UNIX Code Migration Guide

UNIX Application Migration Guide

Chapter 11: Migrating the User Interface

Larry Twork, Larry Mead, Bill Howison, JD Hicks, Lew Brodnax, Jim McMicking, Raju Sakthivel, David Holder, Jon Collins, Bill Loeffler
Microsoft Corporation

October 2002

Applies to:
Microsoft® Windows®
UNIX applications

The *patterns & practices* team has decided to archive this content to allow us to streamline our latest content offerings on our main site and keep it focused on the newest, most relevant content. However, we will continue to make this content available because it is still of interest to some of our users.
We offer this content as-is, without warranty that it is still technically accurate as some of the material is undoubtedly outdated. Note that the content may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.

Summary: Chapter 11: Migrating the User Interface introduces the concepts of both the Windows and X11 windowing systems and discusses the similarities and differences between them. Migration guidance is presented by illustrating how window objects may be coded on UNIX and how that code may be migrated to Windows. (99 printed pages)

Contents

User Interface Coding Examples

Introduction

This chapter describes how to migrate from a UNIX-based user interface to a Windows user interface. Because the overwhelming majority of UNIX graphical interfaces are built on X Windows and Motif, this chapter focuses on porting code from X Windows to the Microsoft® Windows® operating system. It describes:

- The architectural and visual differences between the two environments
- The programming principles used by X Windows and Microsoft Windows
- How to migrate each type of graphical construct from one environment to the other

This chapter also includes short sections on migrating from other UNIX user interface types, including text-based and OpenGL-based interfaces. It concludes with examples of how user interface code can be migrated in the sections under "UI Coding Examples."

Comparing X Windows and Microsoft Windows

The main user interface type in use on the UNIX platform today builds on the X Windows set of standards, protocols and libraries. To gain an understanding of how to migrate such a user interface, it is worth comparing the user interface architecture and the resulting look and feel in the two models. Finally, it is useful to understand differences in windowing terminology between the two environments.

User Interface Architecture

The architecture of X Windows-based interfaces differs significantly from Microsoft Windows architecture. First and most fundamental is the orientation of client and server. For X Windows, the client is the application that requests services and receives information from the user interface. The user-facing elements of the interface are based on what is termed the X Server.

In the X Windows-based system, the client application sends requests to the server to display graphics and to send mouse and keyboard events. The X Server is responsible for doing all the work on the client's behalf. The client might run on a remote system with no graphics hardware or on the same physical machine as the server. In either case, the client does not interact with the display, mouse, or keyboard. This is shown in Figure 1, which represents the X Windows client-server architecture.

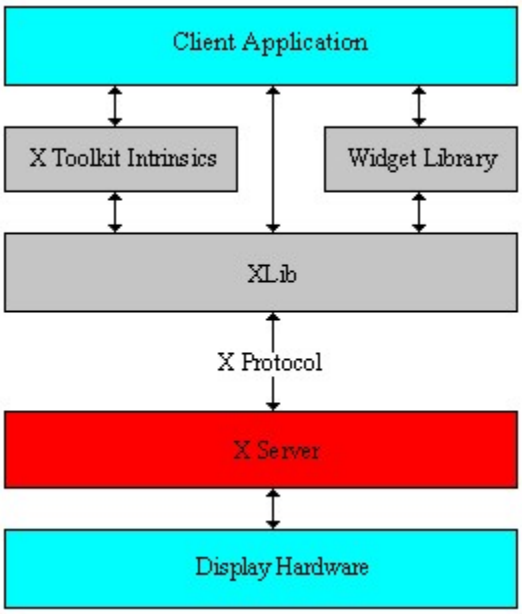

Figure 1. The X Windows architectural model

By contrast, a standard Microsoft Win32® application is not responsible for dealing with the display, mouse or hardware. Figure 2 shows the path from an application down through the layers to the hardware in Win32.

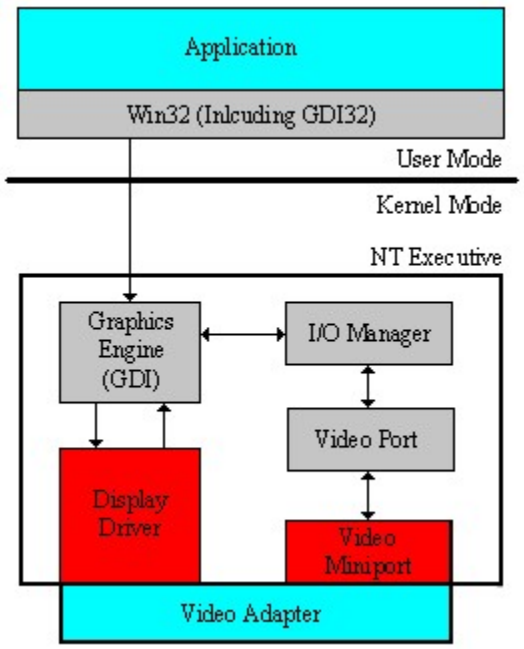

Figure 2. The Microsoft Windows user interface architecture

Look and Feel

X Windows is normally used with the Motif widget library, which is a library of user interface components—such as scroll bars, buttons, drop-down lists and dialog boxes—that can be used off the shelf. Because many X Windows applications use Motif's "look and feel," Motif has transcended being

considered just a third-party library, to the extent that some developers consider X Windows and Motif as one and the same.

