮
}
```

The selected menu item in the pop-up menu is processed by the AHelloWindow::command member function, which is used to handle command events for the frame window.

Creating a Pop-Up Menu in a Container

To create a pop-up menu in a container, create a subclass of ICnrMenuHandler and override the makePopUpMenu member function to handle the pop-up menu events. Use the setCnr member function to set the container control and make it visible for our class. The following statements create the class:

```
class ACnrMenuHandler: public ICnrMenuHandler //
{
  public:

     setCnr(IContainerControl   pcnr)    { pcnrCtl = pcnr; }

   protected:                                          //Define Protected Member
     Boolean makePopUpMenu(IMenuEvent& cnEvt);

  private:
     IContainerControl   pcnrCtl;
};
```

After overriding the makePopUpMenu member function, you can add you own statements. The following statements create a pop-up menu displayed next to a container object with source emphasis:

```
Boolean ACnrMenuHandler :: makePopUpMenu(IMenuEvent& cnEvt) //
{                                                      //
   IPopUpMenu   popUp;                                 //Define popUp variable
   if (popupMenuObject()) {
     popUp=new IPopUpMenu(ID_POPMENU,        //Create PopUp Menu
                  cnEvt.window());                  //
     if (!pcnrCtl->isDetailsView())           //Details View is only way to edit
     {                                                  //    Name, Code and Address so
        popUp->disableItem(MI_EDNAME);        //    Disable these items if not
        popUp->disableItem(MI_EDCODE);        //    details  view.
        popUp->disableItem(MI_EDADDRESS);  //
     }
     else
     {
        popUp->disableItem(MI_EDRECORD);    //
     }
     popUp->setAutoDeleteObject();            //
     popUp->show(cnEvt.mousePosition()); //Show PopUp Menu
     pcnrCtl->showSourceEmphasis(popupMenuObject());
     pcnrCtl->setCursor(popupMenuObject());
     return true;                                      //Return PopUp Menu
   }
  return  false;
};
```

Figure 23 on page 101 shows the pop-up menu in a container object.

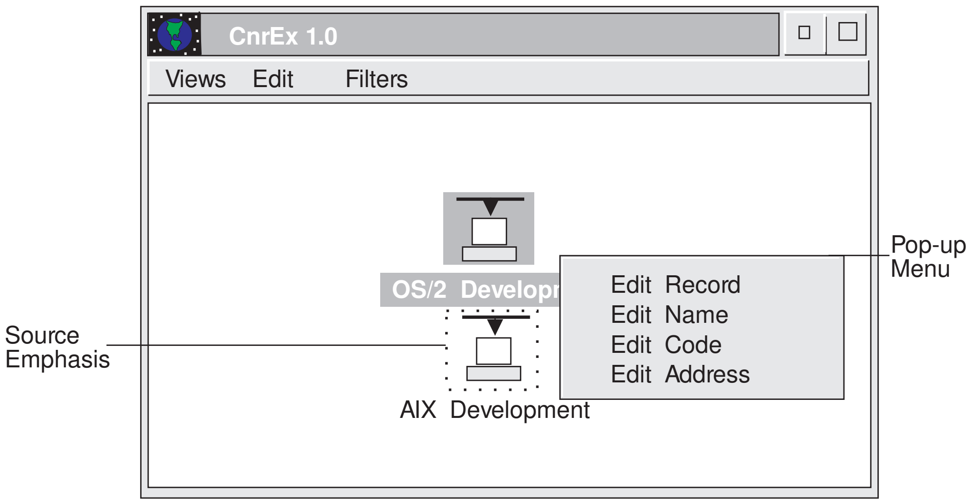


Figure 23. Example of a Pop-up Menu in a Container Object

Organizing Information Using a Notebook

A *notebook* control is visual component that organizes information on individual "pages" so that a user can find and display that information. A user can select and display pages by using a mouse or the keyboard. Use the INotebook class to create and manage the notebook control window. You can create an instance of this class in the following ways:

```
INotebook(unsigned long windowId, IWindow  parent, IWindow  owner,
          const IRectangle& initial = IRectangle(),
          const Style style = defaultStyle());

INotebook(unsigned long windowId, IWindow  parentAndOwner);

INotebook(const IWindowHandle& handle);
```

The default style of this class has solid binding, square corner tabs, and left-justified status-line text. To change the style, define an instance of the INotebook::Style class and initialize it. For example:

```
INotebook::Style style = INotebook::spiralBinding |
                         INotebook::roundedTabs  ;
```

The notebook created using the preceding statements has a spiral binding and rounded corner tabs. Figure 24 on page 102 shows an example of a notebook control.

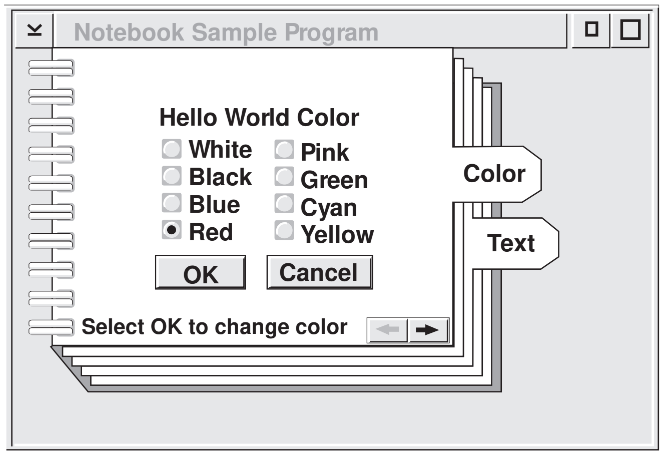


Figure 24. Notebook Control Example

Specifying Notebook Styles

The User Interface Class Library provides styles and functions to change notebook controls. You can create a notebook, specify its style, and change the size of major tabs and minor tabs with the following statements:

```
INotebook       pnoteBook;

pnoteBook= new INotebook ( ID_NOTEBOOK, this , this,
                           IRectangle(),
                           INotebook::spiralBinding  |
                           INotebook::backPagesTopRight  |
                           INotebook::majorTabsRight  |
                           INotebook::statusTextLeft  |
                           IWindow::visible);

pnoteBook->setMajorTabSize(ISize(6 ,3 ));
pnoteBook->setMinorTabSize(ISize(8 ,4 ));
```

Version 6 of the Hello World example creates a notebook using these statements in the ACOLORW6.CPP file:

```
⋮
notebook=new INotebook(WND_COLOR_NOTE,//Create Notebook Color
   this, this, IRectangle(),            //
  INotebook::defaultStyle());           //
notebook->setAutoDeleteObject();        //Delete C++ Obj, if PM Obj is deleted
⋮
```

Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for more information on notebook styles.

Setting and Changing Notebook Pages

The page setting objects allow the user to change and set information about pages in a notebook. Use the INotebook class in conjunction with the PageSettings class.

The following example of PageSettings is taken from the file ACOLORW6.CPP used in Version 6 of the Hello World application:

```
⋮
Boolean AColorWindow :: createStaticPage()
{
  staticCanvas=new  IMultiCellCanvas(
      WND_STATIC_COLOR, notebook, this,
     IRectangle());
  staticCanvas->setAutoDeleteObject();  //Delete  C++  Obj,  if  PM  Obj  is  deleted
vellip.
   INotebook::PageSettings::Attribute       //Define the Page Attributes
      attribute=INotebook::PageSettings::majorTab //    with a Major Tab and
     | INotebook::PageSettings::autoPageSize;      //    AutoPageSize
   staticPage=INotebook::PageSettings(attribute);//Create Static Color Page
   staticPage.setTabText("1");                  //Set Tab Text
   notebook->addFirstPage(staticPage,       //Add as First Page to notebook
  staticCanvas);                                //
  return  true;                                 //Return
} /  end AColorWindow :: createStaticPage()  /
⋮
```

This page is created on a multicell canvas. The tab text is set to 1 and this page is inserted into the notebook as the first page.

Chapter 9. Covering Advanced Topics

This chapter covers the following topics:

Event handling
Tracing
Exception handling
Threads and protecting data

Extending Event Handling

The User Interface Class Library provides handlers for common Presentation Manager (PM) messages. However, you may find it necessary to process messages for which there are no predefined handler classes. The User Interface Class Library makes it easy to add new event and handler classes.

The IHandler class is designed to act as a base class for handlers. All event handlers are derived from this class.

The following steps show you how to provide a handler for PM timer events.

1. Subclass the IHandler class by creating a class declaration for ATimerHandler. The class is derived from IHandler and provides a virtual function timer to process the event. For example:

```
class ATimerHandler : public IHandler
{
public:
  / use default constructor /

   Boolean
      dispatchHandlerEvent( IEvent& evt );

protected:
virtual Boolean
      timer( IEvent& evt );
};
```

2. Override the dispatchHandlerEvent member function. The function determines the relevance of the message. If the message is not relevant, the function returns false and passes the message to other handlers attached to the window. For example:

```
Boolean ATimerHandler::dispatchHandlerEvent( IEvent& evt )
{
   if ( evt.eventId() == WM_TIMER )        //If timer event, call
        return timer( evt );             // function to process
                                                //Note: WM_TIMER is defined in the
   return false;                          // OS/2 Developer's Toolkit
}
```

3. Create the ATimerHandler::timer member function. This provides a default return and acts as a base class. The default provides safe behavior when you create an instance of the class as shown in the following example:

```
Boolean ATimerHandler::timer( IEvent& evt )
{
  return  false;
}
```

   The ATimerHandler class encapsulates the WM_TIMER messages generated by PM. You can derive a class from ATimerHandler and override the ATimerHandler::timer member function to provide whatever event handling you require.

   To prevent users of this class from having to understand how information is encoded in the two message parameters inside the event, derive an event class from IEvent to encapsulate this information. The following statements show an example of how to do this:

```
class ATimerEvent : public IEvent
{
public:
   ATimerEvent( IEvent &evt ) : IEvent( evt ) {;}    // Define functions inline

  unsigned  long
      timerId() const { return parameter1().number1(); }
};
```

   You can only construct objects of this class from an instance of IEvent. Because of the small amount of code required, the example defines the code inline.

   Change the dispatchHandlerEvent member function to create an instance of ATimerEvent. Also, change the ATimerHandler::timer member function to accept an ATimerEvent object as a parameter.

```
Boolean ATimerHandler::dispatchHandlerEvent( IEvent& evt )
{
   if ( evt.eventId() == WM_TIMER )         //If timer event, call
       {                                          // function to process
       ATimerEvent timerEvt( evt );
       Boolean rc = timer( timerEvt );      //Call timer to process
       evt.setResult(timerEvt.result()); //Move results to event
       return rc;                                 //Return with return code
      }

  return  false;
}
```

   The two classes now completely encapsulate timer messages. Users of the classes do not need to know which PM messages are generated or how the information is encoded in the message parameters.

You can restrict the window classes to which the handler can be attached. A handler class can override the handleEventsFor and stopHandlingEventsFor member functions to provide a certain degree of type safety.

The following steps show you how to restrict the attachment of the ATimerHandler class to the ITextControl class and its derived classes:

1. Write the class declaration following this example:

```
class ATimerHandler : public IHandler
{
public:
  /  use default constructor  /

Boolean
    dispatchHandlerEvent( IEvent& evt );

virtual ATimerHandler
    &handleEventsFor          ( ITextControl  textWindow ),
    &stopHandlingEventsFor ( ITextControl  textWindow );

protected:
virtual Boolean
    timer( ATimerEvent& evt );

private:                                            //Make these functions private
virtual IHandler                                  // so they cannot be called
    &handleEventsFor          ( IWindow  window ),
    &stopHandlingEventsFor ( IWindow  window );
};
```

2. Override handleEventsFor member function to accept only ITextControl objects, as shown in the following example:

```
ATimerHandler &ATimerHandler::handleEventsFor( ITextControl  textWindow )
{
  IHandler::handleEventsFor( textWindow );    //Call parent class
  return  this;                                       // member function
}
```

3. Override stopHandlingEventsFor member function to accept only ITextControl objects. For example:

```
ATimerHandler &ATimerHandler::stopHandlingEventsFor( ITextControl  textWindow )
{
  IHandler::stopHandlingEventsFor( textWindow ); //Call parent class
  return  this;                                            // member function
}
```

Simplifying Tracing

Use the ITrace class with its related macros to simplify the process of adding tracing code to an application. Using the trace functions, you can write trace output to standard output stream (STDOUT), standard error stream (STDERR), or an OS/2 queue. You can control the trace options using environment variables or by statements in your program.

The environment variables, ICLUI TRACE and ICLUI TRACETO, provide the default tracing options.

ICLUI TRACE has three valid values:

Values	Trace Setting
OFF	Set trace off. This is the default.
ON	Set trace on.
NOPREFIX	Set trace on, but no prefix information is written to trace.

ICLUI TRACETO has three valid values:

Values	Trace Is Written To:
QUEUE	32-bit named OS/2 queue. The name is \\QUEUES\PRINTF32. This is the default.
STDOUT	Standard output stream.
STDERR	Standard error stream.

The following code example shows you how to write trace information:

```
#include <itrace.hpp>                              //Include trace class
/    ... function to trace    ...  /
void myFunction(int x)
{
   ITrace trc("myFunction");                       //Create an ITrace object
   trc.write("now at this point");              //Use static member function
   ITrace::write(IString("the value is = ") + IString(x));    //write
  return;
}
```

If you provide message text, the ITrace instance writes a message during its constructor and destructor to indicate the start and end of the function. Because of the performance overhead of tracing, you may want to limit your use of the trace code to your development and test phases. For example, to run the preceding example program after testing it, you would remove the tracing lines and recompile the program.

A more flexible approach to tracing is to use the predefined User Interface Class Library macros. These macros expand to calls to the trace function only if another macro is defined. Using this approach, the example becomes the following:

```
#define    IC_TRACE_DEVELOP                                  //Define trace level
#include <itrace.hpp>                                       //Include trace class
/    ... function to trace    ...  /
void myFunction(int x)
{
   IFUNCTRACE_DEVELOP();                                     //Trace entry and exit
   ITRACE_DEVELOP("now at this point");
   ITRACE_DEVELOP(IString("the value is = ") + IString(x));
  return;
}
```

For PM programs, information written to STDOUT and STDERR is discarded. If you start the program from the command line, redirect these streams to a file or named pipe. The commands to redirect the stream to a file are as follows:

```
[C:\]hello1 >stdout.lst           <- redirect stdout to file stdout.lst
[C:\]hello1 2>stderr.lst          <- redirect stderr to file stderr.lst
```

An example of the trace output is shown below:

```
<--- prefix -----> <----- trace ----------------->

    9   595: 1 +myFunction(int x)(121)                    <- function entry
    1    595: 1      >now at this point
    11   595: 1      >the value is = 5
    12   595: 1 -myFunction(int x)                         <- function exit
```

The prefix area shows the trace line number, process ID, and thread ID.

The IFUNCTRACE_DEVELOP macro automatically generates the trace lines that show the entry and exit from the function. The number in brackets after the argument list is the source code line number of the macro. The I_TRACE_DEVELOP macro produces the other two lines.

If the IC_TRACE_DEVELOP macro is defined, the trace statements are generated; otherwise, no trace statements are generated. This means that after testing is complete, it is not necessary to remove all the trace lines. Instead, remove the macro and recompile the code.

Redirecting Trace Output

You can redirect trace output to a file at run time to help diagnose exceptions thrown by the User Interface Class Library. Uncaught exceptions can cause the application to end. Use the following commands to redirect trace output:

SET ICLUI TRACETO=STDERR	Causes tracing to be written to the standard error device
MYAPP 2>TRACE.LOG	Runs MYAPP.EXE and redirects standard error to an output file

Handling Exceptions

The User Interface Class Library uses the C++ exception handling to return errors to the application. Several different classes of exception objects can be thrown. Because all these classes are derived from the IException class, an application can catch specific exceptions or all exceptions.

The following table lists the exception classes and the situations in which they are typically thrown.

Exception Class	Thrown When
IAccessError	A logical error occurs, such as “resource not found”
IAssertionFailure	The expression in an IASSERT macro evaluates to false
IDeviceError	A hardware related error occurs
IInvalidParameter	An invalid parameter is passed
IInvalidRequest	An object is in the wrong state for a function
IResourceExhausted	A resource is exhausted or currently unavailable
IOutOfMemory	The heap storage is exhausted
IOutOfSystemResource	An OS/2 resource is exhausted
IOutOfWindowResource	A PM resource is exhausted

Typically, an application surrounds a function that might fail with a try-catch block. The following example shows how an application attempts to set its default resource library. If this fails, an IAccessError is thrown and the example code explicitly handles the exception. The application passes on any other exception that is thrown.

```
try
   {                                              //Try to use notfound.dll
  IApplication::current().setUserResourceLibrary("NOTFOUND");
   }
catch (IAccessError &exc)                    //Catch only access errors
   {                                              //DLL probably not in libpath
   const char  exText = exc.text();
   unsigned long exId = exc.errorId();
   /  ... add code to process the exception ...  /
   }
```

Each exception object that is thrown contains the following:

- An error number
- A severity indicator
- One or more lines of text
- Information about where the exception was thrown

The IException class provides accessor functions to extract this information from the object. The textCount member function retrieves the number of lines of exception text, and the ITextControl::text member function retrieves the exception text.

The ITHROW, IASSERTSTATE, IASSERTPARAM, and ITHROWGUIERROR macros throw all exceptions in the library, and the RETHROW macro rethrows the exceptions. These macros automatically insert into the exception object the line and program file in which the exception was thrown. These macros also log the exception information. By default, exception information is written to the same destination as the trace output. However, you can provide your own function by deriving a class from IException::TraceFn, overriding the write virtual function, and registering it using IException::setTraceFunction.

Note: C++ exceptions are not the same as OS/2 exceptions.

Providing a Default Exception Handler

The C++ exception mechanism passes exceptions back up the function call chain until it finds a try-catch block that can handle the exception. Uncaught exceptions are passed to PM, which can cause your application to end unpredictably.

A User Interface Class Library application can register a default exception handler. The event dispatching loop catches any exception thrown in a handler or function called from a handler and passes it to the registered default exception function. This allows the application to either try to continue or to end in a controlled way.

The following example shows you how to create a default handler that uses the tracing functions to log the exception and display the information in a message box. The steps to register a default exception handler are:

1. Subclass the IWindow::ExceptionFn class. The IWindow::ExceptionFn class has a single constructor that requires a frame window in which the handler displays a message box. The frame window acts as the owner of the message box. For example:

```
class AExceptionFn : public IWindow::ExceptionFn
{
public:
    AExceptionFn(IFrameWindow  frame) : owner(frame) {;}
Boolean
    handleException (IException& exception, IEvent& event);
private:
   IFrameWindow   owner;
};
```

2. Override the handleException member function. The last of the text messages of the exception object is written to the trace output and displayed in a message box. The function returns true to indicate that the exception should not be rethrown. For example:

```
Boolean AExceptionFn::handleException (IException& exception, IEvent& event)
{
   IFUNCTRACE_DEVELOP();                                        //Trace function entry/exit
   unsigned long cnt    = exception.textCount();
   const char  text = (cnt >  ) ? exception.text( cnt-1 )
                                          : "No error text available" ;
   IString str( text );
   ITRACE_DEVELOP( exception.name() );
   ITRACE_DEVELOP( IString("text count = ") + IString(cnt) );
   ITRACE_DEVELOP( str );
   IMessageBox msgbox( owner );                        //Create message box
   msgbox.setTitle( exception.name() );
   msgbox.show( (char  )str ,
                    IMessageBox::okButton              |
                    IMessageBox::informationIcon    |
                    IMessageBox::applicationModal  |
                    IMessageBox::moveable               );
   return true;                                         //Stop rethrow of exception
}
```

3. Create an object of this class. The object is part of the main application window object. For example:

```
class aListBoxWindow : public IFrameWindow
{
  public:
     aListBoxWindow(unsigned long windowId);         //Constructor for this class
     /  ... other public member functions ...  /
  private:
     AExceptionFn       excptHandler;
     /  ... other private data ...  /
};
```

4. Register using IWindow::setExceptionFunction. Create the exception function in the constructor for the window and register it. For example:

```
aListBoxWindow::aListBoxWindow(unsigned long windowId)
   : IFrameWindow( IFrameWindow::defaultStyle() |
                         IFrameWindow::minimizedIcon,
                         windowId)  ,
     excptHandler( this )
{
  setExceptionFunction(&excptHandler);
   /  ... rest of constructor code ...  /
}
```

Controlling Threads and Protecting Data

The User Interface Class Library provides classes to implement multi-threaded programs. The primary class you use to deal with threads is IThread. Instances of this class represent separate threads of execution and provide the ability to start and stop the thread, set various thread attributes, and determine the default environment for the thread. In addition, with the ICurrentThread class you can set and query attributes for the currently executing thread, start event processing, and suspend the current thread until another thread has terminated.

Accessing the Current Thread

There is only a single instance of the class for each thread, and it can be accessed using the following statement:

```
ICurrentThread      curThread = IThread::current();
```

This static data member accesses information held on a per-thread basis. The member also accesses some functions that can be applied only to the current thread. One example is the initialization of the PM environment for a thread. A thread without a PM environment can initialize one and later terminate it using the following statements:

```
IThread::current().initializePM();
/  ... do thread processing ...  /
IThread::current().terminatePM();
```

If necessary, the thread can enter its event processing loop using:

```
IThread::current().processMsgs();
```

Starting a Thread

Use the IThread class to start an execution thread. Once started, the instance of IThread provides a means of querying and stopping the thread. The thread and the instance of IThread are independent; therefore, when the instance of IThread is destroyed, the thread is unaffected.

The function to be dispatched on a separate thread can be either a member function or a nonmember function. If you create an instance of IThread with the function, a thread is created and dispatched immediately. Alternatively, you can create an instance of the class and later dispatch it. This allows you to set arguments that affect the execution of the thread prior to dispatching.

Starting Nonmember Functions

The IThread class dispatches nonmember functions with either of the following two function prototypes:

```
void (_Optlink  )(void  )

void (_System  )(unsigned long)
```

They provide support for migrating code that uses either _beginthread or DosCreateThread to start the function. The linkage directives, _Optlink (the default) and _System, are discussed in the *C++ Compiler Reference*.

To start a thread with the default environment and default options, the following statements are needed:

```
void threadFn(void  pvParms);          //Function to run on separate thread
void         pv;

IThread      thread(threadFn, pv);      //Dispatch thread with default environment
```

The following example shows you how to set some of the options before dispatching the thread. The environment is created before the function is called, and appropriate cleanup actions are taken after it terminates:

```
IThread      thread;                           //Assume PM environment
thread.setStackSize(65536);              //Set 64K stack size
thread.setQueueSize(32);                   //32 elements in PM queue
thread.start(threadFn, pv);              //Dispatch thread
```

Other functions also exist to change the priority level of the thread, although for threads that process events, changing the priority can adversely affect the overall system performance.

Once you have started a thread, you can suspend, resume, or stop it. You can also query its thread ID. The following example stops the thread if it has a thread ID of 2:

```
void         pv;
IThread      thread(threadFn, pv);      //Dispatch thread with default environment
/  ... let thread process ...  /
if (thread.id() == IThreadId(2))     //If thread ID is 2, then stop it
  thread.stop();
```

Because threads often require a PM environment to be established before they work, the User Interface Class Library automatically establishes a PM environment for all threads created in a PM application. If this is not necessary, you can use a thread to request that this initialization be skipped. For example:

```
void threadFn(void  pvParms);                  //Function to run on separate thread
void         pv;

IThread      thread(threadFn, pv, false);   //No PM environment
```

Starting a Member Function

Use the IThread class to start member functions. Direct support is provided for starting member functions that have no arguments, but you can also start functions that have arguments.

To start a member function that takes no arguments, use the following steps:

1. Create an instance of the template class IThreadMemberFn.
2. Start a thread and pass the instance as an argument.

The following example shows how to execute the function AClass::longFn on a separate thread. Create an instance of the template class with the class that contains the member function. Create the instance of the template class with the operator function new so that the instance is destroyed automatically when the thread ends. The two arguments on the constructor are the object for which the member function is called and the member function itself, as shown in the following example:

```
/  function to run is ...    void AClass::longFn()        /
AClass object;                              //Object to run member function against

IThreadMemberFn<AClass>  aMemberFn =
                          new IThreadMemberFn<AClass>( object
                                                       , AClass::longFn );
IThread    thread( aMemberFn );          //Dispatch thread
```

To start a member function that takes arguments, use the following steps:

1. Derive a class from the IThreadFn class.
2. Define a constructor that takes an object of the class and the arguments you want to pass.
3. Override the ICurrentApplication::run member function to call the member function.
4. Create an instance of the derived class.
5. Start a thread and pass the instance as an argument.

The following example shows how to start a function:

1. Write the class declaration. The class is derived from the IThreadFn class. It has a single constructor that requires an instance of the AClass class and the two parameters. The class overrides the virtual function run and calls the required member function, as shown in the following example:

```
class AClass
{
  public:
      void longFn(int, IString);
   /  ... rest of class declaration ...  /
};

                                    //This class runs the member function
                                    // AClass::longFn(...) on a separate thread
class AThreadLongFn : public IThreadFn
{
  public:
      AThreadLongFn(AClass &obj, int i, IString str)
          : object( obj )
          , value( i )
          , string( str ) {;}
      void run() { object.longFn( value, string ); }
  private:
     AClass     &object;
     int          value;
     IString     string;
};
```

2. Create an instance and dispatch it. As before, create the instance using the new operator so that it is destroyed automatically:

```
AClass   object;                               //Object to run member function against
int        number = 6;
IString greeting( "Hello" );

/  function to run is ...   void AClass::longFn(int, IString)       /
                                                    //Create  object
  AThreadLongFn   .aMemberFn  =  new  AThreadLongFn(  object,  number,  greeting  );
   IThread          thread( aMemberFn );      //Dispatch thread
```

Protecting Data

If your applications have multiple threads, you typically need to serialize their access to certain resources. The User Interface Class Library provides several classes to assist you. Use the IPrivateResource class to serialize access to a resource within a single process. The ISharedResource class extends this function by providing a lock that can also be used between processes.

The simplest way to serialize access to a function is to provide a static instance of the IPrivateResource class. You can use this instance in association with the IResourceLock class to control access. In the following example, the function guarantees that only one thread accesses it at one time:

```
static  IPrivateResource  resourceKey;      //Key must exist when function
                                                   // called
void serializedFunction()
{
IResourceLock resLock(resourceKey);         //Create lock
/ ... serialized code ... /
}                                                  //Lock freed with resLock destructed
```

When a thread calls serializedFunction, it is blocked until any other thread executing the function exits it. This can lead to deadlock problems, so a safer approach is to give a timeout value, which is the number of milliseconds that a thread can be blocked. If this timeout limit is exceeded, an IResourceExhausted exception is thrown, which can then be caught.

The definition of the function becomes the following:

```
static  IPrivateResource  resourceKey;

void serializedFunction()
{
IResourceLock resLock(resourceKey, 1  );      // timeout period =  .1 s
/ ... serialized code ... /
}
```

The code to call the function is:

```
try
   {
   serializedFunction();
   }
catch (IResourceExhausted exc)
   {
    / . ... handle failure to run function ...  ./
   }
```

Suspending Threads for Critical Sections of Code

A *critical section* of code is a portion of code that must be executed by one thread while all other threads in the process are suspended. As an example, one situation would be the need for one thread to modify global data while preventing other threads from accessing the data until the modifications were complete.

The User Interface Class Library provides a critical section object to handle such situations. A thread creates the critical section object before it enters a critical section and destroys the object when it exits the section. The simplest way to do this is to enclose the critical section in its own block and define the object at the start of the block, as in the following:

```
{
ICritSec   lock;
/  ... do critical section processing here ...  /
}                                             // lock destructed when block ends
```

Because a critical section freezes the other threads in the process, use it with care. In addition, be careful when calling certain OS/2 functions within a critical section because the results may be unpredictable.

Using Direct Manipulation

Direct manipulation is a user interface technique that allows a user to initiate application functions by manipulating objects. The user initiates an action by selecting an object, pressing and holding down a mouse button while *dragging* it over another object in the window. The user then *drops* the object over the target object by releasing the mouse button. For this reason, direct manipulation is also know as *drag and drop*.

The User Interface Class Library provides four main types of objects to support direct manipulation:

An event handler (IDMSourceHandler or IDMTargetHandler)
A renderer (IDMSourceRenderer or IDMTargetRenderer)
A drag item (IDMItem)
A drag item provider (IDMItemProvider)

IDMSourceHandler and IDMTargetHandler are derived from IDMHandler. They handle the PM direct manipulation window messages. Objects from these classes pick up the WM_* and DM_* messages for the source and target windows and translate them to virtual function calls to the handler.

In addition to translating messages to virtual function calls, these handlers also manage the second type of objects, the renderers. Renderers transfer the representation of the object being manipulated between the source and target windows. Direct manipulation renderers are objects of classes IDMSourceRenderer and IDMTargetRenderer, derived from IDMRenderer.

When an IDMSourceHandler object is created, the User Interface Class Library creates a default IDMSourceRenderer. The default source renderer support is shown in Figure 25.

Figure 25. Default Source Renderer

Mechanism	**Format**	**Item Type**
IDM::rmLibrary	IDM::rfProcess	IDM::any
IDM::rmLibrary	IDM::rfText	IDM::any
IDM::rmLibrary	IDM::rfSharedMem	IDM::any
IDM::rmPrint	IDM::rfUnknown	IDM::any
IDM::rmDiscard	IDM::rfUnknown	IDM::any
IDM::rmFile	IDM::rfUnknown	IDM::any

When an IDMTargetHandler object is created, the User Interface Class Library creates a default IDMTargetRenderer. The default target renderer support is shown in Figure 26.

Figure 26. Default Target Renderer

Mechanism	Format	Item Type
IDM::rmLibrary	IDM::rfProcess	IDM::any
IDM::rmLibrary	IDM::rfText	IDM::any
IDM::rmLibrary	IDM::rfSharedMem	IDM::any
IDM::rmFile	IDM::rfUnknown	IDM::any

IDM::rmLibrary is the rendering mechanism that is defined for use in the User Interface Class Library. Other rendering mechanisms defined as part of the default renderers are shown in Figure 27.

Figure 27. Default Rendering Mechanisms

Mechanism	Description
IDM::rmPrint	Used when an User Interface Class Library object is dropped on a printer
IDM::rmDiscard	Used when an User Interface Class Library object is dropped on a shredder
IDM::rmFile	Used when an OS/2 file is dragged from the source and dragged over or dropped on a target

Several default rendering formats have been defined to assist the developer in using the direct manipulation classes. They are shown in Figure 28.

Figure 28. Default Rendering Formats

Format	Description
IDM::rfProcess	Used to determine if the source of the direct manipulation operation and the target are in the same process. This format must be constructed by calling the static member function IDMItem::rfForThisProcess.
IDM::rfText	Used when dragging text that has a length of 255 or less and no embedded NULL characters.
IDM::rfSharedMem	Used when a shared memory buffer is required to transfer the data from the source to the target.
IDM::rfUnknown	Used when the format is unknown.

The IDM::any type can be used to represent any drag item type.

The *native* renderer is the first rendering mechanism and format defined at the creation of the item. For example, in the declaration of the default source renderer, the native renderer supports the library rendering mechanism, the process rendering format, and any item type. In the declaration of the default target renderer, the native renderer supports the library rendering mechanism, the process rendering format, and any item type.

To create renderers for controls not supported by User Interface Class Library, you can create your own source or target renderer by deriving from the IDMSourceRenderer or IDMTargetRenderer, creating instances, and then adding to the handler using setDefaultTargetRenderer and setDefaultSourceRenderer.

The third type of objects are the drag items, represented by objects of class IDMItem. These objects encapsulate the logic that serves as the bridge between the context-insensitive handlers and renderers and the application-specific behavior of particular source and target windows. Thus, the drag items provide the application-specific semantics of the direct manipulation operation.

The IDMItemProvider class is an extension of the IWindow class that provides direct manipulation functions. Objects of the IDMItemProvider class allow generic controls, such as an entry field, to generate context-sensitive drag items. For example, a container that contains customer objects can generate a “customer” item; a bit map can provide an item that can extract the picture from a .BMP file.

User Interface Class Library provides direct manipulation for:

- Entry fields
- Multiple-line edit (MLE) fields
- Intra-process containers

The following sections discuss how to add direct manipulation to your applications. Sample applicaton files are provided with the User Interface Class Library product diskettes. Complete listings of the examples used in this chapter are included in the \ibmcpp\samples\iclui directory. This directory also includes additional samples that illustrate advanced features not discussed in this chapter.

Enabling Direct Manipulation for an Entry Field or MLE

To enable direct manipulation for an entry field or an MLE control, call the IDMHandler static function enableDragDropFor().

This static function creates:

- Source and target handlers
- Source and target default renderers
- Entry field item provider

In the following sample code, the highlighted lines enable direct manipulation of text between two entry fields in the same process. Direct manipulation is enabled the same way for an MLE.

```
⋮
void main()
{
  IFrameWindow
     frame( "ICLUI Direct Manipulation Sample 1" );

  IEntryField                              //Create window with two
     client( 1   , &frame, &frame ),       //entry fields, client and ext.
     ext     ( 1  1, &frame, &frame );

  IDMHandler::enableDragDropFor( &client );//Enable direct manipulation
  IDMHandler::enableDragDropFor( &ext );     //for both entry fields.

  frame                                              // Frame setup
     .setClient( &client )
    .addExtension( &ext, IFrameWindow::belowClient,  .5 )
    .setFocus()
    .show();

  IApplication::current().run();          // Run sample

}
```

Enabling Direct Manipulation for a Container

To enable direct manipulation for a container, call the IDMHandler static functions enableDragFrom and enableDropOn.

In the following example, the dmsamp3.hpp file has defined a container control object. The .CPP file creates the container and container objects and, in the highlighted lines, calls enableDragFrom and enableDropOn.

```
 1 #include "dmsamp3.hpp"
 2
 3 void main()
 4 {
 5      MySourceWin sourceWin (WND_SOURCE);
 6      MyTargetWin targetWin (WND_TARGET);
 7      IApplication::current().run();
 8 }
⋮
52
53 MySourceWin :: MySourceWin ( unsigned long windowId ) :
54                          MyWindow ( windowId )
55 {
56     ITitle title (this, "C Set ++ Direct Manipulation - Source Container" );
57     IDMHandler::enableDragFrom( cnrCtl );
58 };
59
6  MyTargetWin :: MyTargetWin ( unsigned long windowId ) :
61                          MyWindow ( windowId )
62 {
63     ITitle title (this, "C Set ++ Direct Manipulation - Target Container" );
64     IDMHandler::enableDropOn( cnrCtl );
65 }
⋮
```

Lines 53 through 56 create a source window.

Line 57 enables the window as a source.

Lines 60 through 63 create a target window.

Line 64 enables the window as a target.

Enabling a Control as a Drop Target

To enable other controls as drop targets, you must specifically create the item providers that the User Interface Class Library generates automatically for entry field, MLE, and container controls. You should:

Derive a class from the base class IDMItem and override the targetDrop member function.

Write a drag item provider class for the customized item class using the IDMItemProviderFor template class.

Use the default target handler and renderer for the customized object.

The following example adds drop support to a bit-map control.

The header file dmsamp2.hpp defines two classes, ABitmapItem and ABitmapProvider, and overrides the targetDrop and provideEnterSupport member functions.

```
 1 #include <idmprov.hpp>
 2 #include <idmitem.hpp>
 3 #include <idmevent.hpp>
 4
 5 class ABitmapItem : public IDMItem {
 6 public:
 7      ABitmapItem ( const IDMItem::Handle &item );
 8
 9 virtual Boolean
1       targetDrop ( IDMTargetDropEvent & );
11 };
12
13 class ABitmapProvider : public IDMItemProviderFor< ABitmapItem > {
14 public:
15 virtual Boolean
16      provideEnterSupport ( IDMTargetEnterEvent &event );
17
18 };
```

Lines 5 through 11 declare IDMItem as the base class for objects of a specialized class named ABitmapItem. Objects of this class provide bit-map control drop behavior when a source bit-map file is dropped on a bit-map control that is properly configured with a target handler and an ABitmapProvider.

Lines 13 through 17 define a drag item provider for a bit-map control and override provideEnterSupport so that it verifies that the dragged object is a bitmap.

The .CPP file adds the drag item provider and the target handler and uses the default target renderer.

```
:
void main()
  {
   IFrameWindow
       frame ( "C Set ++ Direct Manipulation - Sample 2" );

    IBitmapControl                              // Create an empty bit-map control.
       bmpControl (  , &frame, &frame );

// Create target handler for the bit-map control and use default renderers.
    IDMHandler::enableDropOn ( &bmpControl );

   ABitmapProvider                      // Create a bit-map drag item provider.
      itemProvider;

    bmpControl.setItemProvider( &itemProvider );// Attach provider to the bit-map control.

    bmpControl.setText( "Drop .bmp files here." );// Set the bit-map control
    frame.setClient( &bmpControl )                         // as the frame's client and
            .showModally();                                         // display the frame.
   }


ABitmapItem :: ABitmapItem ( const IDMItem::Handle &item )
   : IDMItem( item )
   {
   }

Boolean ABitmapItem :: targetDrop ( IDMTargetDropEvent & )
   {
    IBitmapControl                                                 // Get pointer to target bitmap control.
      bmpControl = (IBitmapControl )this->targetOperation ()->targetWindow();

    IString                                                        // Construct dropped .bmp file name from this item.
       fname = this->containerName() + "\\" + this->sourceName();

    struct stat                                                  // Get file size.
      buf;
    stat( fname, &buf);

    FILE                                                           // Open and read the file.
      fileptr = fopen( fname, "rb" );
   char
      buffer = new char[ buf.st_size ];
    fread( buffer, sizeof(char), buf.st_size, fileptr );

    BITMAPARRAYFILEHEADER2                            // Construct the bitmap from the file.
      array = ( BITMAPARRAYFILEHEADER2   )buffer;
   BITMAPFILEHEADER2
     header;

  if ( array->usType == BFT_BITMAPARRAY ) {      // First, see if file holds array of bitmaps.
     header = &array->bfh2;                      // It is, point to first file header in array.
  } else {
      header = ( BITMAPFILEHEADER2   )buffer;     // It isn't, point to file header at start of file.
  }
  if ( header->usType == BFT_BMAP) {              // Now check to see if this is a bitmap.

      IPresSpaceHandle                            // We can proceed, first get a presentation space.
          hps = bmpControl -> presSpace();

      if ( hps ) {
         IBitmapHandle                            // Now create the bit map from the file contents.
              hbm = GpiCreateBitmap( hps,
                                     &header->bmp2,
                                     CBM_INIT,
                                     (PBYTE) buffer + header->offBits,
                                     (BITMAPINFO2 )&header->bmp2 );
          if ( hbm ) {
              IBitmapHandle                       // Get previously dropped bit map.
                  old = bmpControl -> bitmap();

             bmpControl -> setBitmap( hbm );      // Set new one.

              GpiDeleteBitmap( old );             // Destroy old because we no longer need it.

              bmpControl -> setText( fname );     // Indicate name of dropped file.
          } else {
              bmpControl -> setText( "Couldn't create bit map!" );
          }
          bmpControl -> releasePresSpace(hps);// Release the presentation space.
      } else {
         bmpControl -> setText ( "Couldn't get PS!" );
     }
  } else {
      bmpControl -> setText( fname + " isn't a bit map!" );
  }

  delete [] buffer;                               // Free buffer.

 return  true;
  }

Boolean ABitmapProvider :: provideEnterSupport ( IDMTargetEnterEvent &event )
{
                                                  //Get handle to the target operation
 IDMTargetOperation::Handle targetOp = IDMTargetOperation::targetOperation();

 IString strTypes = targetOp->item(1)->types(); //Get the types for the drag item
   if ((strTypes.indexOf(IDM::bitmap))    ||      //If type is either bitmap or plainText
        (strTypes.indexOf(IDM::plainText)))       //(used by WPS), we can display the item.
      {
     return(true);
      }
  event.setDropIndicator(IDM::notOk);     //Type  is  unrecognized,
   return(true);                                  //set the drop indicator to prevent drop.
  }
```

First, the .CPP file creates an empty bit-map control object called **bmpControl** and then creates and attaches the handler, provider, and renderer.

Lines 24 and 25 create the bit-map control object.

Line 28 constructs a target handler, which creates a default target renderer.

Lines 30 and 31 construct a drag item provider named **itemProvider**.

Line 33 attaches the drag item provider to **bmpControl** window.

The rest of the .CPP file defines the overridden member functions, targetDrop and provideEnterSupport, for the classes that were declared in the .HPP file.

Lines 46 through 109 define targetDrop. This member function gets the dropped file, creates the bitmap, and displays the bitmap in the target window.

Lines 111 through 124 use the provideEnterSupport member function to verify that the object over the target is a bit map. (This data type verification is in addition to the RMF checking that is done by the User Interface Class Library default target renderer.) If it is not a bit map, the drop is not allowed. This function is called when a target enter event (IDMTargetEnterEvent) occurs on a target window.

Enabling a Control as a Drag Source

To enable other controls as drag sources, you must specifically create the item providers that the User Interface Class Library generates automatically for entry field, MLE, and container controls. You should:

- Derive a class from the base class IDMItem and override the generateSourceItems member function.
- Write a drag item provider class for the customized item class using the IDMItemProviderFor template class.
- Use the default source handler and renderer for the customized object.

The following code example enables the user to drag objects from a static text control.

Note: This example is not included in the \ibmcpp\samples\iclui directory.

The header file defines two classes, STextItem and MyWindow, and overrides the generateSourceItems member function.

```
⋮
#include "static.h"

class STextItem : public IDMItem {
public:

   STextItem ( IDMSourceOperation        pSrcOp );
   STextItem ( const IDMItem::Handle &item );

static Boolean
   generateSourceItems ( IDMSourceOperation  pSrcOp );
};

class MyWindow : public IFrameWindow {
public:

  MyWindow();
  ~MyWindow();

private:
  ITitle  title;
  ISetCanvas  canvas;
  IStaticText  staticText;
};
```

The .CPP file adds the drag item provider and the source handler and uses the default source renderer.

```
:
void    main()
{
    MyWindow myWin;
    IApplication::current().run();
}

MyWindow :: MyWindow( )    :
                    IFrameWindow( ID_MYWINDOW ),
                     title( this, "Static Control" ),
                     canvas( ID_CANVAS, this, this ),
                     staticText( ID_STEXT, &canvas, &canvas )
{
      setClient (&canvas);      //Set the canvas as the frame client.

      IDMHandler::enableDragFrom (&staticText); //Enable the static text for dragging from

      // Use the IDMItemProviderFor template class to create a template
     // for the static text item, and set it into the window.
      IDMItemProvider  pSTProvider = new IDMItemProviderFor< STextItem >;
      staticText.setItemProvider  (pSTProvider);

      staticText.setText ("Static Text");    //Put text into the static text control.
      setFocus ();                                            // Set the keyboard focus and show it.
     show  ();
}

MyWindow :: ˜MyWindow() {};
STextItem :: STextItem (IDMSourceOperation  pSrcOp)    :
                    IDMItem  (pSrcOp,
                                     IDM::text,
                                      (IDMItem::moveable | IDMItem::copy
                                     none)
{
      IStaticText  pSText = (IStaticText  )pSrcOp->sourceWindow(); //Get a pointer to the static text
                                                       // control from the source operation
      setContents(pSText->text());      // Store the static text within the item
      setRMFs(rmfFrom(IDM::rmLibrary, IDM::rfText));//Use the default RMF for text
 }

STextItem :: STextItem (const IDMItem::Handle &item) :
                                          IDMItem ( item ) {};

Boolean STextItem :: generateSourceItems(IDMSourceOperation  pSrcOp)
{
    STextItem  pSTItem = new STextItem (pSrcOp);      //Create the static text drag item
    pSrcOp->addItem (pSTItem);                                       //and add it to the source operation.

   return(true);
}
```

Adding Images to Drag Items

A visual image is displayed for each object while it is dragged. User Interface Class Library provides default system images or you can change the image style and provide your own images.

To change the drag image style, use the setImageStyle member function. setImageStyle is called by the generateSourceItems member function of the application's derived item class.

The following table describes the IDMImage styles and the steps you must take to use them:

IDMImage Style	Description	What to Code
systemImages	If one item is dragged, the ISystemPointerHandle::singleFile icon is used. For more than one item, the ISystemPointerHandle::multipleFile icon is used. Any images supplied with drag items are ignored.	default
allStacked	Shows each image provided in each drag item. If no images are specified, system images are used.	Attach IDMImage objects to each IDMItem object
stack3AndFade	Shows the first three images provided in the drag items and then shows a special icon that looks like the rest of the images fading out. This is optimal when the user can drag more than three items. If no images are specified, system images are used.	Attach IDMImage objects to three IDMItem objects.

You can attach IDMImage objects to IDMItem objects by using the IDMItem::setImage member function:

- In the constructor of the derived item object
- In the generateSourceItems member function

The following example adds the text I-beam pointer as an image to a derived IDMItem in its constructor:

```
MyItem::MyItem (IDMSourceOperation  pIDMSrcOp)
{
:
 IDMImage image = IDMImage(ISystemPointerHandle(
                              ISystemPointerHandle::text));
 setImage(image);
 }
```

Drag Image Resources for stack3AndFade

When the stack3AndFade style is used, User Interface Class Library uses a fade icon that looks like images fading out. If your application is shipped as a product and uses the stack3AndFade option, you will need to ensure its availability to your application.

The fade icon is stored in the User Interface Class Library resource dynamic link library DDE4U001.

If your application is dynamically linked to the User Interface Class Library take the following steps to use the fade icon:

1. Rename the resource DLL with the DLLRNAME tool shipped with the IBM C/C++ Tools compiler.

   For information on DDLRNAME, see the *IBM C/C++ Tools Compiler Utilities Reference*.

2. Call ICurrentApplication::setResourceLibrary with the new DLL name as its argument.

   See ICurrentApplication in the *IBM C/C++ Tools: User Interface Class Library Reference* for more information about setResourceLibrary.

If your application is linking with User Interface Class Library static libraries, take the following steps to use the fade icon:

1. Bind the fade icon to your application .EXE file. The fade icon, fade.ico, and its resource file, DDE4U001.RC, are in the \ibmcpp\ibmclass directory on the drive you installed the product on.

2. Call ICurrentApplication::setResourceLibrary with 0 as its argument. The parameter 0 indicates that the fade icon is in the application .EXE file.

   See ICurrentApplication of the *IBM C/C++ Tools: User Interface Class Library Reference* for more information about setResourceLibrary.

Providing Help Information

Help information is the information about how to use a product. By describing a product's choices, objects, and interaction techniques, help information can assist users in learning to use a product.

The User Interface Class Library provides an IHelpWindow class that uses the OS/2 Information Presentation Facility (IPF) to provide help information for applications. An IHelpWindow is created and associated with one of the application's main windows. The User Interface Class Library also provides an IHelpHandler class to deal with help window events. When an application window is associated with a help window, a help event is dispatched to the handlers attached to the application window.

Creating Help Information

Use the following steps to create help information in your application:

1. Define the help submenu and the help window title in your resource file. In Version 5 of the Hello World application, the help menu is defined as follows:

```
STRINGTABLE
  BEGIN
     STR_HTITLE, "C++ Hello World - Help Window"        //Help title
  END
MENU WND_MAIN
  BEGIN
     SUBMENU "~Help", MI_HELP                            //Help submenu
       BEGIN
          MENUITEM "~General help...",    MI_GENERAL_HELP
          MENUITEM "~Extended help...",   SC_HELPEXTENDED, MIS_SYSCOMMAND
          MENUITEM "~Keys help...",       SC_HELPKEYS, MIS_SYSCOMMAND
          MENUITEM "Help ~index...",      SC_HELPINDEX, MIS_SYSCOMMAND
       END
  END
```

   MI_HELP is the help menu ID. The contents of the help information are stored in an IPF file, AHELLOW5.IPF.

2. Define a help table in the resource file to establish the relationship between the menu item ID and the panel ID that is defined in the IPF file.

```
HELPTABLE HELP_TABLE
  BEGIN
    HELPITEM  WND_MAIN,            SUBTABLE_MAIN,     1
    HELPITEM  WND_TEXTDIALOG,   SUBTABLE_DIALOG,  2
  END

HELPSUBTABLE SUBTABLE_MAIN                              //Main window help subtable
  BEGIN                                          //
     HELPSUBITEM WND_HELLO, 1                          //Hello <-> help ID 1
     HELPSUBITEM WND_LISTBOX,1 2                       //List box help
     HELPSUBITEM MI_EDIT, 11                         //Edit menu
     HELPSUBITEM MI_ALIGNMENT, 111                     //Alignment menu
     HELPSUBITEM MI_LEFT, 112                        //Left menu item
     HELPSUBITEM MI_CENTER, 113                        //Center menu item
     HELPSUBITEM MI_RIGHT, 114                       //Right menu item
     HELPSUBITEM MI_TEXT, 199                        //Text menu item
  END                                            //

HELPSUBTABLE SUBTABLE_DIALOG                            //Text dialog help subtable
  BEGIN                                          //
     HELPSUBITEM DID_ENTRY, 2 1                      //Entry field <-> help ID 2 1
     HELPSUBITEM DID_OK, 2 2                        //OK button <-> help ID 2 2
     HELPSUBITEM DID_CANCEL, 2 3                      //OK button <-> help ID 2 3
  END                                            //
```

WND_HELLO is a static text control ID and MI_* are menu item IDs. Each of these IDs is related to a panel ID. The main frame window ID, WND_MAIN, is also related to a panel ID. In this example, WND_MAIN and WND_HELLO both correspond to help panel ID 100. That is, pressing the F1 key in the main window area displays the same help panel as selecting **General help...** from the **Help** menu.

3. Add a pointer help, which points to the IHelp class, into the AHelloWindow class. An AHelpHandler, which is derived from IHelpHandler, overrides the keysHelpId member function, so that the correct **Keys Help** panel is displayed when keys help is requested.

```
class IHelpHandler: public IHelpHandler
{
  protected:
    virtual  Boolean
        keysHelpId(IEvent&  evt);
};
```

4. Set the identity of the help window. The keysHelpId member function is called when the user requests the keys help function. The default action is to set the event result to zero, which indicates to IPF to do nothing. In the following example, this function is overridden and the result is set to the identity of the help window IPF is to display, in this case, the keys help panel.

   ```
   Boolean AHelpHandler :: keysHelpId(IEvent& evt)
   {
     evt.setResult(1   );
     return  true;
   }
   ```

   The number 1000 is the keys help ID defined in the AHELLOW5.IPF file.

5. Add the IPF file to the help window object. The AHELLOW5.IPF file is compiled to produce AHELLOW5.HLP and added to the help window object (pointed to by IHelp) in the following example:

   ```
   help = new IHelpWindow(HELP_TABLE,this);
   help->addLibraries("AHELLOW5.HLP");
   ```

6. Use the addLibraries member function to add a library or list of libraries to the help window object. So when you look for a help panel by panel ID, these libraries can be used. (If multiple library names are specified, they should be separated by a blank space).

7. Create a special help handler if you have a child frame window. You need to attach the handler to the child frame window so that help processes correctly. For example:

   ```
   class ChildFrameHelpHandler : public IHandler {
   typedef IHandler Inherited;
   /
       This handler enables the OS/2 Help Manager to use help tables to display
       contextual help for a child frame window (one whose parent window is not
       the desktop).    This handler should only be attached to child frame windows.
                                                                   /
   public:
   virtual ChildFrameHelpHandler
    &handleEventsFor            ( IFrameWindow  frame ),
    &stopHandlingEventsFor ( IFrameWindow  frame );
   protected:
   virtual Boolean
     dispatchHandlerEvent   (  IEvent&  evt  );
   ChildFrameHelpHandler
    &setActiveWindow            ( IEvent& evt, Boolean active = true );
   private:
   virtual IHandler
    &handleEventsFor            ( IWindow  window    ),
    &stopHandlingEventsFor ( IWindow  window    );
   };

   Boolean ChildFrameHelpHandler :: dispatchHandlerEvent ( IEvent& evt )
   {
      switch ( evt.eventId() )
     {
   ```

```
      case  WM_ACTIVATE:
           setActiveWindow(evt,  evt.parameter1().number1());
           break;
      case  WM_INITMENU:
           setActiveWindow(evt,  true);
           break;
      default:
           break;
  } /  endswitch  /

  return false;                              // Never stop processing of event
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: setActiveWindow ( IEvent& evt,
                                              Boolean active )
{
  IHelpWindow  help = IHelpWindow::helpWindow(evt.window());
  if  (help)
  {
       IFrameWindow  frame =  ;
       if  (active)
       {
            frame = (IFrameWindow )evt.window();
       }
       help->setActiveWindow(frame,  frame);
  }
  return   this;
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: handleEventsFor ( IFrameWindow  frame )
{
  IASSERTPARM(frame !=  );
  Inherited::handleEventsFor(frame);
  return   this;
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: stopHandlingEventsFor ( IFrameWindow  frame )
{
  IASSERTPARM(frame !=  );
  Inherited::stopHandlingEventsFor(frame);
  return   this;
}

IHandler& ChildFrameHelpHandler :: handleEventsFor ( IWindow  window    )
{                    // private to hide version in IHandler
  ITHROWLIBRARYERROR(IC_MEMBER_ACCESS_ERROR,
                     IErrorInfo::invalidRequest,
                     IException::recoverable);
  return   this;
}

IHandler& ChildFrameHelpHandler :: stopHandlingEventsFor ( IWindow  window    )
{                    // private to hide version in IHandler
  ITHROWLIBRARYERROR(IC_MEMBER_ACCESS_ERROR,
                     IErrorInfo::invalidRequest,
                     IException::recoverable);
  return   this;
}
```

Chapter 10. Creating Dialogs

This chapter covers the following topics:

- Standard file dialogs
- Standard font dialogs
- Message boxes

Specifying Standard File Dialog Information

The *standard file dialog* enables a user to specify a file to be opened or a file name under which current work is to be saved. It includes the ability to switch directories and logical drives. The IFileDialog class allows you to define the standard dialog for files. Figure 29 shows an example of a standard file dialog:

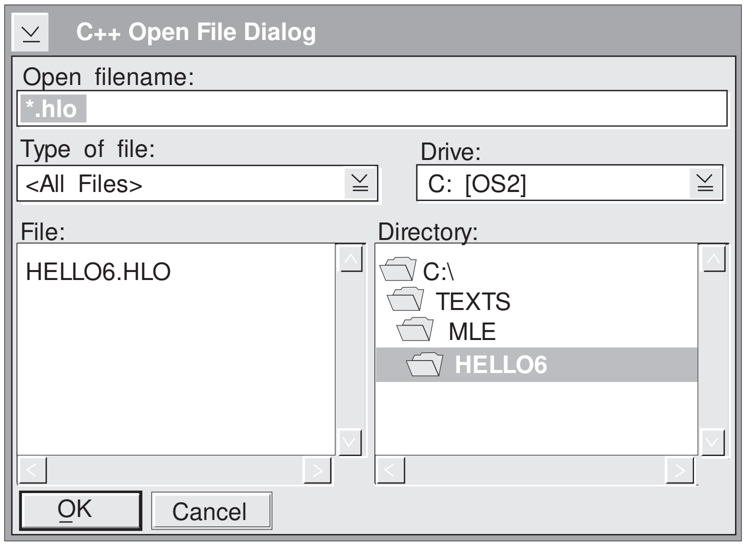


Figure 29. Example of a File Dialog

Creating a Standard File Dialog

To create a file dialog, follow these steps:

1. Set up the file dialog using the optional feature of the IFileDialog class to specify initial settings for the dialog you create. To use this feature, create an instance of the Settings class when you create the dialog, as shown in the following:

   ```
   IFileDialog::Settings fsettings;
   ```

   The Settings class has several member functions, including:

   setSaveAsDialog
   setFileName
   setPosition

   Note: The setOpenDialog is the default. If you want a **Save As** dialog, use the setSaveAsDialog member function.

   To set up the dialog, use the following statements:

   ```
   fsettings.setTitle(STR_FILEDLGT);        //Set open dialog title from resource
   fsettings.setFileName(" .hlo");          //Set FileNames to  .hlo
   ```

2. Create an instance of the IFileDialog class after setting up the dialog. Use the following statements:

   ```
   IFileDialog   fd=new IFileDialog(         //Create file open dialog
        desktopWindow(),                         //Parent is desktop
        this,                                    //Owner is me
       fsettings);                            //    with  settings
   ```

   Refer to the *IBM C/C++ Tools: User Interface Class Library Reference* for other ways to define an instance of the IFileDialog class.

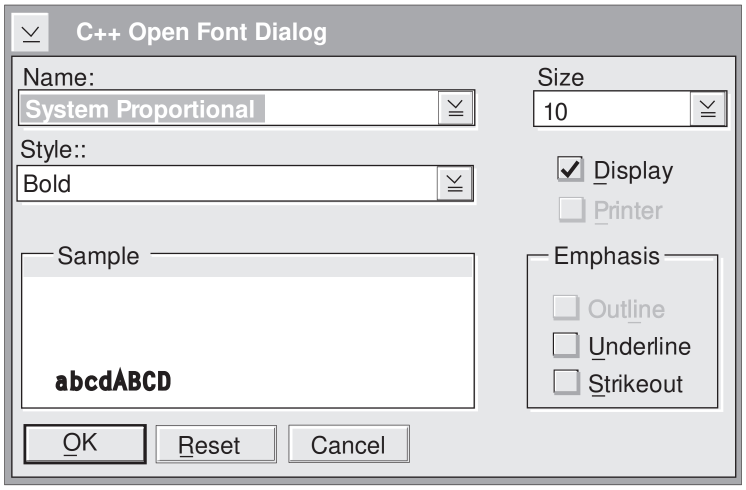
3. Test the response from the file dialog using the pressedOK member function. This member function returns true if the user ended the dialog by pressing OK.

4. Read the resulting file name from the file dialog. Use the fileName member function to return the fully qualified name that the user selected.

   For the complete sample code, see the openFile member function in the AHELLOW6.CPP file (Version 6 of the Hello World application).

Specifying Standard Font Dialog Information

The *standard font dialog* enables a user to specify a choice of font names, styles, and sizes from the range of those available in a given application. Use the IFontDialog class to handle fonts in your applications. Figure 30 shows an example of a standard font dialog.



Name: System Proportional

Size: 10

Style:: Bold

[✓ Display]

Printer

Outline

Underline

Strikeout

Figure 30. Example of a Font Dialog

Creating a Standard Font Dialog

The following example from the Hello World application shows you how to use a font dialog.

```
// <in AHELLOW6.CPP>
{                                             //                                  .
  IFont  curFont(hello);                     //Define  curFont                   .
  IFontDialog::Settings  fsettings(&curFont);//                                  .
   fsettings.setTitle(STR_FONTDLGT);          //Set Open Dialog Title from Resource   .

   IFontDialog fontd(                         //Create Font Open Dialiog          .
      (IWindow )desktopWindow(),                //    Parent is Desktop              .
      (IWindow )this,                          //    Owner is me                     .
      (IFontDialog::defaultStyle() |       //    Set default Style with only     .
     IFontDialog::bitmapOnly),               //    BitMap  Fonts                   .
     fsettings);                              //    settings                        .
   if (fontd.pressedOK())                    //Check if ok from Font open dialog     .
   {                                           //                                  .
      hello->setFont(curFont);               //Change hello font to be curFont       .
   } /  endif  /                             //                                  .
```

In the preceeding example, the font in an IStaticText control is changed to the font the user selects from an IFontDialog. This is done by:

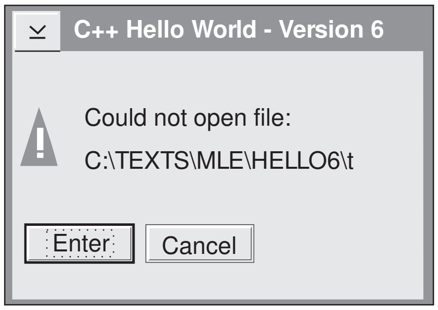
1. Creating an IFont object called curFont that represents the Font currently being used by the IStaticText control pointed to by hello.
2. Passing a pointer to the curFont object on the constructor to an IFontDialog::Settings object called fsettings.
3. Passing the fsettings object on the IFontDialog constructor.

Because fsettings is constructed using curFont, the IFontDialog initially displays the name, style, size, and emphasis associated with curFont (for example, the font currently used by the IStaticText object). If the user dismisses the IFontDialog by pressing **OK** then curFont automatically updates to reflect the font the user chose via the IFontDialog. The setFont() member function can be used to actually change the font of the IStaticText control to curFont.

Refer to "Setting and Changing Fonts" on page 82 to see how to set up a font.

Specifying Message Box Information

A *message box* is a specialized dialog box that displays information and a limited set of options to the user. The User Interface Class Library provides an IMessageBox class for displaying messages in a message box.


C++ Hello World - Version 6

Could not open file:
C:\TEXTS\MLE\HELLO6\t

Figure 31. Example of a Message Box

Creating a Message Box

You can only construct instances of the IMessageBox class by using an instance of IWindow. The IWindow instance becomes the owner of the new message box. Following is an example:

```
IMessageBox   mbox(owner);
```

The following statements create a message box:

```
1 IMessageBox msgbox(this);                     //Creates an instance of IMessageBox
2 msgbox.setTitle( IResourceId(STR_MSGBOX) ); //Load a String using its resource id res
3 msgbox.show("This is a message", IMessageBox::okButton            |
4                               IMessageBox::informationIcon    |
5                               IMessageBox::applicationModal |
6                               IMessageBox::moveable             );
```

On line 2, the setTitle member function sets the title of the message box. When given a message string, the show member function on line 3 shows the message box.

The displayButtonStatus member function in the AMCELCV.HPP file provides more examples of these statements.

Part 4. Learning from the Sample Application

Chapter 11. Introducing the Sample Applications

Sample application files are provided with the User Interface Class Library product diskettes. Install and use the samples to understand the classes. Complete listings are included in the \ibmcpp\samples\iclui directory.

About the Hello World Application

This section gives an overview of the Hello World sample application.

The Hello World application is divided into several versions, starting with the simplest form, Version 1, and building up to the most complicated form, Version 6. Each version shows you a different aspect of the User Interface Class Library.

Chapters 11–16 show you how to build an application, called "Hello World," using the User Interface Class Library. This sample application does not teach you C++ programming. If you are not familiar with the principles and aspects of C++ programming, consult the *IBM C++ Programming Guide* before continuing with this section.

Running the Sample Files

Files are included to help you compile and link each version of the Hello World sample application. READMEn.TXT files, where "n" is the version number, contain complete instructions for compiling and linking each version.

Notice that many versions of the sample application create pointers to new objects. For simplicity, the Hello World versions do not always show object cleanup. When you create pointers to objects in your applications, the objects are not destroyed unless you delete them. Therefore, it is up to you to use the C++ delete statement or to specify setAutoDeleteObject on your window objects to free the used memory when an object is no longer needed in your application.

Reviewing the Conventions Used in the Samples

The User Interface Class Library uses conventions to enhance the usability and readability of the code. The following conventions will help you as you create applications.

- Class names begin with a capital letter. For example, all classes belonging to the User Interface Class Library with a global scope begin with the letter "I," as in IApplication. If a class name consists of more than one word, the first letter of each word is capitalized, such as IFrameWindow.

  In keeping with this standard, the letter "A" was chosen as the first letter (for example, AHelloWindow) for the Hello World application-defined classes. This convention helps you distinguish the Hello World application classes from the User Interface Class Library classes. This naming convention also helps you distinguish the classes you create from those supplied by the class library.

- Member functions begin with a lowercase letter. If a member function name consists of more than one word, the first letter of each word that follows the first word is capitalized, such as setText.

- A version indicator (for example, v2 or v4) appears in columns 79-80 in the sample code comments, indicating which statements were added to enhance the previous version. The following example illustrates this convention:

```
#include <istattxt.hpp>          //IStaticText Class
#include <iinfoa.hpp>            //IInfoArea Class          v2
#include <imenubar.hpp>          //IMenuBar Class           v3
#include <ifont.hpp>             //IFont                    v3
#include <istring.hpp>           //IString Class            v4
#include <isetcv.hpp>            //ISetCanvas Class         v4
```

See "Conventions Used in This Book" on page 7 for information about other User Interface Class Library conventions.

Chapter 12. Creating an Application with a Main Window

Version 1 of the Hello World sample application shows you how to create a main window and insert a text string into it using the static text control. A *static text control* is a text field, bit map, icon, or box that you can use to label or box another control. In Version 1, the “Hello World!” text string is inserted into a static text control.

Version 1 shows you how to do the following:

1. Create the main window
2. Create a static text control
3. Set the focus and show the main window

The main window for Version 1 of the application looks like this:

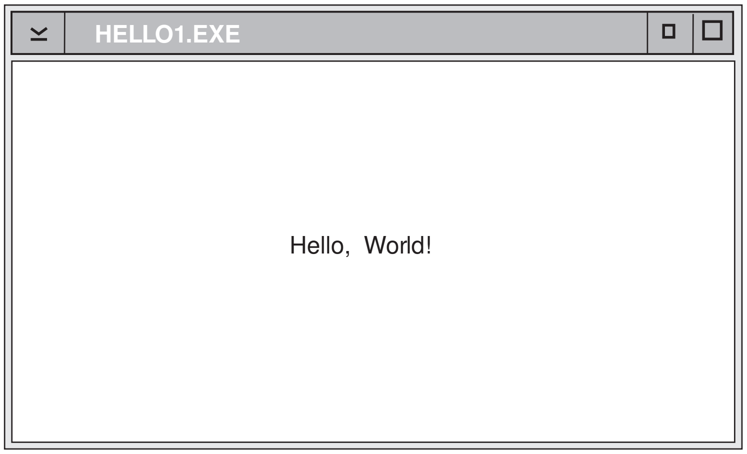


Figure 32. Version 1 of the Hello World Application

Establishing the Version 1 Window-Parent Relationships

Figure 33 shows the relationships between the objects built for Version 1 of the Hello World application:

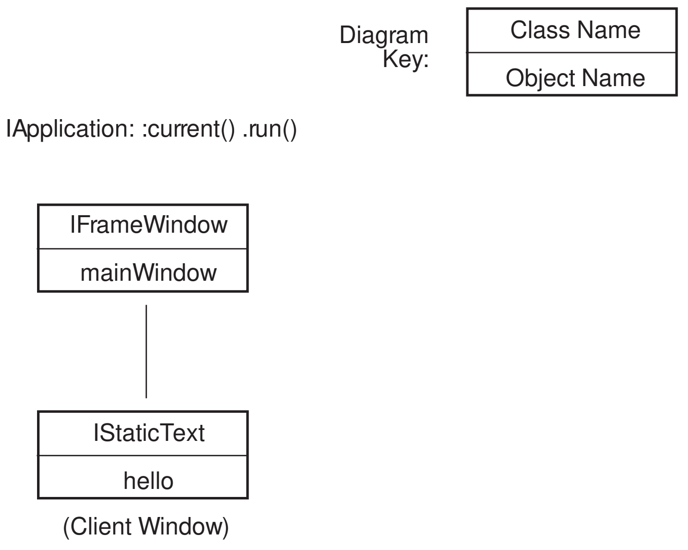


Figure 33. Window-Parent Relationship Diagram, Version 1

As the figure shows, Version 1 creates two objects: a main window and a static text control. The mainWindow object is the main window of the Hello World application.

The static text control, the hello object, is an instance of the IStaticText class. The phrase “(Client Window)” indicates that the static text control displays in the main window’s client area. In this case, the client area is that part of the primary or main window inside the borders and below the title bar. In general, all space not used by the frame and its extensions belongs to the client area.

Listing the Version 1 Files

The AHELLOW1.CPP file contains the source code for the main procedure. The tasks performed by this code are described in “Exploring Version 1” on page 150 and its related sections.

File	Type of Code
AHELLOW1.CPP	Source code for the main procedure
AHELLOW1.DEF	Module definition file for HELLO1.EXE

The Primary Source Code File

AHELLOW1.CPP contains the source code used for Version 1. Here is a listing of the source code:

```
                                              //Include IBM UI class headers:
#include <iapp.hpp>                           //IApplication Class
#include <istattxt.hpp>                       //IStaticText Class
#include <iframe.hpp>                         //IFrameWindow Class Header

//
// main   - Application entry point
//
void main()                                   //Main procedure with no parameters
{
  IFrameWindow   mainWindow=new               //Create our main window on the desktop
     IFrameWindow( x1   );                    //   Pass in our Window ID

  IStaticText   hello=new IStaticText(        //Create static text control with
     x1 1 , mainWindow, mainWindow);          //   mainWindow as parent & owner
  hello->setText("Hello, World!");            //Set text in Static Text Control
  hello->setAlignment(                        //Set Alignment to Center in both
     IStaticText::centerCenter);              //   directions

  mainWindow->setClient(hello);               //Set hello control as Client Window
  mainWindow->setFocus();                     //Set focus to main window
  mainWindow->show();                         //Set to show main window

  IApplication::current().run();              //Get the current application and
                                              // run it
} /  end main  /
```

Hello World — Version 1

The Module Definition File

A module definition (.DEF) file is created to define certain aspects of the application to the linker. This file provides the following information:

NAME	Application name and type	
DESCRIPTION	Short description of the application	
CODE	Information about the attributes for the code segment, including:	
	LOADONCALL	Specifies that the code segment is loaded when called
	MOVEABLE	Specifies that the code segment is moveable
DATA	Information about the data segment, including:	
	MOVEABLE	Specifies that the data segment is moveable
	MULTIPLE	Causes a data segment to be created for each instance of the executable code

AHELLOW1.DEF, the module definition file for Version 1, contains the following:

```
NAME      HELLO1          WINDOWAPI

DESCRIPTION 'Hello World Sample C++ Program - Version 1'

CODE      LOADONCALL  MOVEABLE
DATA      MOVEABLE    MULTIPLE
```

Exploring Version 1

The following sections describe each of the tasks performed by Version 1 of the Hello World application.

Creating the Main Window

The first task creates the main window, an instance of the IFrameWindow class, for the application. To make this class available, the application must include the IFRAME.HPP library header file, as follows:

```
<in AHELLOW1.CPP>
⋮
 4 #include <iframe.hpp>                        //IFrameWindow Class Header
⋮
```

Now that the IFrameWindow class is available, a variable, in this case mainWindow, is defined as a pointer to a new instance of this class. This creates the main window of the application. For example:

```
<in AHELLOW1.CPP>
⋮
{
  IFrameWindow   mainWindow=new              //Create our main window on the desktop
     IFrameWindow( x1   );                   //   Pass in our Window ID
⋮
```

The window ID is assigned the hexadecimal value 0x1000.

Creating a Static Text Control

Next, create a static text control for the “Hello, World!” text string. Because this control is an instance of the IStaticText class, another library header file, ISTATTXT.HPP, must be included as follows:

```
<in AHELLOW1.CPP>
⋮
#include <istattxt.hpp>                      //IStaticText Class
⋮
```

Now, define another variable, hello, as a pointer to a new instance of the IStaticText class, which creates a static text control. Use the following code:

```
<in AHELLOW1.CPP>
⋮
  IStaticText   hello=new IStaticText(    //Create static text control with
      x1 1 , mainWindow, mainWindow);     //   mainWindow as parent & owner
⋮
```

The control ID is assigned the hexadecimal value 0x1010.

The argument that follows the hexadecimal value identifies the parent of the static text control, represented by the mainWindow variable. This positions the static text control in relation to the main window and displays it on top of the main window.

The last argument identifies the main window as the owner of the static text control. Controls notify their owner windows when significant events take place by using command, help, or control events. In this case, if an action is performed on the static text control, such as modifying its text string, that action is reported to the main window, which is specified as the owner. In Version 1, no actions can be performed on the static text control, but they can in Versions 2 through 6.

Setting a Text String for the Static Text Control

After the static text control is created, give it a static text string. The IStaticText class is derived from the ITextControl class, and thus inheriting its functions. One of those functions, setText, defines the text string for the static text control. For example:

```
<in AHELLOW1.CPP>
⋮
hello->setText("Hello, World!");          //Set text in Static Text Control
⋮
```

Aligning the Static Text Control

Next, the setAlignment member function of the IStaticText class aligns the text string in the static text control. In this example, it is centered both horizontally and vertically.

```
<in AHELLOW1.CPP>
⋮
hello->setAlignment(                      //Set Alignment to Center in both
  IStaticText::centerCenter);             //    directions
⋮
```

If you do not align the text string, the default places it in the upper left corner of the client area.

Setting Static Text Control as the Client Window

Next, designate the static text control as the frame's client window so that the "Hello, World!" text string displays in the main window's client area. Use the setClient member function of the IFrameWindow class, as follows:

```
<in AHELLOW1.CPP>
⋮
mainWindow->setClient(hello);             //Set hello control as Client Window
⋮
```

The frame's client window is the window corresponding to the client area, which is the rectangular portion of the frame window not occupied by the other frame controls (for example, title bar, window border, and minimize and maximize buttons). Setting the static text control as the client window causes it to occupy the entire client area and to be aligned within the boundaries of that area. When the user resizes the main window, the client area (static text control in this example) grows or shrinks, but the frame and its extensions remain the same size.

Setting the Focus and Showing the Main Window

The next two tasks are:

- Designating the main window as the active window
- Displaying the main window when running the application

These tasks use the setFocus and IWindow::show member functions:

```
<in AHELLOW1.CPP>
⋮
   mainWindow->setFocus();                //Set focus to main window
   mainWindow->show();                    //Set to show main window
⋮
```

The setFocus and IWindow::show member functions are inherited from the IWindow class. IFrameWindow is derived from IWindow. Classes inherit functions from the base classes from which they are derived. An application does not have to include those base classes. Therefore, the IWindow class does not need to be included in this application for its functions to be available.

Running the Application

The last task displays the main window and starts the user interface event processing for the application. This involves getting and dispatching window events, using the function ICurrentApplication::run(), until the application ends. This sample application accomplishes the task using member functions belonging to the IApplication and ICurrentApplication classes. Therefore, include IAPP.HPP, another library header file, as follows:

```
<in AHELLOW1.CPP>
⋮
#include <iapp.hpp>                       //IApplication Class
⋮
```

The IApplication::current member function of the IApplication class returns the current application as an instance of the ICurrentApplication class. Next, the ICurrentApplication::run member function displays the main window and starts event processing for this application, using the following code:

```
<in AHELLOW1.CPP>
⋮
   IApplication::current().run();         //Get the current application and
                                          // run it
} /  end main  /
```

Hello World — Version 1

Chapter 13. Adding a Resource File and Frame Extensions

Version 2 of the Hello World application shows you how to use a resource file and how to add frame extensions to the application window.

A *resource file* is a file that contains data used by an application, such as text strings and icons. This data is often easier to maintain in a resource file than in the source code of an application because the resource file keeps all of the application's data together in one place.

Frame extensions are controls that you can add to a frame window in addition to those that are provided for you by basic Presentation Manager frame windows. For example, in Version 2, an information area is added below the client area.

Version 2 of the Hello World application extends Version 1 by showing you how to:

- Get the "Hello, World!!" text string and text for an information area from a resource file
- Construct the main window and set the title and system menu icon from a resource file
- Create and set the information area below the client area

The main window for Version 2 of the Hello World application looks like this:

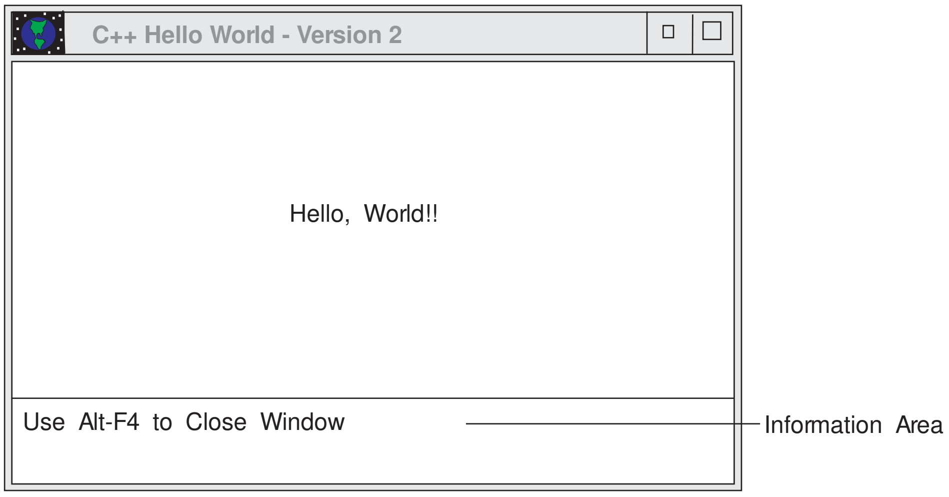


Figure 34. Version 2 of Hello World Application

Establishing the Version 2 Window-Parent Relationships

Figure 35 shows the relationships between the objects built for Version 2 of the Hello World application:

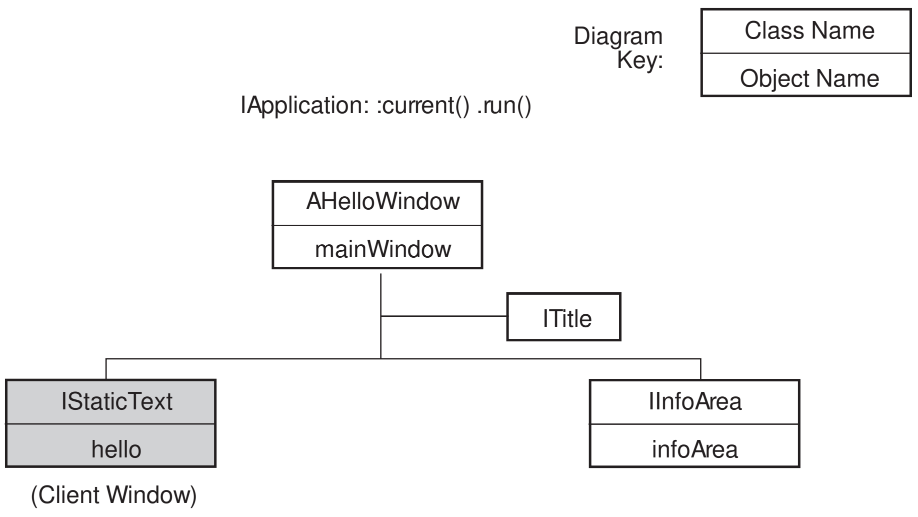


Figure 35. Window-Parent Relationship Diagram, Version 2

As the figure shows, Version 2 creates the mainWindow object as an instance of the AHelloWindow class, a subclass created for Version 2 and derived from the IFrameWindow class.

The hello object is the same as in Version 1.

In addition to the mainWindow and hello object, Version 2 provides:

- An instance of the ITitle class for the window title.
- An infoArea object in the main window. The infoArea oject is an instance of the IInfoArea class that displays text in an information area.

Listing the Version 2 Files

The following files contain the code used to create Version 2:

File	Type of Code
AHELLOW2.CPP	Source code for the main procedure and window constructor
AHELLOW2.HPP	Header file for the AHellowWindow class
AHELLOW2.H	Constant definitions file for HELLO2.EXE
AHELLOW2.RC	Resource file for HELLO2.EXE
AHELLOW2.ICO	Icon file for HELLO2.EXE
AHELLOW2.DEF	Module definition file for HELLO2.EXE

The Primary Source Code File

The AHELLOW2.CPP file contains the source code for the main procedure and the window constructor. If columns 79-80 contain a v2 or a period, then this source line was modified or added in this version. The tasks performed by this code are described in “Exploring Version 2” on page 159 and its related sections.

The AHelloWindow Class Header File

Althought the AHELLOW2.HPP file is not a User Interface Class Library header file, it is the type of header file that you would create for your own classes. In this case, it contains the class definition and interface specifications for the AHelloWindow class, a subclass of IFrameWindow that was created specifically for this application.

The Constant Definitions File

AHELLOW2.H contains the constant definitions for this application. These constants and their definitions provide the IDs for the application main window, controls, and text strings. They are required because, in this version of the Hello World application, the text strings are pulled in from a resource file.

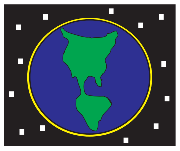
The Resource File

Version 2 of the Hello World application provides a resource file, AHELLOW2.RC. This resource file assigns an icon and three text strings to the constants defined in the AHELLOW2.H file shown in “The Constant Definitions File.” AHELLOW2.H is included in this resource file so the icon and text strings can be associated with the appropriate IDs.

Hello World — Version 2

The Icon File

AHELLOW2.ICO is used as both the title bar icon and the icon that displays when the application is minimized. We do not provide a listing for the AHELLOW2.ICO file, but this is how the icon appears when minimized:



Hello World Icon

Figure 36. Hello World Icon

The Module Definition File

The AHELLOW2.DEF file is required for the same reasons that AHELLOW1.DEF is needed for Version 1. Create a module definition file to define certain aspects of the application to the linker.

The only difference between the .DEF file used in Version 1 and Version 2 is the change in the version number.

```
NAME      HELLO2       WINDOWAPI

DESCRIPTION 'Hello World Sample C++ Program - Version 2'

CODE      LOADONCALL  MOVEABLE
DATA      MOVEABLE    MULTIPLE
```

Discussing the Advantages of the C++ File Structure

In Version 1, all of the source code was intentionally put in the AHELLOW1.CPP file to make that version of the application simple. However, for Version 2, the source code has been distributed among a variety of files to show that you can structure your applications this way.

First, the AHelloWindow class, the subclass of IFrameWindow, is defined in the header file (AHELLOW2.HPP). Putting the class definition and interface specifications in the header file separates them from their implementation in the source code (AHELLOW2.CPP). This allows the class and its specifications to be used again with other applications and to be implemented in different ways. If the class definition or interface specifications change, they change in only one place, the header file.

Similarly, the constant definitions file (AHELLOW2.H) assigns IDs to the windows and text strings in one place. Defining the constants this way allows you to use constants in a variety of places, such as the source code and the resource file, while keeping their definitions in one place. Then, if you need to change the constant definitions, you only modify the AHELLOW2.H file.

The advantage of placing the application's data in a resource file (AHELLOW2.RC) is that all of the resources are specified in one place. For example, finding and modifying text strings is easier when they are all grouped in one place, rather than searching through the source code for each one.

Exploring Version 2

The following sections describe each of the tasks performed by Version 2 of the Hello World application. Some of the tasks are the same as those performed by Version 1, but they are described again because they are performed differently in Version 2.

Creating the Main Window

One of the major differences between Version 1 of the Hello World application and Version 2 is the manner in which you create the main window. Version 1 simply creates an instance of the IFrameWindow class. However, Version 2 provides its own class, AHelloWindow, to create the main window.

The AHelloWindow class is defined in the AHELLOW2.HPP header file and is derived from the IFrameWindow class. The IFrameWindow class is defined in the IFRAME.HPP library header file. Therefore, the AHELLOW2.HPP header file contains the following line make the derivation of the AHelloWindow class from the IFrameWindow class possible:

```
// <in AHELLOW2.HPP>
⋮
#include <iframe.hpp>                    //Include the IFrameWindow class
                                         //    header
⋮
```

Note: See “Listing the Version 2 Files” on page 157 to learn about reasons for putting class definitions and interface specifications in a header file.

Hello World — Version 2

The AHELLOW2.CPP file, which contains most of the source code for the application, includes the AHELLOW2.HPP header file on line 6 to have access to the AHelloWindow class:

```
// <in AHELLOW2.CPP>
⋮
#include "ahellow2.hpp"                    //Include the AHelloWindow class          v2
                                           //    header                              v2
⋮
```

The following lines in the AHELLOW2.CPP file create the main window by using the AHelloWindow class constructor:

```
// <in AHELLOW2.CPP>
⋮
AHelloWindow mainWindow (WND_MAIN);     //Create our main window on the
                                        //    desktop
⋮
```

In Version 1, the main window is given a hexadecimal value of 0x1000 as its window ID when the main window was created. The same value is used for the window ID of the main window in Version 2. However, instead of specifying that value in the primary source code file, Version 2 uses a constant, WND_MAIN, which is defined in the AHELLOW2.H file, as follows:

```
// <in AHELLO2.H>
⋮
#define  WND_MAIN            x1               //Main  window  ID
⋮
```

Note: See "Listing the Version 2 Files" on page 157 to learn about reasons for using a constants definition file.

To have access to this definition, the primary source code file, AHELLOW2.CPP, must include the AHELLOW2.H file, as follows:

```
// <in AHELLO2.CPP>
⋮
#include "ahellow2.h"                      //Include our symbolic definitions        v2
⋮
```

Running the Current Application

When the main window is constructed, the following line gets the current application and runs it:

```
// <in AHELLOW2.CPP>
⋮
IApplication::current().run();            //Get the current application and
                                          //    run  it
⋮
```

See "Running the Application" on page 153 for a more detailed explanation.

Constructing the Main Window

After the main window has been created, next it is constructed. This section explain how to do this. Version 2 constructs the main window using the AHelloWindow class. Here is the class constructor as it is defined in the AHELLOW2.HPP header file:

```
// <in AHELLOW2.HPP>
⋮
AHelloWindow(unsigned long windowId);//Constructor for this class
⋮
```

In the primary source code file, Version 2 uses the following lines to construct the main window:

```
// <in AHELLOW2.CPP>
⋮
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow (                          //Call the IFrameWindow constructor
      IFrameWindow::defaultStyle()           //   using the default style, plus       v2
      | IFrameWindow::minimizedIcon,          //   get minimized icon from RC file     v2
      windowId)                              //   and set the main window ID
⋮
```

Two capabilities provided by the IFrameWindow class used here were not used in Version 1:

Setting the main window to the default style

The defaultStyle member function on line 24 is inherited from the IFrameWindow class. It returns the current default style that all applications use for frame windows. The current default style is either the original default style that is provided by the User Interface Class Library for frame windows, or a new default style that you establish by using the setDefaultStyle member function.

In this case, because the setDefaultStyle member function has not been used, the current default style is the same as the original default style, which provides a title bar, title bar icon, minimize button, maximize button, window border, window list, and an initial shell position for the window.

In this application, the text and icon for the title bar are specified in the resource file, AHELLOW2.RC, which is described in the following sections. The text string for the window title is included in the resource file, and the icon, AHELLOW2.ICO, is specified.

Refer to “Adding Styles” on page 63 and to the *IBM C/C++ Tools: User Interface Class Library Reference* for more information about styles.

Getting an icon to use when the main window is minimized

The minimizedIcon member function on line 27 is also inherited from the IFrameWindow class. This member function allows an application to use an icon, contained in the .EXE file and specified in the resource file, to represent the application when it is minimized on the desktop. The Hello World application provides the AHELLOW2.ICO icon file for this purpose. Refer to Figure 36 on page 158 to see how this icon appears when the main window is minimized.

Creating a Static Text Control

Another difference between Version 1 and Version 2 is the means of creating a static text control to display a text string. In Version 1, this was done simply by setting hello equal to a new instance of the IStaticText class, associating an ID with the control window (0x1010), and making the main window the parent and owner of the control, as follows:

```
// <in AHELLOW1.CPP>
⋮
IStaticText   hello=new IStaticText(    //Create static text control with
    x1 1 , mainWindow, mainWindow);       //    mainWindow as parent & owner
⋮
```

In Version 2, however, this code is divided into separate parts and placed in different files. As shown in the following lines, hello is now declared in the AHelloWindow class:

```
// <in AHELLOW2.HPP>
⋮
IStaticText       hello;                      //Define a Static Text Control to
                                                   //    keep the "Hello, World" text
                                                   //    and as the client window
⋮
```

In the AHELLOW2.CPP file, hello is used to create a new instance of a static text control:

```
// <in AHELLOW2.CPP>
⋮
hello=new IStaticText(WND_HELLO,          //Create a static text control
   this, this);                            //    Pass in this AHelloWindow as the
                                              //    parent and owner of the control
⋮
```

The WND_HELLO constant provides the ID for the static text control. All Presentation Manager windows must have a unique ID, including controls. Therefore, the AHELLOW2.CPP file must include AHELLOW2.H, because that is where this constant is defined:

```
// <in AHELLOW2.CPP>
⋮
#include "ahellow2.h"                       //Include our symbolic definitions       v2
⋮
```

With the AHELLOW2.H included, the ID is associated with the WND_HELLO constant:

```
// <in AHELLOW2.H>
⋮
#define WND_HELLO          x1 1              //Hello World window ID
⋮
```

The other two arguments (this, this) pass in the main window (this instance of the AHelloWindow class) as the parent and owner of the static text control.

See “Creating a Static Text Control” on page 151 for information about parent and owner windows.

Setting a Text String for the Static Text Control

After the static text control is created, the next task is to set text in it. Version 2 gets the text string from a resource file. To do this, it uses the setText member function, which inherits from the ITextControl class:

```
// <in AHELLOW2.CPP>
⋮
hello->setText(STR_HELLO);                 //Set text in the static text control v2
                                           // from the RC file                  v2
⋮
```

The setText member function finds this constant in the AHELLOW2.RC resource file and puts it into the static text control:

```
// <in AHELLOW.RC>
⋮
STR_HELLO,   "Hello, World!!"                 //Hello World text string       v2
⋮
```

As noted earlier, each window, even a control, must have a numeric value assigned as its ID. The STR_HELLO constant is associated with a string ID, hexadecimal value 0x1200, in the constant definition file (AHELLOW2.H). The resource file includes the constant definition file, so this constant definition is available.

```
// <in AHELLOW2.H>
⋮
#define STR_HELLO          x12               //Hello World string ID              v2
⋮
```

Aligning the Static Text Control

As in Version 1, the static text control for the client area is centered both horizontally and vertically in the static text control as follows:

```
// <in AHELLOW2.CPP>
⋮
hello->setAlignment(                    //Set the alignment to center both
   IStaticText::centerCenter);          // horizontally and vertically
⋮
```

Setting Static Text Control as the Client Window

Next, set the static text control as the client window.

See “Setting Static Text Control as the Client Window” on page 152 for an explanation of client windows.

```
// <in AHELLOW2.CPP>
⋮
setClient(hello);                       //Set the static text control as the
                                        // client window
⋮
```

Creating an Information Area

The following code creates a new instance of an information area using the IInfoArea class. This class provides a frame extension at the bottom of the client area that shows information about the application.

```
// <in AHELLOW2.CPP>
⋮
infoArea=new IInfoArea(this);           //Create the information area       v2
⋮
```

Setting the Information Area Text

Typically, the information shown in the information area pertains to the frame menu item at which the selection cursor is currently positioned. The information is taken from a resource string table. A different text string displays for each menu item, changing dynamically in the information area as the cursor moves from item to item. The information area also has a special string (called “inactive text”) that displays whenever no menu item is selected.

Version 2 sets the information area's inactive text to the same string placed in the static text control in Version 1. As a result, this text appears whenever the menu is inactive. The only difference is the setInactiveText member function is used instead of the setText member function:

```
// <in AHELLOW2.CPP>
⋮
infoArea->setInactiveText(STR_INFO);    //Set information area text from RC   v2
⋮
```

The setInactiveText member function finds the STR_INFO constant in the AHELLOW2.RC resource file and puts it into the information area:

```
// <in AHELLOW2.RC>
⋮
STR_INFO,     "Use Alt-F4 to close window"     //Information area text          v2
⋮
```

The STR_INFO constant is associated with a string ID, hexadecimal value 0x1220, in the AHELLOW2.H constant definition file. The resource file includes the constant definition file, so this constant definition is available.

```
// <in AHELLOW2.RC>
⋮
#define  STR_INFO            x122             //Information  area  string  ID              v2
⋮
```

Setting the Size of the Main Window

In Version 1, the main window's default size is used when it displays. Version 2 shows you how to change the size:

```
// <in AHELLOW2.CPP>
⋮
sizeTo(ISize(4  ,3  ));                        //Set the pixel size of main window     v2
⋮
```

This sets the size of the main window to 400 pixels wide by 300 pixels high.

Setting the Focus and Showing the Main Window

As in Version 1, the last two member functions you use are setFocus and show. However, because the AHelloWindow class is the parent and owner of the main window, you only need to specify the member function names, as follows:

```
// <in AHELLOW2.CPP>
⋮
setFocus();                                    //Set the focus to the main window
show();                                        //Show the main window
⋮
```

Hello World — Version 2

Chapter 14. Adding an Event Handler and Menu Bars

Version 3 provides a menu bar with an **Alignment** choice. A *menu bar* is the area near the top of a window, below the title bar and above the client area of the window, which contains a list of choices. By selecting the **Alignment** choice, the user can display a pull-down menu and align the "Hello, World!!!" text string to the left, right, or center. In addition, this version adds a status area to show the status of the text string, and an event handler for the menu bar and the pull-down menu.

In covering these topics, this chapter shows you how to:

- Create a status line to show the status of the text string alignment
- Set an event handler
- Add a menu bar
- Set an initial check mark in the pull-down menu
- Add command processing (event handling) to align a text string

The main window for Version 3 of the Hello World application looks like this:

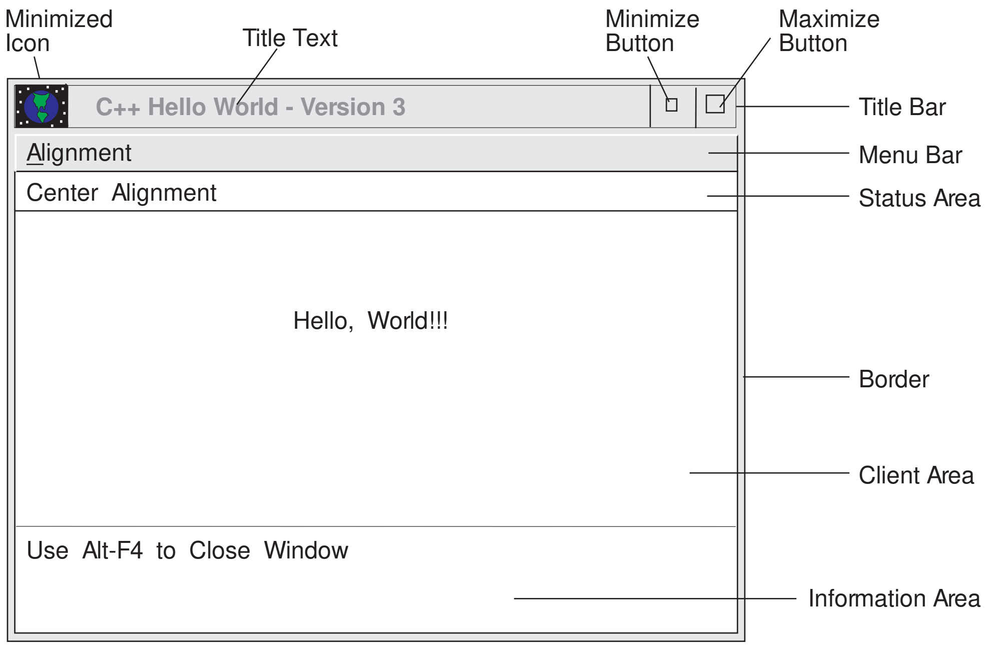


Figure 37. Version 3 of Hello World Application

Establishing the Version 3 Window-Parent Relationships

Figure 38 shows the relationships between the objects built for Version 3 of the Hello World application:

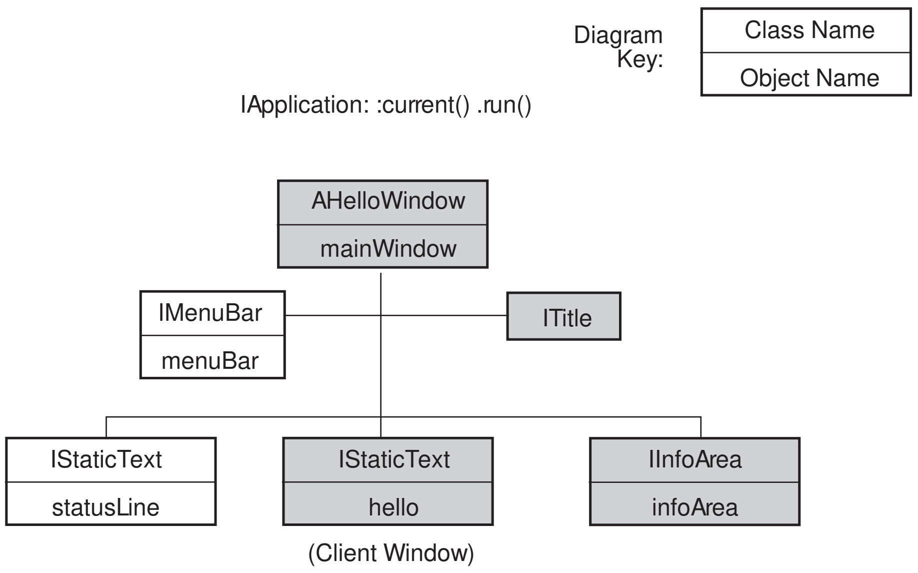


Figure 38. Window-Parent Relationship Diagram, Version 3

As the figure shows, Version 3 of the Hello World application creates the following objects:

- menuBar, which is an instance of the IMenuBar class, a subclass created for Version 3 and derived from the IMenu class.
- statusLine, an instance of the IStaticText class that creates the static text control for displaying a text string in a status area.

The instance of the ITitle class, the hello object, and the infoArea object are the same as Version 2.

Listing the Version 3 Files

The following files contain the code used to create Version 3:

File	Type of Code
AHELLOW3.CPP	Source code for the main procedure, main window constructor, and command processing
AHELLOW3.HPP	Header file for the AHellowWindow class
AHELLOW3.H	Constant definitions file for HELLO3.EXE
AHELLOW3.RC	Resource file for HELLO3.EXE
AHELLOW3.ICO	Icon file for HELLO3.EXE
AHELLOW3.DEF	Module definition file for HELLO3.EXE

The Primary Source Code File

The AHELLOW3.CPP file contains the source code for the main procedure, window constructor, and menu commands. The tasks performed by this code are described in "Exploring Version 3" on page 171 and its related sections.

The AHelloWindow Class Header File

AHELLOW3.HPP, like AHELLOW2.HPP, contains the class definition and interface specifications for the AHelloWindow class, with a few modifications for Version 3.

The Constant Definitions File

AHELLOW3.H contains the constant definitions for this application. These constants and their definitions provide the IDs for the application window components.

For Version 3, the constants definition file contains a new window ID (WND_STATUS) for the status area and three new string IDs (STR_CENTER, STR_LEFT, and STR_RIGHT) for the text strings used in the status area. In addition, menu IDs (MI_ALIGNMENT, MI_CENTER, MI_LEFT, and MI_RIGHT) have been added for the menu bar **Alignment** choice and the **Center**, **Left**, and **Right** choices in the pull-down menu.

The Resource File

Version 3 provides a resource file, AHELLOW3.RC. This resource file assigns an icon and several text strings with the constants defined in the AHELLOW3.H file shown in "The Constant Definitions File." It also contains the text strings for the menu bar. AHELLOW3.H is included in this resource file so the icon and text strings can be associated with the appropriate IDs.

The resource file for Version 3 contains two primary additions. The first is the text strings that are assigned to the new string constants that were defined in AHELLOW3.H. These text strings are used in the status area to show the state of the static "Hello, World!!!" text string in the client area. For example, when the main window is first displayed, the "Center Alignment" text string is shown in the status area.

The second addition provides the text that appears on the menu bar (**Alignment**) and pull-down menu (**Left**, **Center**, and **Right**), indicating which choices are available. Each text string is assigned to a constant, also defined in AHELLOW3.H.

The tilde (˜) to the left of the first letter in each text string indicates that those letters can be used in combination with the Alt key to provide shortcut keys for the application. For example, pressing Alt-R aligns the “Hello, World!!!” text string on the right side of the main window, just as if the **Right** choice had been selected from the pull-down menu.

The Icon File

AHELLOW3.ICO is used as both the title bar icon and the icon that displays when the application is minimized. We do not provide a listing for the AHELLOW3.ICO file. This icon is the same as for Version 2. Refer to Figure 36 on page 158 to see how this icon appears.

The Module Definition File

The AHELLOW3.DEF file is required to define certain aspects of the application to the linker.

The only difference between the Version 3 .DEF file and Version 1 and Version 2 .DEF files is the change in the version number.

```
NAME      HELLO3        WINDOWAPI

DESCRIPTION 'Hello World Sample C++ Program - Version 3'

CODE      LOADONCALL  MOVEABLE
DATA      MOVEABLE    MULTIPLE
```

Exploring Version 3

The following sections describe each of the tasks performed by Version 3 of the Hello World application that have not been described for previous versions.

Creating a Status Line

The status line shows the text string alignment status. Use the IStaticText class to create the static text control to display a text string in a status area. The *status area* is a small rectangular area that is usually located at the top of a window, below the menu bar. As shown in the following lines, statusLine is declared in the AHelloWindow class declaration in the header file:

```
// <in AHELLOW3.HPP>
⋮
IStaticText      statusLine;            //Status Line at top of client window v3
⋮
```

In the AHELLOW3.CPP file, statusLine is set equal to the IStaticText library class to create a static text control for the status area and to pass in the main window, this instance of the AHelloWindow class, as the parent and owner of this control:

```
// <in AHELLOW3.CPP>
⋮
statusLine=new IStaticText                //Create Status Area using Static Text v3
  (WND_STATUS, this, this);                //    Window ID, Parent, Owner Parameters.
⋮
```

The WND_STATUS constant provides the window ID for this static text control. This constant is defined in AHELLOW3.H.

Adding Text for a Status Line

The status area text strings are specified in the resource file, as shown in the following code:

```
// <in AHELLOW3.RC>
⋮
STR_CENTER, "Center Alignment"                 //Status Line Text - Center     v3
STR_LEFT,     "Left Alignment"                //Status Line Text - Left         .
STR_RIGHT,    "Right Alignment"               //Status Line Text - Right       v3
⋮
```

The following code gets the “Center Alignment” text string from the resource file and centers it in the static text control for the status area:

```
// <in AHELLOW3.CPP>
⋮
statusLine->setText(STR_CENTER);    //Set Status Text to "Center" from Res .
⋮
```

Specifying the Location and Height of the Status Area

Use the IFrameWindow member function addExtension to specify where the status area is positioned and how high it is. For example:

```
// <in AHELLO3.CPP>
⋮
addExtension(statusLine,                        //Add Status Line above the client        .
   IFrameWindow::aboveClient,                  //   and specify the location                  .
   IFont(statusLine).maxCharHeight()); //   and specify height                          v3
⋮
```

The aboveClient argument of the Location enumeration, on line 45, specifies that the static text control displays the status area above the client window.

The maxCharHeight member function, on line 46, returns the status area's maximum height, based on the current font.

Setting AHelloWindow as the Event Handler

In Version 3, the AHelloWindow class is derived from both the IFrameWindow and the ICommandHandler classes. This is necessary because, for the first time, this application handles events, in this case, the commands that align the “Hello, World!!!” text string.

The next line of code contains the handleEventsFor member function of the ICommandHandler class. Use this member function to set the event handler for the application. In this case, the this argument is specified, setting this instance of the AHelloWindow class as the event handler for the Hello World application:

```
// <in AHELLO3.CPP>
⋮
handleEventsFor(this);                          //Set self as event handler (commands)v3
⋮
```

This member function is available because the header file includes the ICMDHDR.HPP library header file, which contains the ICommandHandler class.

```
// <in AHELLOW3.HPP>
⋮
#include <icmdhdr.hpp>                  //Include ICommandEvent & ICommandHandler         v3
⋮
```

Creating a Menu Bar

Now you can create the “Alignment” menu bar to display the **Left**, **Center**, and **Right** choices. On line 30 of the header file, menuBar is defined as an instance of the IMenuBar class.

```
// <in AHELLOW3.HPP>
⋮
IMenuBar        menuBar;          //Define Menu Bar                     v3
⋮
```

Use menuBar to create a new instance of that class in the main window, as follows:

```
// <in AHELLOW3.CPP>
⋮
menuBar=new IMenuBar(WND_MAIN, this); //Create Menu Bar for main window        .
⋮
```

The WND_MAIN argument identifies the following menu in the AHELLOW3.RC resource file:

```
// <in AHELLOW3.RC>
⋮
MENU WND_MAIN                                      //Main Window Menu (WND_MAIN) v3
  BEGIN
    SUBMENU "~Alignment", MI_ALIGNMENT             //Alignment Submenu           v3
      BEGIN
        MENUITEM "~Left",    MI_LEFT               //Left Menu Item              v3
        MENUITEM "~Center", MI_CENTER              //Center Menu Item            v3
        MENUITEM "~Right",   MI_RIGHT              //Right Menu Item             v3
      END
  END
```

This menu puts one choice, **Alignment**, on the menu bar, and provides a pull-down menu with three choices: **Left**, **Center**, and **Right**.

In addition, the MI_ALIGNMENT, MI_LEFT, MI_CENTER, and MI_RIGHT menu item attributes correspond to those in the resource file’s string table:

```
// <in AHELLOW3.RC>
⋮
MI_ALIGNMENT,"Alignment Menu"              //InfoArea - Alignment Menu    v3
MI_CENTER,    "Set Center Alignment"       //InfoArea - Center Menu        .
MI_LEFT,      "Set Left Alignment"         //InfoArea - Left Menu          .
MI_RIGHT,     "Set Right Alignment"        //InfoArea - Right Menu        v3
⋮
```

When the user moves the selection cursor over each menu item, the text string associated with that menu item displays in the information area below the client window. For example, when the cursor is on the **Right** menu item, the text string “Set Right Alignment” appears in the information area.

Hello World — Version 3

Setting an Initial Check Mark in the Pull-down Menu

The pull-down menu that displays when the **Alignment** choice is selected on the menu bar contains three choices for aligning the "Hello, World!!!" text string: **Left**, **Center**, and **Right**. Because this text string is aligned in the center of the client area when the application is created, a check mark should display next to the **Center** choice the first time the pull-down menu displays.

The checkItem member function of the IMenuBar class allows you to place a check mark on a pull-down menu choice. The following line places a check mark on the **Center** choice:

```
// <in AHELLOW3.CPP>
⋮
menuBar->checkItem(MI_CENTER);      //Place Check on Center Menu Item          .
⋮
```

The MI_CENTER constant is defined in the AHELLOW3.RC resource file as the "Center" text string for the menu. Do not confuse this with the MI_CENTER menu item attribute defined in the string table, which is used only by the information area.

Adding Command Processing to Align a Text String

This section shows you how to associate commands with the menu items to align the text string.

An example of the command processing for one of the menu items follows. This code is used to left-align the "Hello, World!!!" text string in the client window:

```
// <in AHELLOW3.CPP>
⋮
case MI_LEFT:                               //Code to Process Left Command Item      v3
   hello->setAlignment(                     //Set alignment of hello text to           .
     IStaticText::centerLeft);           //   center-vertical,  left-horizontal     .
   statusLine->setText(STR_LEFT);        //Set Status Text to "Left" from Res     .
  menuBar->uncheckItem(MI_CENTER);   //Uncheck  Center  Menu  Item                   .
   menuBar->checkItem(MI_LEFT);          //Place Check on Left Menu Item             .
   menuBar->uncheckItem(MI_RIGHT);     //Uncheck Right Menu Item                    .
   return(true);                            //Return command processed                 .
  break;                                    //                                          v3
⋮
```

This code does the following:

- Uses the setAlignment member function to center the static text control vertically and align it on the left horizontally
- Sets the appropriate text string in the status area (Left Alignment)
- Uses the uncheckItem member function to remove any existing check marks from the **Center** and **Right** menu items
- Uses the checkItem member function to set a check mark on the **Left**
- Returns true and ends

Chapter 15. Adding Dialogs and Push Buttons

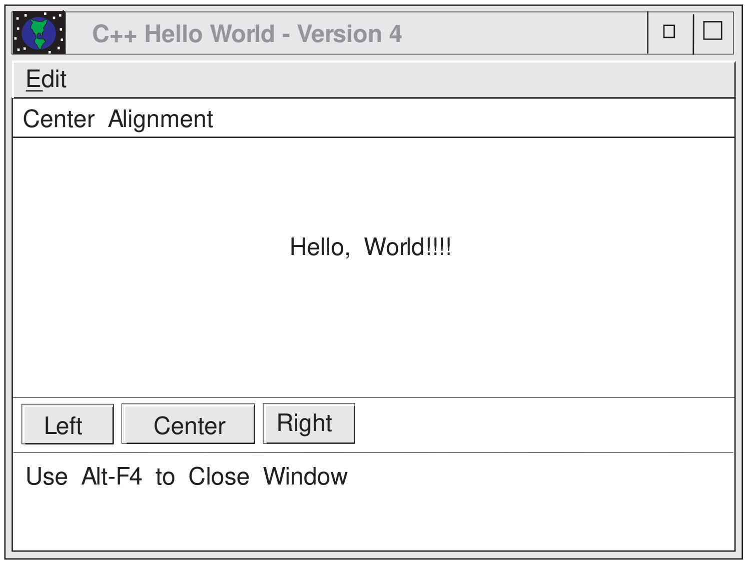
This section shows you how to:

- Modify the menu bar
- Create a dialog box
- Set push buttons in a set canvas

Version 4 modifies menu bar and the pull-down menu in the following ways:

- Creates an **Edit** choice on the menu bar
- Moves the **Alignment** choice from the menu bar to the pull-down menu
- Moves the menu items associated with the **Alignment** choice (**Left**, **Center**, and **Right**) from the pull-down menu into a cascaded menu that displays when the **Alignment** choice is selected. These items still align the "Hello, World!!!!" text string in the client window. However, the commands assigned to these menu items are also assigned to accelerator keys so the keyboard can bypass the menu choices and establish the text alignment.
- Adds a **Text...** choice on the pull-down menu. Selecting this choice displays a dialog box that contains an entry field in which the "Hello, World!!!!" text string can be edited.

The main window for Version 4 of the Hello World application looks like this:



Center Alignment

Hello, World!!!!

Use Alt-F4 to Close Window

Figure 39. Version 4 of Hello World Application

Establishing the Version 4 Window-Parent Relationships

Figure 40 shows the relationships between the objects built for Version 4 of the Hello World application:

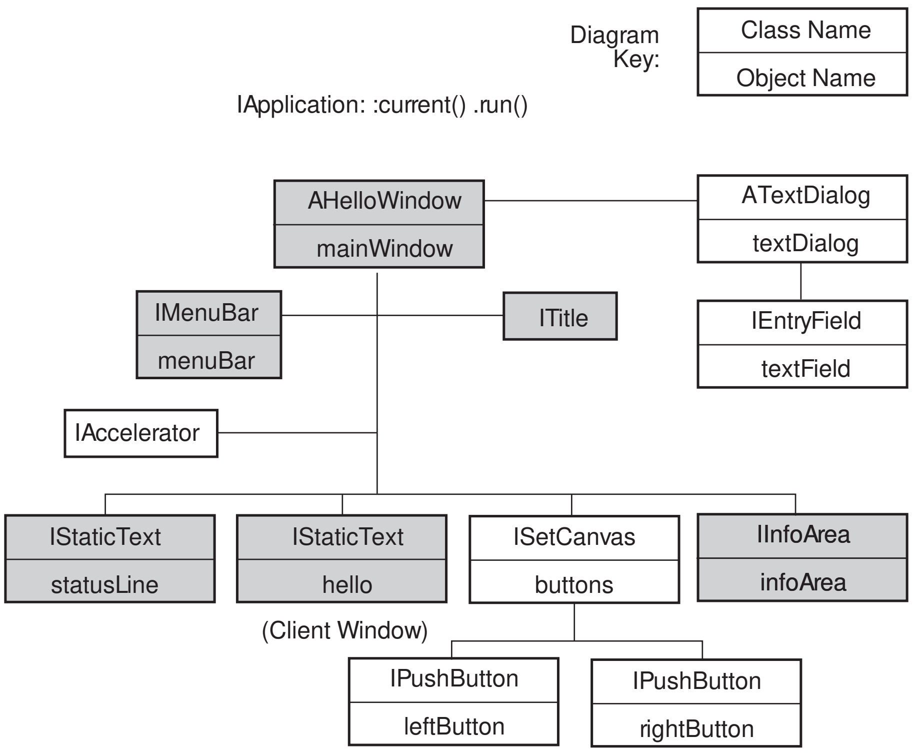


Figure 40. Window-Parent Relationship Diagram, Version 4

As the figure shows, Version 4 of the Hello World application creates the following objects:

- textDialog, which is an instance of the ATextDialog class, a subclass created for Version 4 and derived from the IFrameWindow and ICommandHandler classes
- textField, which is an instance of the IEntryField class that creates and manages an entry field control
- buttons, which is an instance of the ISetCanvas class that organizes push buttons
- leftButton and a rightButton, which are instances of the IPushButton class that creates and manages the push button control window

In addition to these objects, Version 4 provides an instance of the IAccelerator class to allow access to the tables of accelerator or shortcut keys and the associated command IDs that are stored in resource files.

Listing the Version 4 Files

The following files contain the code used to create Version 4:

File	Type of Code
AHELLOW4.CPP	Source code for the main procedure, main window constructor, and command processing
AHELLOW4.HPP	Header file for the AHellowWindow class
AHELLOW4.H	Constant definitions file for HELLO4.EXE
ADIALOG4.CPP	Source code to create the ATextDialog class
ADIALOG4.HPP	Header file for the ATextDialog class
AHELLOW4.RC	Resource file for HELLO4.EXE
AHELLOW4.ICO	Icon file for HELLO4.EXE
ADIALOG4.DLG	Dialog resource source file for HELLO4.EXE
ADIALOG4.RES	Dialog resource file for HELLO4.EXE
AHELLOW4.DEF	Module definition file for HELLO4.EXE

The Primary Source Code File

The AHELLOW4.CPP file contains the source code for the main procedure, window constructor, and menu commands. The tasks performed by this code are described in "Exploring Version 4" on page 180 and its related sections.

The AHelloWindow Class Header File

AHELLOW4.HPP, like AHELLOW3.HPP, contains the class definition and interface specifications for the AHelloWindow class, with a few modifications for Version 4.

Hello World — Version 4

The Constant Definitions File

AHELLOW4.H contains the constant definitions for this application. These constants and their definitions provide the IDs for the application window components.

For Version 4, the constants definition file contains new window IDs (WND_TEXTDIALOG and WND_BUTTONS) for the text dialog and the push button controls on the canvas, respectively. It also contains new string IDs (STR_CENTERB, STR_LEFTB, and STR_RIGHTB) for the text strings used in the push buttons. In addition, menu IDs (MI_EDIT and MI_TEXT) have been added for the menu bar **Edit** choice and the **Text** choice in the pull-down menu.

The Text Dialog Source Code File

The ADIALOG4.CPP file contains the source code for the text dialog window constructor, the ATextDialog class, created for Version 4.

The ATextDialog Class Header File

The ADIALOG4.HPP file contains the class definition and interface specifications for the ATextDialog class.

The Resource File

Version 4 provides a resource file, AHELLOW4.RC. This resource file assigns an icon and several text strings with the constants defined in the AHELLOW4.H file shown in "The Constant Definitions File." It also contains resources for the menu bar, the accelerator keys and the text dialog.

AHELLOW4.H is included in this resource file so the icon, text strings, and other resources can be associated with the appropriate IDs. OS.H is included because it is the top level include file that includes all the files necessary for writing an OS/2 application.

The resource file for Version 4 contains two primary additions. The first is the accelerator table of text strings assigned to the function keys. These text strings are used in the cascaded menu to show the accelerator, or shortcut, key assignments. For example, with these assignments and the command processing in AHELLOW4.CPP, when the user presses the **F7** key, it is the same as if they select the **Left** choice in the cascaded menu.

The second addition is an rcinclude statement that includes the text dialog template. See "The Text Dialog Template" on page 179 for information about that file.

The Icon File

AHELLOW4.ICO is used as both the title bar icon and the icon that displays when the application is minimized. We do not provide a listing for the AHELLOW4.ICO file. This icon is the same as for Version 2. Refer to Figure 36 on page 158 to see how this icon appears.

The Text Dialog Template

ADIALOG4.DLG contains the template used to build the text dialog. Here is that template:

```
DLGINCLUDE 1 "AHELLOW4.H"

DLGTEMPLATE WND_TEXTDIALOG LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "Hello World Edit Dialog", WND_TEXTDIALOG, 17, 22, 137, 84,
            WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
    BEGIN
        DEFPUSHBUTTON   "OK", DID_OK, 6, 4, 4 , 14
        PUSHBUTTON      "Cancel", DID_CANCEL, 49, 4, 4 , 14
        LTEXT           "Edit Text:", DID_STATIC, 8, 62, 69, 8
        ENTRYFIELD      "", DID_ENTRY, 8, 44, 114, 8, ES_MARGIN
    END
END
```

The Text Dialog Resource File

ADIALOG4.RES is created by the resource compiler as input to HELLO4.EXE.

The Module Definition File

The AHELLOW4.DEF file is required to define certain aspects of the application to the linker.

The only difference between the Version 4 .DEF file and .DEF files for the previous versions is the change in the version number.

The only difference between the two .DEF files used in Version 1 and Version 4 is the change in the version number.

```
NAME    HELLO4     WINDOWAPI

DESCRIPTION 'Hello World Sample C++ Program - Version 4'

CODE    LOADONCALL  MOVEABLE
DATA    MOVEABLE    MULTIPLE
```

Exploring Version 4

The following sections describe each of the tasks performed by Version 4 of the Hello World application that have not been described for previous versions.

Modifying the Menu Bar

For Version 4, there are several modifications to the menu bar and its associated pull-down menu. First, change the Version 3 **Alignment** item on the menu bar to **Edit**, using the following code:

```
// <in AHELLOW4.RC>
⋮
MENU WND_MAIN                                    //Main Window Menu (WND_MAIN) v3
  BEGIN
     SUBMENU "~Edit", MI_EDIT                   //Edit Submenu                 v4
⋮
```

Adding a Cascaded Menu

Next, add the **Alignment** choice to the pull-down menu. This choice displays a cascaded menu, which is a menu that displays to the right of the pull-down menu. An arrow next to the **Alignment** choice indicates that a cascaded menu will display when it is selected, as shown in Figure 41 on page 181. The menu is also defined in the following code:

```
// <in AHELLOW4.RC>
⋮
BEGIN
   SUBMENU "~Alignment", MI_ALIGNMENT          //Alignment Submenu          v3
⋮
```

Adding Accelerator, or Shortcut, Keys

The accelerator, or shortcut, keys are function keys that perform the same actions as menu items. In Version 3, the **Left**, **Center**, and **Right** choices appeared as items in the pull-down menu. Now, add these choices to the cascaded menu and assign a function key each choice. To add a function key as an accelerator key, use the following code:

```
// <in AHELLOW4.RC>
⋮
BEGIN
   MENUITEM "~Left\tF7",     MI_LEFT        //Left Menu Item - F7 Key       v4
   MENUITEM "~Center\tF8", MI_CENTER      //Center Menu Item - F8 Key     v4
   MENUITEM "~Right\tF9",    MI_RIGHT       //Right Menu Item - F9 Key      v4
END
⋮
```

The corresponding accelerator key displays to the right of each choice. The **F7**, **F8**, and **F9** keys can be used in place of the **Left**, **Center**, and **Right** menu items to align the “Hello, World!!!!” text string, as shown in Figure 41 on page 181.

The default processing of the accelerator style uses the accelerator that matches the frame window ID. In this example, the frame window ID is WND_MAIN. For Version 4, the following line is added to the main window constructor:

```
// <in AHELLOW4.CPP>
⋮
| IFrameWindow::accelerator,             //    Get Accelerator Table from RC file v4
⋮
```

This code gets the accelerator table from the resource file to define accelerator, or shortcut, keys for the Hello World application. Here is the code for the accelerator table:

```
// <in AHELLOW4.RC>
⋮
ACCELTABLE WND_MAIN                                    //Acc. Table for Main Window     .
  BEGIN                                        //                                   .
     VK_F7,   MI_LEFT,     VIRTUALKEY                //F7 - Left Command              .
     VK_F8,   MI_CENTER, VIRTUALKEY                   //F8 - Center Command            .
     VK_F9,  MI_RIGHT,   VIRTUALKEY                 //F9  -  Right  Command            .
  END                                          //                                 v4
⋮
```

Adding a Pull-down Menu Choice

The final modification to the pull-down menu adds the **Text...** choice. The ellipsis (...) indicates that selecting this choice causes a dialog box to display. Use the following code to add the **Text...** choice:

```
// <in AHELLOW4.RC>
⋮
MENUITEM "~Text...", MI_TEXT                  //Text Menu Item                    v4
⋮
```

Figure 41 shows the pull-down menu choices and the cascaded menu.

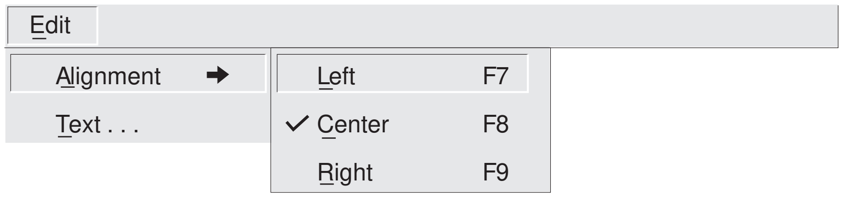


Figure 41. Cascaded Menu for Version 4 of Hello World

Hello World — Version 4

Creating a Dialog Box

As mentioned in the previous section, the **Text...** choice on the pull-down menu causes a dialog box to display. In this case, the dialog that displays is a text dialog that uses the entry field control to allow the user to edit the “Hello, World!!!!” text string. The dialog looks like this:

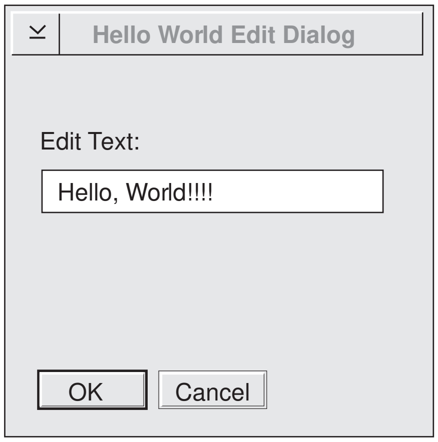


Figure 42. Text Dialog for Version 4 of Hello World

Processing the Menu Item

The following code processes the **Text...** menu item:

```
// <in AHELLOW4.CPP>
⋮
case MI_TEXT:                                   //Code to Process Text Command            v4
  {
   temp=hello->text();                     //Get current Hello text                      .
   infoArea->setInactiveText(              //Set Info Area to Dialog Active           .
       STR_INFODLG);                            //    Text from Resource File                   .
   ATextDialog   textDialog=new            //Create a Text Dialog                            .
      ATextDialog(temp,  this);            //                                                     .
   textDialog->showModally();              //Show this Text Dialog as Modal           .
   value=textDialog->result();             //Get result (eg OK or Cancel)             .
   if (value != DID_CANCEL)                     //Set new string if not cancelled          .
       hello->setText(temp);               //Set Hello to Text from Dialog             .
   infoArea->setText(STR_INFO);              //Set Info Text to "Normal" from Res     .
  delete  textDialog;                      //Delete  textDialog                               .
   return(true);                           //Return Command Processed                       .
  break;                                        //                                                    v4
  }
⋮
```

Getting the Text String for a Dialog Box

The IString class uses the temp data member to get the text string for the dialog box. The following code is added to the declaration of the command member function in AHELLOW4.CPP to accomplish this:

```
⋮
IString temp;                     //String to pass in/out from dialog    v4
⋮
```

This means the IString class must be included:

```
// <in AHELLOW4.CPP>
⋮
#include <istring.hpp>            //IString Class                        v4
⋮
```

The temp data member is set to the text string that is currently in the hello control, the static text control that contains the “Hello, World!!!!” text string.

```
// <in AHELLOW4.CPP>
⋮
temp=hello->text();               //Get current Hello text               .
⋮
```

Putting Dialog Status Text in the Information Area

Use the STR_INFODLG constant to put a text string in the information area to show that the dialog is active:

```
// <in AHELLOW4.CPP>
⋮
infoArea->setInactiveText(        //Set Info Area to Dialog Active       .
  STR_INFODLG);                   //  Text from Resource File            .
⋮
```

This constant is defined in the AHELLOW4.RC file:

```
⋮
STR_INFODLG,"Modal Edit Text Dialog Active" //Information Area String    v4
⋮
```

Creating a Dialog

Version 4 uses the textDialog data member to create an instance of the ATextDialog class, a new class created as a subclass of the IFrameWindow class. For example:

```
// <in AHELLOW4.CPP>
⋮
ATextDialog  textDialog=new       //Create a Text Dialog                 .
  ATextDialog(temp, this);        //                                     .
⋮
```

The temp data member passes the current text string to the dialog.

Hello World — Version 4

The code for the text dialog comes from the ADIALOG4.CPP file. The declaration and interface specifications for the ATextDialog class are contained in the ADIALOG4.HPP file, which is included by both the AHELLOW4.CPP and ADIALOG4.CPP files.

In addition, the dialog template is in the ADIALOG.DLG file. The AHELLOW4.RC resource file uses the following line of code to include the dialog template:

```
⋮
rcinclude adialog4.dlg                    //Text Dialog Template         v4
```

Setting Push Buttons in a Set Canvas

Use the setupButtons member function to set push buttons that can be used as an alternate way to align the "Hello, World!!!!" text string. This function is declared as follows:

```
// <in AHELLOW4.HPP>
⋮
virtual Boolean setupButtons();     //Setup Buttons                v4
⋮
```

The function is specified in AHELLOW4.CPP with no arguments, as follows:

```
⋮
setupButtons();                     //Setup  Buttons               v4
⋮
```

The setupButtons member function is defined as a member function of AHellowWindow:

```
// <in AHELLOW4.CPP>
⋮
Boolean AHelloWindow :: setupButtons()   //Setup Buttons                    .
⋮
```

Creating the Set Canvas

The buttons data member is an instance of the ISetCanvas class that sets a canvas area to position the push buttons in.

See "Creating a Set Canvas" on page 57 for more information about the ISetCanvas features described in this chapter.

```
// <in AHELLOW4.CPP>
⋮
{                                   //                                  .
  ISetCanvas      buttons;          //Define canvas of buttons         .
⋮
```

To make the ISetCanvas class available to the application, include the ISETCV.HPP library header file, as follows:

```
// <in AHELLOW4.CPP>
⋮
#include <isetcv.hpp>                //ISetCanvas Class                 v4
⋮
```

Next, the buttons data member creates a set canvas control with the main window as the parent and owner of the control. The WND_BUTTONS constant provides the window ID for this set canvas control.

```
// <in AHELLOW4.CPP>
⋮
buttons=new ISetCanvas(WND_BUTTONS,    //Create a Set Canvas for Buttons       .
   this, this) ;                       //    Parent and Owner=me               .
⋮
```

The WND_BUTTONS constant is defined in AHELLOW4.H on line 13:

```
⋮
#define WND_BUTTONS      x1 21        //Button Canvas Window ID              v4
⋮
```

Use the setMargin and setPad member functions to set the canvas margins and pad to zero. The following code shows how to do this:

```
// <in AHELLOW4.CPP>
⋮
buttons->setMargin(ISize());          //Set Canvas Margins to zero             .
buttons->setPad(ISize());             //Set Button Canvas Pad to zero          .
⋮
```

Defining the Push Buttons

Now that you have a set canvas, define three push button data members in the header file, as shown in the following code:

```
// <in AHELLOW4.HPP>
⋮
IPushButton     leftButton;          //Define Left Button                 .
IPushButton     centerButton;        //Define Center Button               .
IPushButton     rightButton;         //Define Right Button               v4
⋮
```

Creating Push Buttons

The AHELLOW4.CPP file includes the IPUSHBUT.HPP library header file and makes the IPushButton class available to Version 4. You need the data members defined in the AHELLOW4.HPP file to create three push buttons in the set canvas: **Left**, **Center**, and **Right**. Use the following code to include the AHELLOW4.HPP file:

```
// <in AHELLOW4.CPP>
⋮
#include <ipushbut.hpp>                    //IPushButton Class                    v4
⋮
```

The following code creates a new instance of the **Left** push button control and specifies that it uses the command processing associated with the MI_LEFT menu item attribute to align the “Hello, World!!!!” text string on the left side of the client window.

```
// <in AHELLOW4.CPP>
⋮
leftButton=new IPushButton(MI_LEFT,     //Create Left Push Button                .
   buttons, buttons, IRectangle(),      //   Parent, Owner=Button Canvas          .
   IPushButton::defaultStyle() |        //   Use Default Styles plus             .
  IControl::tabStop);                   //   tabStop                             .
⋮
```

Other than the data member used (centerButton is used for the **Center** push button and rightButton is used for the **Right** push button), this attribute is the only difference in the code that is used to create all three push buttons. Specify the MI_CENTER menu item attribute for the **Center** push button and MI_RIGHT for the **Right** push button.

The set canvas control is identified as the owner and parent of the push button control.

The defaultStyle member function specifies that the default style defined for the IPushButton class is to be used for this push button with one exception. The tabStop style, inherited from the IControl class, is specified so the user can tab to this push button.

Setting Text in Push Buttons

Use the setText member function to set text strings in each push button. Here is the code that sets the text in the **Left** push button:

```
// <in AHELLOW4.CPP>
⋮
leftButton->setText(STR_LEFTB);          //Set Left Button Text                    .
⋮
```

Other than the data member for which the text is set (centerButton is used for the **Center** push button and rightButton is used for the **Right** push button), the only difference between this code and the code that puts text in the other two push buttons is the STR_LEFTB constant, which associates with the appropriate text string in the AHELLOW4.RC file. Here are the text string associations for all three push buttons:

```
// <in AHELLOW4.RC>
⋮
STR_LEFTB,    "Left"                         //String for Left Button        v4
STR_CENTERB,"Center"                        //String for Center Button      .
STR_RIGHTB, "Right"                        //String for Right Button       v4
⋮
```

Hello World — Version 4

Chapter 16. Adding a Canvas, User-Created Controls, and Help

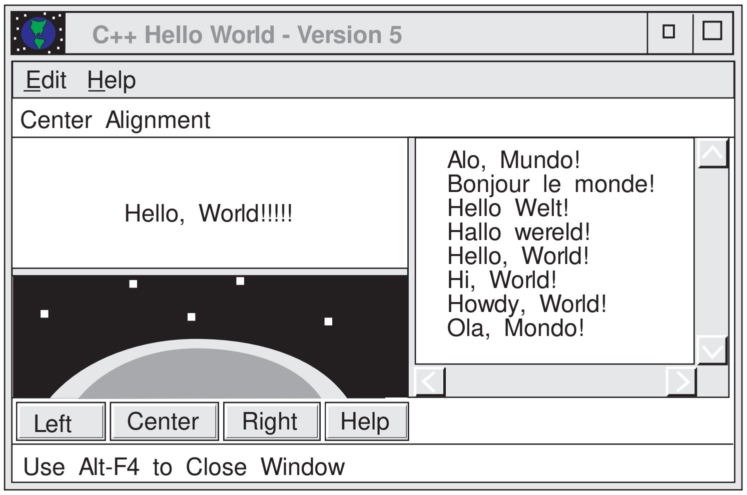
Version 5 of the Hello World application adds on to the previous versions by providing the source file for the help window. The source file for the help window contains the text and the OS/2 Information Presentation Facility (IPF) tags that produce the help information for the Hello World application.

In addition, Version 5 also uses five new member functions to create the main window.

In covering these topics, this section shows you how to:

- Construct the client window
  - Create a split canvas control as the client window
  - Add the Earth graphic to the split canvas
  - Create a list box in the client area to change the "Hello, World!!!!!" text
- Create help for the main window, dialog box, and entry fields
- Create the information area
- Set up the menu bar
- Set up the status area

The main window for Version 5 of the Hello World application looks like this:



Center Alignment

Hello, World!!!!!

Alo, Mundo!
Bonjour le monde!
Hello Welt!
Hallo wereld!
Hello, World!
Hi, World!
Howdy, World!
Ola, Mondo!

Use Alt-F4 to Close Window

Figure 43. Version 5 of Hello World Application

Establishing the Version 5 Window-Parent Relationships

Figure 44 shows the relationships between the objects built for Version 5 of the Hello World application:

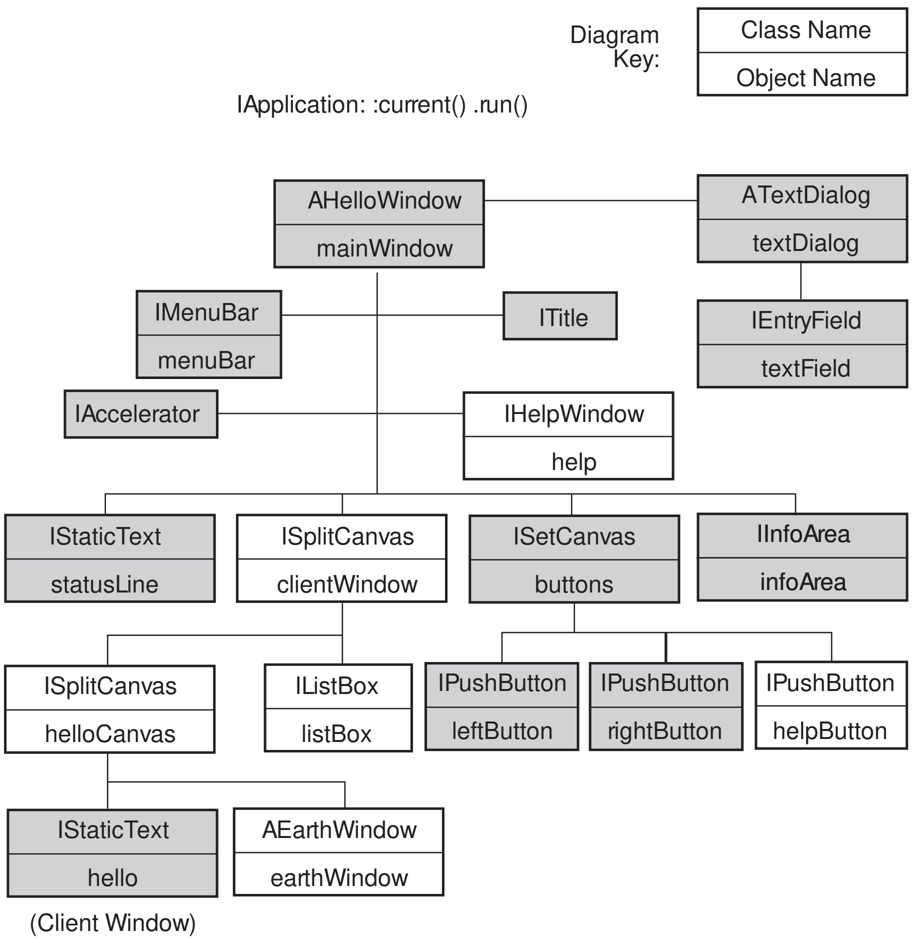


Figure 44. Window-Parent Relationship Diagram, Version 5

As the figure shows, Version 5 of the Hello World application creates the following objects:

- help, which is an instance of the IHelpWindow class that provides help for application windows using the IPF tags
- clientWindow and hellowCanvas, which are instances of the ISplitCanvas class, a control class that provides a split window
- listBox, which is an instance of the IListBox class that creates and manages the list box control window
- helpButton, which is an instance of the IPushButton class that creates and manages the push button control window
- earthWindow, which is an instance of the AEarthWindow class created for Version 5 and derived from the IStaticText and IPaintHandler classes

Listing the Version 5 Files

The following files contain the code used to create Version 5:

File	Type of Code
AHELLOW5.CPP	Source code for main procedure and AHelloWindow class
AHELLOW5.HPP	Class header file for AHellowWindow
AHELLOW5.H	Constant definitions file for HELLO5.EXE
ADIALOG5.CPP	Source code to create the ATextDialog class
ADIALOG5.HPP	Class header file for ATextDialog
AEARTHW5.CPP	Source code to create the AEarthWindow class
AEARTHW5.HPP	Class header file for AEarthWindow
AHELLOW5.RC	Resource file for HELLO5.EXE
AHELLOW5.ICO	Icon file for HELLO5.EXE
ADIALOG5.DLG	Dialog resource source file for HELLO5.EXE
ADIALOG5.RES	Dialog resource file for HELLO5.EXE
AHELLOW5.IPF	Help file for HELLO5.EXE
AHELLOW5.DEF	Module definition file for HELLO5.EXE

Hello World — Version 5

The Primary Source Code File

The AHELLOW5.CPP file contains the source code for the main procedure, window constructor, and menu commands. The tasks performed by this code are described in "Exploring Version 5" on page 194 and its related sections.

The AHelloWindow Class Header File

AHELLOW5.HPP contains the class definition and interface specifications for the AHelloWindow class, with a few modifications for Version 5.

The Constant Definitions File

AHELLOW5.H contains the constant definitions for this application. These constants and their definitions provide the IDs for the application window components.

The Text Dialog Source Code File

The ADIALOG5.CPP file contains the source code for the text dialog window constructor, the ATextDialog class, created for Version 5. The ADIALOG5.CPP file is the same as the ADIALOG4.CPP file.

The ATextDialog Class Header File

The ADIALOG5.HPP contains the class definition and interface specifications for the ATextDialog class. The ADIALOG5.HPP file is the same as the ADIALOG4.HPP file.

The Earth Window Source File

The AEARTHW5.CPP contains the source code for the Earth window graphic.

The AEarthWindow Class Header File

The AEARTHW5.HPP contains the class definition and interface specifications for the AEarthWindow class.

The Resource File

Version 5 of the Hello World application provides a resource file, AHELLOW5.RC.

The Icon File

AHELLOW5.ICO is used as both the title bar icon and the icon that is displayed when the application is minimized. We do not provide a listing for the AHELLOW3.ICO file. This icon is the same as for Version 2. Refer to Figure 36 on page 158 to see how this icon appears.

The Text Dialog Template

ADIALOG5.DLG contains the template used to build the text dialog. Here is that template:

```
DLGINCLUDE 1 "AHELLOW5.H"

DLGTEMPLATE WND_TEXTDIALOG LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "Hello World Edit Dialog", WND_TEXTDIALOG, 17, 22, 137, 84,
            WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
   BEGIN
        DEFPUSHBUTTON   "OK", DID_OK, 6, 4, 4 , 14
        PUSHBUTTON      "Cancel", DID_CANCEL, 49, 4, 4 , 14
        LTEXT           "Edit Text:", DID_STATIC, 8, 62, 69, 8
        ENTRYFIELD      "", DID_ENTRY, 8, 44, 114, 8, ES_MARGIN
   END
END
```

The Text Dialog Resource File

ADIALOG5.RES is created by the resource compiler as input to HELLO5.EXE.

The Help Window Source File

The AHELLOW5.IPF file contains the text and IPF tags used to produce the help information for the Hello World application. IPF uses a tag language to format the text that appears in a help window. For example, :p. is the paragraph tag, which is used to start a new paragraph.

Refer to the *OS/2 2.0 Information Presentation Facility Guide and Reference* for descriptions of other tags used in the following source file. The IPFC complier, provided by the OS/2 2.0 Developer's Toolkit, is used to compile this file.

The Module Definition File

The AHELLOW5.DEF file is required for the same reasons that AHELLOW1.DEF was needed for Version 1. A module definition file may be created to define certain aspects of the application to the linker.

The only differences between the two .DEF files used in Version 1 and Version 5 are the change in the version number and the stack size.

```
NAME      HELLO5       WINDOWAPI

DESCRIPTION 'Hello World Sample C++ Program - Version 5'

CODE      LOADONCALL  MOVEABLE
DATA      MOVEABLE    MULTIPLE
```

Exploring Version 5

The following sections describe each of the tasks performed by Version 5 of the Hello World application that were not described for previous versions. This version provides several new member functions that the AHelloWindow class uses to construct the main window. They are declared as protected member functions in the AHELLOW5.HPP file, as follows:

```
// <in AHELLOW5.HPP>
⋮
virtual Boolean setupClient();        //Setup Client Window                 v5
virtual Boolean setupHelp();          //Setup Help                           .
virtual Boolean setupInfoArea() ;     //Setup Information Area              .
virtual Boolean setupMenuBar();       //Setup Menu Bar                        .
virtual Boolean setupStatusArea();    //Setup Status Area                   v5
⋮
```

These member functions are implemented in the AHelloWindow window constructor in AHELLOW5.CPP, as follows:

```
// <in AHELLOW5.CPP>
⋮
setupClient();                        //Setup Client Window                 v5
setupStatusArea();                     //Setup Status Area                   .
setupInfoArea();                       //Setup Information Area             v5
⋮
setupMenuBar();                          //Setup Menu Bar                       v5
setupHelp();                          //Setup  Help                          v5
⋮
```

The following sections show you how to implement these member functions.

Constructing the Client Window

Use the setupClient member function to set up the client window for the main window, as shown in the following steps:

1. Create a split canvas control as the client window. The default splits the canvas vertically into a left pane and a right pane. For example:

```
// <in AHELLOW5.CPP>
⋮
//                                              v5
// AHelloWindow :: setupClient()                                                  .
//    Setup  Client                                                              .
//                                              .
Boolean AHelloWindow :: setupClient()     //Setup Client Window                  .
{                                            //                                   .
  clientWindow=new  ISplitCanvas(          //Create  Canvas                     .
      WND_CANVAS, this, this);               //    with Window Id, parent, owner  .
   setClient(clientWindow);                //Set canvas as Client Window        .
⋮
```

2. Use the helloCanvas member function to create another split canvas control in the left pane of the first split canvas control and to split the second canvas horizontally, as shown below:

```
// <in AHELLOW5.CPP>
⋮
helloCanvas=new ISplitCanvas(              //Create Hello Canvas                  .
   WND_HCANVAS, clientWindow,               //   with Window Id, parent           .
  clientWindow);                           //  and  owner                         .
helloCanvas->setOrientation(               //Set the orientation                  .
  ISplitCanvas::horizontalSplit);      //  to  horizontal                    v5
⋮
```

3. Create the static text control, which displays the "Hello, World!!!!!" text string in the top pane of the second split canvas, sets the text in it, and centers the text:

```
// <in AHELLOW5.CPP>
⋮
hello=new IStaticText(WND_HELLO,          //Create Static Text Control
   helloCanvas, helloCanvas);             //   Pass in client as owner & parent    v5
⋮
```

4. Add the Earth window graphic in the bottom pane of the second split canvas using the AEarthWindow class, which is new for Version 5:

```
// <in AHELLOW5.CPP>
⋮
earthWindow   = new AEarthWindow          //Create Earth Graphic Window            v5
 (WND_EARTH, helloCanvas);                //   Set Window ID, client-owner/parentv5
⋮
```

The AEARTHW5.HPP file, shown in "The AEarthWindow Class Header File" on page 192, contains the interface specifications and implements the AEarthWindow class, in the AEARTHW5.CPP file, shown in "The Earth Window Source File" on page 192.

5. Create a list box in the right pane of the client window split canvas and fill it with text strings:

```
// <in AHELLOW5.CPP>
⋮
listBox=new IListBox(WND_LISTBOX,        //Create ListBox                          v5
   clientWindow, clientWindow,            //   Parent/Owner is ClientWindow         .
  IRectangle(),                           //                                        .
  IListBox::defaultStyle() |              //                                        .
   IControl::tabStop |                    //   Set Tab Stop                         .
   IListBox::noAdjustPosition);           //   Allow the Canvas to control size     .
listBox->addAscending("Hello, World!");   //Add "Hello, World!"                     .
listBox->addAscending("Hi, World!");      //Add "Hi, World!"                        .
listBox->addAscending("Howdy, World!");   //Add "Howdy, World!"                     .
listBox->addAscending("Alo, Mundo!");     //Add Portuguese Version                  .
listBox->addAscending("Ola, Mondo!");     //Add Spain                               .
listBox->addAscending("Hallo wereld!");   //Add Dutch                               .
listBox->addAscending("Hallo Welt!");     //Add German                              .
listBox->addAscending("Bonjour le monde!");//Add French                             .
ISelectHandler::handleEventsFor(listBox);//Set self as select event handler   .
⋮
```

6. Allocate 60 percent of the screen for the left pane of the client window split canvas and 40 percent for the right pane:

```
// <in AHELLOW5.CPP>
⋮
clientWindow->setSplitWindowPercentage(//Set the Window Percentage for            .
   helloCanvas, 6 );                        //    the helloCanvas to 6                  .
clientWindow->setSplitWindowPercentage(//Set the Window Percentage for            .
   listBox, 4 );                            //    the listBox to 4                      .
⋮
```

Setting Up the Help Area

Use the setupHelp member function to set up the help area.

1. Create a help window using the HELP_TABLE constant:

```
// <in AHELLOW5.CPP>
⋮
//                                              v5
// AHelloWindow :: setupHelp()                                                      .
//    Setup  Help                                                                   .
//                                              .
Boolean AHelloWindow :: setupHelp()       //Setup Help Area                         .
{                                           //                                      .
   help=new IHelpWindow(HELP_TABLE,         //Create Help Window Object             .
      this);                                //Setup Help info                       .
⋮
```

The HELP_TABLE constant identifies the following help table in the resource file:

```
// <in AHELLOW5.RC>
⋮
HELPTABLE HELP_TABLE                                                       // .
  BEGIN                                                                   //  .
    HELPITEM  WND_MAIN,            SUBTABLE_MAIN,    1                         //  .
    HELPITEM  WND_TEXTDIALOG,   SUBTABLE_DIALOG,  2                             //  .
  END                                                                     //v5
⋮
```

This help table provides help for the main window (WND_MAIN) and also for the text dialog (WND_TEXTDIALOG) that is used to edit the “Hello, World!!!!!” text string (see Chapter 15, “Adding Dialogs and Push Buttons” on page 175 for a description of the text dialog). The window IDs for WND_MAIN and WND_TEXTDIALOG are specified in AHELLOW5.H:

```
// <in AHELLOW5.H>
⋮
#define  WND_MAIN            x1              //Main  Window  ID
⋮
#define WND_TEXTDIALOG      x1 13            //Text Dialog Window ID                v4
⋮
```

In the resource file, the SUBTABLE_MAIN and SUBTABLE_DIALOG constants identify two help subtables, which define other windows, menu items, the entry field in the text dialog, and push buttons for the available help. For example:

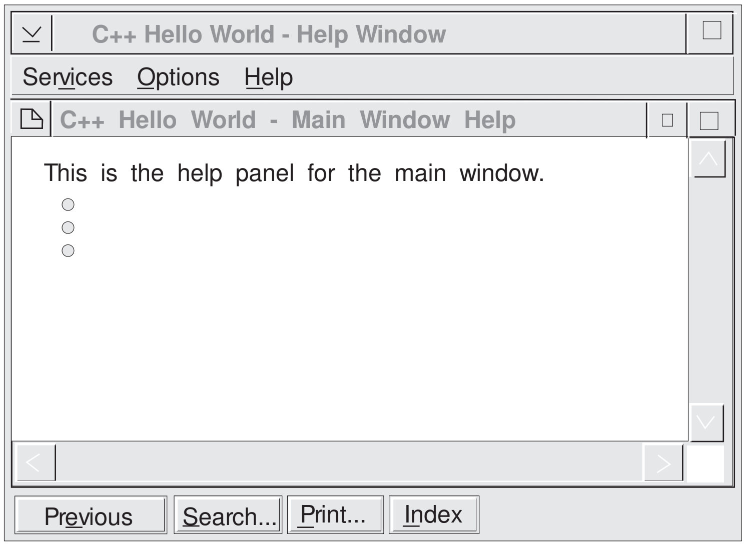
```
// <in AHELLOW5.RC>
⋮
HELPSUBTABLE SUBTABLE_MAIN                     //Main Window Help Subtable      v5
  BEGIN                                  //                                    .
    HELPSUBITEM WND_HELLO, 1                  //Hello <-> Help ID 1            .
    HELPSUBITEM WND_LISTBOX,1 2                //List Box Help
    HELPSUBITEM MI_EDIT, 11                //Edit Menu                           .
    HELPSUBITEM MI_ALIGNMENT, 111              //Alignment Menu                    .
    HELPSUBITEM MI_LEFT, 112                 //Left Menu Item                      .
    HELPSUBITEM MI_CENTER, 113                //Center Menu Item                   .
    HELPSUBITEM MI_RIGHT, 114                //Right Menu Item                     .
    HELPSUBITEM MI_TEXT, 199                 //Text Menu Item                      .
  END                                    //                                    v5

HELPSUBTABLE SUBTABLE_DIALOG                    //Text Dialog Help Subtable    v5
  BEGIN                                  //                                    .
    HELPSUBITEM DID_ENTRY, 2 1               //Entry Field <-> Help ID 2 1     .
    HELPSUBITEM DID_OK, 2 2                 //OK Button <-> Help ID 2 2        .
    HELPSUBITEM DID_CANCEL, 2 3              //OK Button <-> Help ID 2 3       .
  END                                    //                                    v5
```

2. Designate the AHELLOW5.HLP file as the source of the help information:

```
// <in AHELLOW5.CPP>
⋮
help->addLibraries("AHELLOW5.HLP");     //    set self, help table filename           .
⋮
```

Use the IPFC compiler, which is included with the OS/2 2.0 Developer's Toolkit, to compile the AHELLOW5.IPF file and to produce the AHELLOW5.HLP file.

The main window help for the Hello World application looks like this:


C++ Hello World - Help Window
Services Options Help
C++ Hello World - Main Window Help
Previous
Search...
Print...
Index

This is the help panel for the main window.

Figure 45. Main Window Help for Hello World Version 5

The help window displays when the user presses the **F1** key or selects the **Help** choice on the menu bar and then selects **General help...** from the pull-down menu. Use the following code:

```
// <in AHELLOW5.CPP>
:
case MI_GENERAL_HELP:                    //Code to Process Help for help         v5
   help->show(IHelpWindow::general); //Show General Help Panel                  .
   return(true);                         //Return command processed              .
  break;                                 //                                      v5
:
```

3. Set the title of the help window using the following code:

```
// <in AHELLOW5.CPP>
:
help->setTitle(STR_HTITLE);              //Set the Help Window Title              .
:
```

4. Create a handler, AHelpHandler, to customize the keys help:

```
// <in AHELLOW5.CPP>
:
AHelpHandler  phelpHandler=              //Create Custom Help Handler to          .
   new AHelpHandler();                   //   handle the Keys Help                .
:
```

The interface specifications for the AHelpHandler class are declared in the AHELLOW5.HPP header file:

```
// <in AHELLOW5.HPP>
⋮
class AHelpHandler: public IHelpHandler//                                .
{                                      //                                .
   protected:                          //Define Protected Member        .
    virtual  Boolean                   //                                .
        keysHelpId(IEvent& evt);       //Override this function         .
};                                                                   //v5
#endif
```

The keysHelpId member function is implemented in the AHELLOW5.CPP file as follows:

```
// <in AHELLOW5.CPP>
⋮
Boolean AHelpHandler :: keysHelpId(IEvent& evt) //                       .
{                                               //                       .
   evt.setResult(1   );                        //1   =keys help id in   .
                                                //    ahellow5.ipf  file  .
   return true;                                //Return command processed .
} /  end AHelpHandler :: keysHelpId(...)  /                          //v5
```

5. Start the help handler using the following code:

```
// <in AHELLOW5.CPP>
⋮
phelpHandler->handleEventsFor(this);   //Start Help Handler              .
⋮
```

Setting Up the Information Area

The following steps show you how to use the setupInfoArea member function to set up the information area for the main window.

1. Create the information area, using the following code:

```
// <in AHELLOW5.CPP>
⋮
//                                       v5
// AHelloWindow :: setupInfoArea()                                        .
//     Setup Information Area                                             .
//                                       .
Boolean AHelloWindow :: setupInfoArea() //Setup Information Area          .
{                                      //                              v5
   infoArea=new IInfoArea(this);       //Create the information area   v2
⋮
```

2. Set the text in the information area from the resource file:

```
// <in AHELLOW5.CPP>
⋮
infoArea->setInactiveText(STR_INFO);   //Set information area text from RC     v2
⋮
```

3. Use the height of the current font as the height of the information area:

```
// <in AHELLOW5.CPP>
⋮
  setExtensionSize(infoArea,                    //                                             v5
      (int)IFont(infoArea).maxCharHeight());//and specify height                          .
  return  true;                                 //                                              .
} /  end AHelloWindow :: setupInfoArea()  /                                    //v5
⋮
```

Setting Up the Menu Bar

Use the setupMenuBar member function to set up the menu bar for the main window, as follows:

1. Set the main window as the event handler for commands:

```
// <in AHELLOW5.CPP>
⋮
//                                                  v5
// AHelloWindow :: setupMenuBar()                                                                .
//     Setup Menu Bar                                                                            .
//                                                        .
Boolean AHelloWindow :: setupMenuBar()    //Setup Menu Bar                                  .
{                                                  //                                         .
   ICommandHandler::handleEventsFor(this);//Set self as command event handler    v5
⋮
```

2. Create the menu bar, as in the previous versions:

```
// <in AHELLOW5.CPP>
⋮
menuBar=new IMenuBar(WND_MAIN, this); //Create Menu Bar for main window          .
⋮
```

3. The following code places a check on the **Center** choice in the cascading menu that displays when the user selects the **Alignment** choice on the **Edit** menu:

```
// <in AHELLOW5.CPP>
menuBar->checkItem(MI_CENTER);             //Place Check on Center Menu Item         v3
⋮
```

Setting Up the Status Area

Use the setupStatusArea member function to set up the status area for the main window, as shown in the steps below:

1. Create the status area:

```
// <in AHELLOW5.CPP>
⋮
//                                              v5
// AHelloWindow :: setupStatusArea()                                              .
//     Setup Statue Area                                                          .
//                                             .
Boolean AHelloWindow :: setupStatusArea()//Setup Status Area                     .
{                                          //                                    v5
   statusLine=new IStaticText             //Create Status Area using Static Text v3
      (WND_STATUS, this, this);            //    Window ID, Parent, Owner Parameters.
⋮
```

2. Use the STR_CENTER constant to get the “Center Alignment” text string from the resource file and set it in the status area:

```
// <in AHELLOW5.CPP>
⋮
statusLine->setText(STR_CENTER);        //Set Status Text to "Center" from Res .
⋮
```

3. Set the position and height of the status area. The status area is placed above the client area and its height is that of the current font:

```
// <in AHELLOW5.CPP>
⋮
addExtension(statusLine,                   //Add Status Line above the client        .
   IFrameWindow::aboveClient,              //    and specify the height                .
   IFont(statusLine).maxCharHeight()); //    and specify height                    v3
⋮
```

Hello World — Version 5

Chapter 17. Enabling National Language Support and Advanced Functions

Version 6 of the Hello World application shows you how to do the following:

- Use English, German, or Portuguese DLL resources
- Add an **Open...** menu item and use a file dialog
- Show a message box when the input file cannot be read from the file dialog
- Add a pop-up menu for changing the alignment
- Change the status area to a split canvas and add the date and time
- Add a time handler (ATimeHandler) and update the time on the status area
- Add code to delete objects when the application ends
- Add the HELLOWPS.CMD file to create a folder with programs on the OS/2 Workplace Shell

The main window for Version 6 of the Hello World application looks like this:

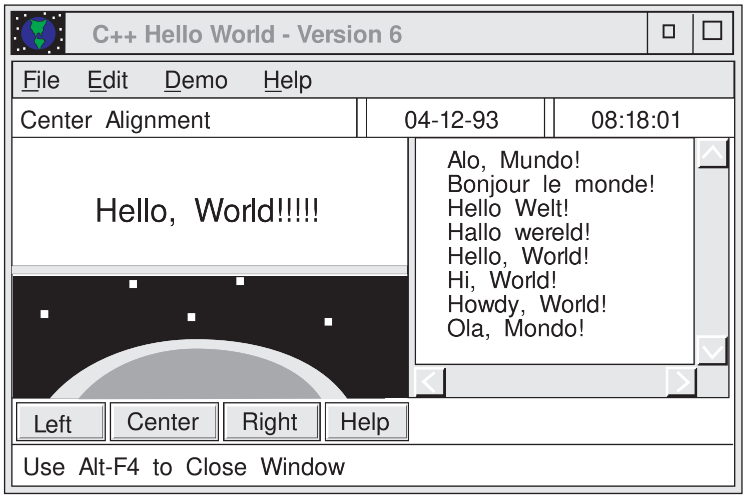


Figure 46. Version 6 of Hello World Application

Establishing the Version 6 Window-Parent Relationships

Figure 47 shows the relationship between the objects built for Version 6 of the Hello World application:

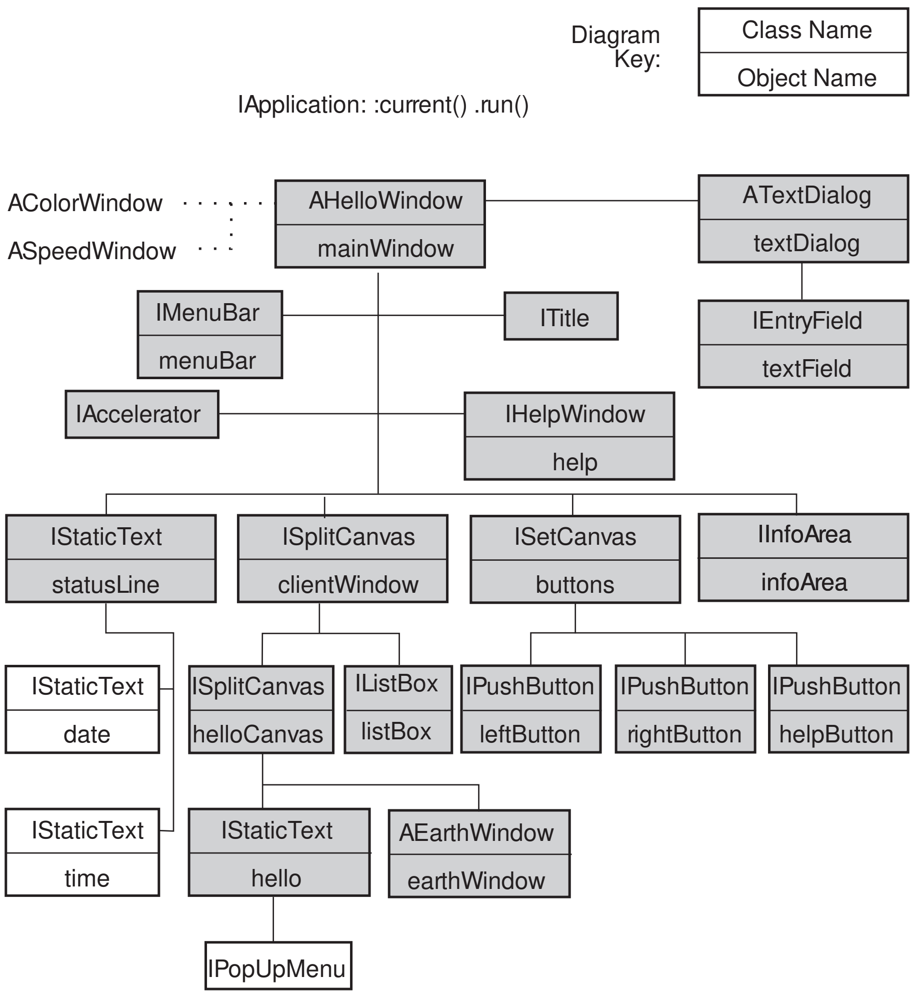


Figure 47. Window-Parent Relationship Diagram, Version 6

As the figure shows, Version 6 of the Hello World application creates the a date object and a time object; both are instances of the IStaticText class, a class that creates and manages the static control window. In addition, Version 6 creates an instance of the IPopUpMenu class to allow you to construct and operate on pop-up menus.

Listing the Version 6 File Names

The following files contain the code used to create Version 6:

File	Type of Code
AHELLOW6.CPP	Source code for main procedure and AHelloWindow class
AHELLOW6.HPP	Header file for the AHellowWindow class
AHELLOW6.H	Constant definitions file for HELLO6.EXE
ADIALOG6.CPP	Source code to create the ATextDialog class
ADIALOG6.HPP	Header file for the ATextDialog class
AEARTHW6.CPP	Source code to create the AEarthWindow class
AEARTHW6.HPP	Header file for the AEarthWindow class
ACOLORW6.CPP	Source code to create the AColorWindow class
ACOLORW6.HPP	Header file for the AColorWindow class
ASPEEDW6.CPP	Source code to create the ASpeedWindow class
ASPEEDW6.HPP	Header file for the ASpeedWindow class
ATIMEHDR.CPP	Source code to create the ATimeHandler class
ATIMEHDR.HPP	Header file for the ATimeHandler class
ADUMMY6.CPP	File to provide dummy file for resource DLLs
AHELLOWE.RC	English resource file for HELLO6.EXE
AHELLOWG.RC	German resource file for HELLO6.EXE
AHELLOWP.RC	Portuguese resource file for HELLO6.EXE
AHELLOW6.ICO	Icon file for HELLO6.EXE
BRAZIL.ICO	Icon file for Portuguese option of HELLO6.EXE
GERMANY.ICO	Icon file for German option of HELLO6.EXE
ADIALOGE.DLG	English dialog resource source file for HELLO6.EXE
ADIALOGG.DLG	German dialog resource source file for HELLO6.EXE
ADIALOGP.DLG	Portuguese dialog resource source file for HELLO6.EXE
ADIALOGE.RES	Dialog resource file for HELLO6.EXE
AHELLOW6.IPF	Help file for HELLO6.EXE
AHELLOW6.DEF	Module definition file for HELLO6.EXE
AHELLOWE.DEF	Module definition file for AHELLOWE.DLL
AHELLOWG.DEF	Module definition file for AHELLOWG.DLL
AHELLOWP.DEF	Module definition file for AHELLOWP.DLL

Hello World — Version 6

Exploring Version 6

The following list describes the tasks performed by Version 6 of the Hello World application that are not already described for previous versions. The tasks are:

- Using English, German or Portuguese DLL resources
  - Updating the main routine in the AHELLOW6.CPP file
  - Creating the AHELLOWE.RC, AHELLOWG.RC, and AHELLOWP.RC resource files
  - Creating the ADIALOGE.DLG, ADIALOGG.DLG, and ADIALOGP.DLG dialog files
  - Creating the BRAZIL.ICO and GERMAN.ICO icon files
- Adding an **Open...** menu item and using a file dialog
  - Updating Menu in resource files
  - Adding the openFile member function to the AHELLOW6.CPP and the AHELLOW6.HPP files
- Showing a message box
  - Adding code in the openFile member function
  - Adding the STR_MSGTXT string resource to resource files
- Adding a pop-up menu for changing the alignment
  - Adding a new menu to the resource files
  - Adding the AMenuHandler class with the makePopUpMenu member function in the AHELLOW6.CPP and AHELLOW6.HPP files
  - Updating the setupClient member function in the AHELLOW6.CPP file to create this handler and attach it to the hello static text window
- Changing the status area to a split canvas and adding the date and time
  - Updating the setupStatusArea member function in the AHELLOW6.CPP file
- Adding a time handler and updating the time in the status area
  - Adding the ATimeHandler class in the new ATIMEHDR.CPP and ATIMEHDR.HPP files
  - Adding ATimeHandler::handleEventsFor(this); in the constructor for AHelloWindow
  - Adding ATimeHandler::stopHandlingEventsFor(this); in the destructor for AHelloWindow
  - Adding the tick member function in the AHELLOW6.CPP and AHELLOW6.HPP files
- Adding code to delete objects when the application ends
- Adding the HELLOWPS.CMD to create a workplace folder for applications

Appendix. Class Hierarchy by Category

The User Interface Class Library contains over 260 classes and over 2600 member functions. To assist you in learning about the classes and to guide you as you develop applications, the classes are divided into categories.

Application Classes

The application classes provide support for the application, threads, profiles, and the resources used by the application.

```
IBase
  ICritSec
  IProcedureAddress
  IReference
  IResourceId
  IVBase
    IApplication
      ICurrentApplication
    IDMImage
    IDMItemProvider
      IDMItemProviderFor
    IProfile
    IRefCounted
      IDMItem
        IDMCnrItem
        IDMEFItem
        IDMMLEItem
      IDMOperation
        IDMSourceOperation
        IDMTargetOperation
      IThreadFn
        IThreadMemberFn
     IDMRenderer
       IDMSourceRenderer
       IDMTargetRenderer
    IResource
      IPrivateResource
      ISharedResource
    IResourceLibrary
      IDynamicLinkLibrary
    IResourceLock
    IThread
      ICurrentThread
```

Data Types and Attributes Classes

The data type classes model basic data types, such as strings, points, and rectangles. These classes hide the structure of the data, while providing the capability to access and alter the data.

```
IBase
  IColor
    IDeviceColor
    IGUIColor
  IDate
  IHandle
    IAccelTblHandle
    IAnchorBlockHandle
    IBitmapHandle
      ISystemBitmapHandle
    IEnumHandle
    IMessageQueueHandle
    IModuleHandle
    IPageHandle
    IPointerHandle
      ISystemPointerHandle
    IPresSpaceHandle
    IProcessId
    IProfileHandle
    ISemaphoreHandle
    IStringHandle
    IThreadId
    IWindowHandle
  IPair
    IPoint
    IRange
    ISize
  IRectangle
  IString
    I0String
  ITime
  IVBase
    IBuffer
      IDBCSBuffer
    IFont
    IStringTest
      IStringTestMemberFn
IStringEnum
```

Error Handling and Exception Classes

The error handling and exception classes inform the application that the library cannot complete a request. Instances of these classes capture the type of exception and other information about the exception.

```
IBase
  IVBase
    IErrorInfo
      IGUIErrorInfo
      ISystemErrorInfo
    ITrace
    IWindow::ExceptionFn
IException
  IAccessError
  IAssertionFailure
  IDeviceError
  IInvalidParameter
  IInvalidRequest
  IResourceExhausted
    IOutOfMemory
    IOutOfSystemResource
    IOutOfWindowResource
IException::TraceFn
IExceptionLocation
IMessageText
```

Event Classes

The event classes encapsulate the user's interaction with application windows. The library creates event objects as a result of some action by the user or by other applications. These event objects contain information about what occurred; they are passed to handler objects for processing.

```
IBase
  IVBase
    IEvent
      ICnrDrawBackgroundEvent
      ICommandEvent
      IControlEvent
        ICnrEvent
          ICnrEditEvent
            ICnrBeginEditEvent
            ICnrEndEditEvent
            ICnrReallocStringEvent
          ICnrEmphasisEvent
          ICnrEnterEvent
          ICnrHelpEvent
          ICnrQueryDeltaEvent
          ICnrScrollEvent
        IDrawItemEvent
          ICnrDrawItemEvent
          IListBoxDrawItemEvent
          IMenuDrawItemEvent
          INotebookDrawItemEvent
        IPageEvent
          IPageHelpEvent
          IPageRemoveEvent
          IPageSelectEvent
      IDDEBeginEvent
      IDDEEndEvent
        IDDEClientEndEvent
      IDDEEvent
        IDDEAcknowledgeEvent
          IDDEClientAcknowledgeEvent
            IDDEAcknowledgePokeEvent
            IDDEAcknowledgeExecuteEvent
          IDDEServerAcknowledgeEvent
        IDDESetAcknowledgeInfoEvent
          IDDEClientHotLinkEvent
          IDDEDataEvent
          IDDEExecuteEvent
          IDDEPokeEvent
          IDDERequestDataEvent
          IDDEServerHotLinkEvent
```

  - IDMEvent
    - IDMSourceBeginEvent
      - IDMCnrInitEvent
    - IDMSourceDiscardEvent
    - IDMSourceEndEvent
    - IDMSourcePrintEvent
    - IDMSourceRenderEvent
      - IDMSourcePrepareEvent
    - IDMTargetDropEvent
      - IDMCnrDropEvent
    - IDMTargetEndEvent
    - IDMTargetEnterEvent
      - IDMCnrOverAfterEvent
    - IDMTargetHelpEvent
    - IDMTargetLeaveEvent
  - IFileDialogEvent
  - IFrameEvent
    - IFrameFormatEvent
  - IHelpErrorEvent
  - IHelpHyperTextEvent
  - IHelpMenuBarEvent
  - IHelpNotifyEvent
  - IHelpSubItemNotFoundEvent
  - IHelpTutorialEvent
  - IKeyboardEvent
  - IMenuEvent
  - IMouseClickEvent
  - IPaintEvent
  - IResizeEvent
  - IScrollEvent
- IEventData
  - IEventParameter1
  - IEventParameter2
  - IEventResult

Handler Classes

Handler classes are provided to access window or application-specific handlers. Each window has some default event processing; however, the application can create instances of the handler classes to process certain event objects to override the default behavior.

```
IBase
  IVBase
    IHandler
      ICnrDrawHandler
      ICnrEditHandler
      ICnrHandler
      ICommandHandler
      IDDEClientConversation
      IDDETopicServer
      IDMHandler
        IDMSourceHandler
          IDMCnrSourceHandler
        IDMTargetHandler
          IDMCnrTargetHandler
      IEditHandler
      IFileDialogHandler
      IFocusHandler
      IFontDialogHandler
      IFrameHandler
      IHelpHandler
      IKeyboardHandler
      IListBoxDrawItemHandler
      IMenuDrawItemHandler
      IMenuHandler
        ICnrMenuHandler
        IInfoArea
      IMouseClickHandler
      IPageHandler
      IPaintHandler
      IResizeHandler
      IScrollHandler
      ISelectHandler
      IShowListHandler
      ISliderDrawHandler
      ISpinHandler
```

Settings and Styles Classes

The settings and style classes change the appearance or behavior of window classes.

```
IBase
  IBitFlag
    I3StateCheckBox::Style
    IBitmapControl::Style
    IButton::Style
    ICanvas::Style
    ICheckBox::Style
    IComboBox::Style
    IContainerControl::Attribute
    IContainerControl::Style
    IControl::Style
    IEntryField::Style
    IFileDialog::Style
    IFontDialog::Style
    IFrameWindow::Style
    IGroupBox::Style
    IIconControl::Style
    IListBox::Style
    IListBoxDrawItemHandler::DrawFlag
    IMenu::Style
    IMenuBar::Style
    IMenuDrawItemHandler::DrawFlag
    IMenuItem::Attribute
    IMenuItem::Style
    IMessageBox::Style
    IMultiLineEdit::Style
    INotebook::PageSettings::Attribute
    INotebook::Style
    IOutlineBox::Style
    IProgressIndicator::Style
    IPushButton::Style
    IRadioButton::Style
    IScrollBar::Style
    ISetCanvas::Style
    ISlider::Style
    ISpinButton::Style
    ISplitCanvas::Style
    IStaticText::Style
    IViewPort::Style
    IWindow::Style
  IFileDialog::Settings
  IFontDialog::Settings
  IHelpWindow::Settings
  IVBase
    INotebook::PageSettings
```

Support Classes

The support classes work with other classes. This category includes nested classes.

```
IBase
  IAccelerator
  IDDEActiveServer
  IFrameExtension
  IMenu::Cursor
  IMenuItem
  ISubMenu::Cursor
  ISWP
  ISWPArray
  IVBase
    IComboBox::Cursor
    IContainerColumn
    IContainerControl::ColumnCursor
    IContainerControl::CompareFn
    IContainerControl::FilterFn
    IContainerControl::Iterator
    IContainerControl::ObjectCursor
    IContainerControl::TextCursor
    IContainerObject
    IListBox::Cursor
    INotebook::Cursor
    IProfile::Cursor
    ISpinButton::Cursor
    IWindow::ChildCursor
ISequence<>
  IFrameExtensions
ISet<>
  IDDEActiveServerSet
  IDDEClientHotLinkSet
```

Window Classes

The window classes encapsulate the basic graphical building blocks that are used to construct application windows. These range from the simple graphical objects like title bars, which display the title of the window, to complex objects like containers, which can contain other objects and provide different views on those objects. Window classes support both parent and owner windows. This allows window position and appearance (parent windows) to be separated from event handling (owner windows).

```
IBase
  IMessageBox
  IVBase
    IWindow
      IControl
        ICanvas
          IMultiCellCanvas
          ISetCanvas
          ISplitCanvas
          IViewPort
        IContainerControl
        IListBox
        INotebook
        IOutlineBox
        IProgressIndicator
          ISlider
        IScrollBar
        ISpinButton
        ITextControl
          IButton
            IPushButton
            ISettingButton
              I3StateCheckBox
              ICheckBox
              IRadioButton
          IEntryField
            IComboBox
          IGroupBox
          IMultiLineEdit
          IStaticText
            IBitmapControl
              IIconControl
            IInfoArea
          ITitle
      IFrameWindow
        IFileDialog
        IFontDialog
      IHelpWindow
      IMenu
        IMenuBar
        IPopUpMenu
        ISubMenu
        ISystemMenu
      IObjectWindow
```

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, SC20-1699.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot have a direct object of an abstract class. See also *base class*.

abstraction (data). See *data abstraction*.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access declaration. A declaration used to restore access to members of a base class.

access function. A function that returns information about the elements of a data object so that you can analyze various elements of a string.

access resolution. The process by which the accessibility of a particular class member is determined.

access specifier. One of the C++ keywords public, private, or protected.

argument. In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Synonymous with parameter.

B

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

C

catch block. A block associated with a try block that receives control when a C++ exception matching its argument is thrown.

class. An aggregate that can contain functions, types, and user-defined operators in addition to data. Classes can be defined hierarchically, allowing one class to be an expansion of another, and can restrict access to its members.

class hierarchy. A tree-like structure showing relationships among object classes. It places one abstract class at the top (a base class) and one or more layers of less abstract classes (derived classes) below it.

class lattice. A structure that has an object class inheriting from multiple object classes. See also *multiple inheritance.*

class library. A collection of classes.

const. An attribute of a data object that declares the object cannot be changed.

constructor. A special class member function that has the same name as the class and is used to construct and possibly initialize class objects.

critical section. Code that must be executed by one thread while all other threads in the process are suspended.

D

data abstraction. A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

DBCS. See *double-byte character set.*

deck. A line of child windows in a set canvas that is direction-independent. A horizontal deck is equivalent to a row and a vertical deck is equivalent to a column.

declaration. A description that makes an external object or function available to a function or a block.

declare. To identify the variable symbols to be used at preassembly time.

DDE. Dynamic data exchange.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default constructor. A constructor that takes no arguments, or a constructor for which all the arguments have default values.

delete. (1) A C++ keyword that identifies a free-storage deallocation operator. (2) A C++ operator used to destroy objects created by new.

derivation. The creation of a new or derived class from an existing or base class.

derived class. A class that inherits from a base class. You can add new data members and member functions to the derived class. You can manipulate a derived class object as if it were a base class object. The derived class can override virtual functions of the base class.

Synonym for *subclass*.

destructor. A special member function that has the same name as its class, preceded by a tilde (˜), and that "cleans up" after an object of that class, for example, by freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

detent. A point on a slider that represents an exact value to which a user can move the slider arm.

direct manipulation. A user interface technique whereby the user initiates application functions by manipulating the objects, represented by icons, on the Presentation Manager (PM) or Workplace Shell desktop. The user typically initiates an action by:

1. Selecting an icon
2. Pressing and holding down a mouse button while “dragging” the icon over another object’s icon on the desktop
3. Releasing the mouse button to “drop” the icon over the target object.

Thus, this technique is also known as “drag and drop” manipulation.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, you need hardware and supporting software that are DBCS-capable to enter, display, and print DBCS characters.

drag and drop. See *direct manipulation*.

dynamic data exchange (DDE). A protocol that uses messages to communicate between applications sharing data and that uses shared memory as the means of exchanging data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in which the applications send updates to one another as new data becomes available.

E

enumeration constant. An identifier that is defined in an enumeration and that has an associated integer value. You can use an enumeration constant anywhere an integer constant is allowed.

enumeration data type. A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

exception. (1) Under the OS/2 operating system, a user or system error detected by the system and passed to an OS/2 or user exception handler. (2) For C++, any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called “throwing the exception”).

exception handler. (1) Under the OS/2 operating system, a function that receives the OS/2 exception and either corrects the problem and returns execution to the program, or terminates the program. (2) In C++, a catch block that catches a C++ exception when it is thrown from a function in a try block.

exception handling. A type of error handling that allows control and information to be passed to an exception handler when an exception occurs. Under the OS/2 operating system, exceptions are generated by the system and handled by user code. In C++, try, catch, and throw expressions are the constructs used to implement C++ exception handling.

F

frame extension. A control you can add if it is not available it is not available in the basic Presentation Manager frame windows.

friend class. A class in which all the member functions are granted access to the private and protected members of another class. It is named in the declaration of the other class with the prefix friend.

friend function. A function that is granted access to the private and protected parts of a class. It is named in the declaration of the other class with the prefix friend.

G

global name space. The first position in class names. IBM C Set ++ Class Libraries uses "I," for "IBM."

H

handler. A routine that controls an application's reaction to specific external events, for example, an interrupt handler.

header file. A file that contains system-defined control information that precedes user data.

I

inheritance. (1) A mechanism by which an object class (derived class) can use the attributes, relationships, and methods defined in more abstract classes related to it (its base classes). See also *multiple inheritance.* (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

instance. Synonym for object, a particular instantiation of a data type.

instantiate. To create or generate a particular instance or object of a data type.

item. A "proxy" for the object being manipulated.

L

library. (1) A collection of functions, function calls, subroutines, or other data. (2) A set of object modules that can be specified in a link command.

linkage editor. Synonym for linker.

linker. A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single executable program.

M

member. (1) A data object in a structure or a union. (2) In C++, classes and structures can also contain functions and types as members.

member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class.

message. A request from one object that the receiving object implement a method. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the method to be implemented, and any parameters the method needs for implementation.

method. Synonym for member function.

multiple inheritance. (1) An object-oriented programming technique implemented in C++ through derivation, in which the derived class inherits members from more than one base class. (2) The structuring of inheritance relationships among classes so a derived class can use the attributes, relationships, and methods used by more than one base class.

See also *inheritance* and *class lattice.*

multithread. Pertaining to concurrent operation of more than one path of execution within a computer.

N

nested class. A class defined within the scope of another class.

new. (1) A C++ keyword identifying a free storage allocation operator. (2) A C++ operator used to create class objects.

NULL. A pointer guaranteed not to point to a data object.

null character (\0). The ASCII or EBCDIC character with the hex value 00 (all bits turned off).

O

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods (procedures) that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of (attributes) a data structure and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, though it has unique values assigned to its attributes.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type. See *overloading*.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

pad. To fill unused positions in a field with data, usually 0's, 1's, or blanks.

parameter. See *argument.*

pointer. A variable that holds the address of a data object or function.

private. Pertaining to a class member that is only accessible only to member functions and friends of that class.

process. A program running under OS/2, along with the resources associated with it (memory, threads, file system resources, and so on).

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

public. Pertaining to a class member that is accessible to all functions.

R

RMFs. Rendering mechanisms and formats.

rendering. The transfer or re-creation of the dragged object from the source window to the target window.

renderer. An object that renders data using a particular mechanism, such as using files or shared memory. It contains definitions of supported rendering mechanisms and formats and types. Renderers are maintained positionally (1-based).

resource file. A file that contains data used by an application, such as text strings and icons.

S

scope. That part of a source program in which an object is defined and recognized.

scope operator (::). An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Also called a scope resolution operator.

structure. A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

subclass. See *derived class.*

superclass. See *base class* and *abstract class.*

T

template. A family of classes or functions with variable types.

this. A C++ keyword that identifies a special type of pointer in a member function, one that references the class object with which the member function was invoked.

thread. A unit of execution within a process.

U

User Interface Class Library. A set of C++ classes that simplifies the construction of AIX, OS/2, and Windows/NT applications with graphical user interfaces.

V

virtual function. A function of a class that is declared with the keyword virtual. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called. This is determined at run time.

W

white space. Space characters, tab characters, form feed characters, and new-line characters.

Special Characters

(::) (double colon). Scope operator. An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Also called a scope resolution operator.

Index

Communicating Your Comments to IBM

IBM C/C++ Tools:
User Interface Class Library
User's Guide
Version 2.01

Publication No. S82G-3743-00

If there is something you like—or dislike—about this document, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, and topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

If you prefer to send comments by mail, use the RCF in this document.

If you prefer to send comments by FAX, use this number:

– United States and Canada: 416-448-6161
– Other countries: (+1)-416-448-6161

If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.

– Internet: torrcf@vnet.ibm.com
– IBMLink*: toribm(torrcf)
– IBM/PROFS*: torolab4(torrcf)
– IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

**IBM C/C++ Tools:
User Interface Class Library
User's Guide
Version 2.01**

Publication No. S82G-3743-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction					

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate					
Complete					
Easy to find					
Easy to understand					
Well organized					
Applicable to your tasks					

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? Yes No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Readers' Comments — We'd Like to Hear from You
S82G-3743-00

IBM®

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape **Please do not staple** Fold and Tape

S82G-3743-00

Cut or Fold
Along Line

IBM®

Part Number: 82G3743

Printed in U.S.A.

82G3743

S82G-3743-