According to the Motif Programming Manual and the Microsoft Official Guidelines for User Interface Developers and Designers, all applications that a user can run on the desktop should have a consistent "look and feel" as well as functional design. Anything else is likely to confuse users, possibly to the point where they will not use the application.

Although there are many differences, Microsoft Windows and X Windows with Motif both have roots in the IBM Common User Access (CUA) guidelines. The resulting similarity in look and feel is not too surprising. Every windowing system needs to perform the same tasks:

- Determine which font is used to display text
- Determine background color
- Specify where a checkbox appears
- Show that a user has clicked a particular button

After satisfying the task list, it just becomes a matter of methodology. Notice the similarities in terminology and appearance of in Figure 3, An example Motif dialog box, and Figure 4, An example Microsoft Windows dialog box.

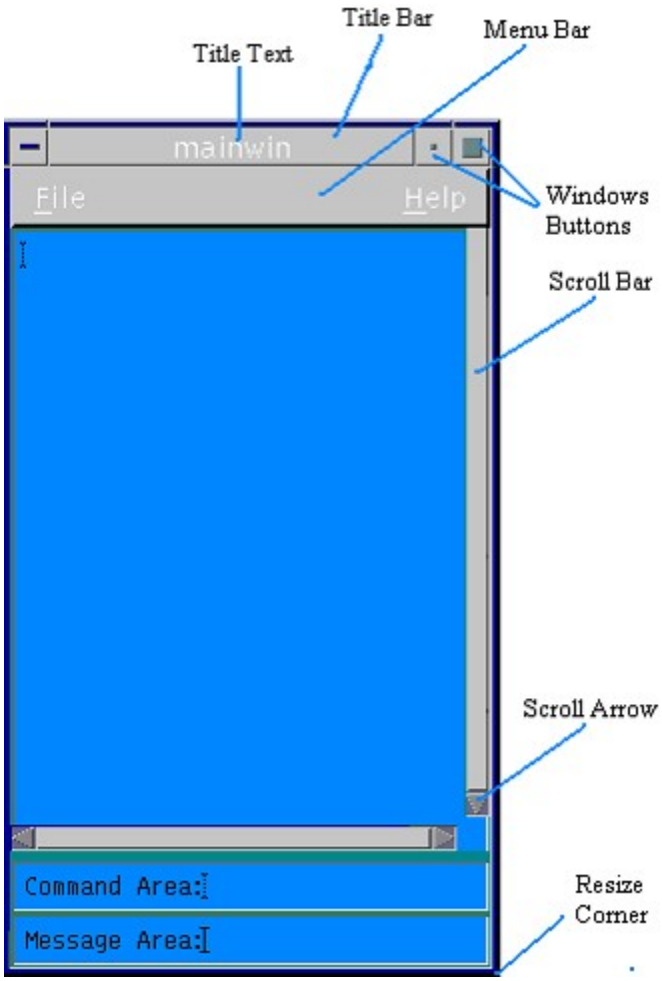

Figure 3. An example Motif dialog box

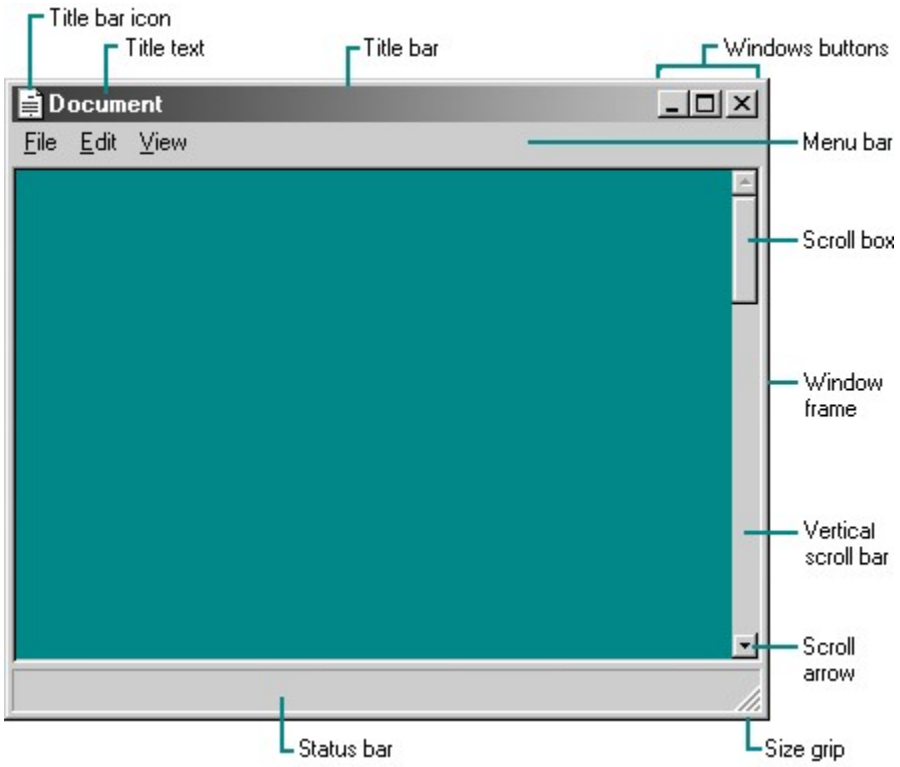

Figure 4. An example Microsoft Windows dialog box

> **Note** To ensure a consistent look and feel with other Windows applications, migration of X/Motif applications to native Microsoft Win32 should be governed by the official Microsoft guidelines.

Window Types

Windows types are very similar between the X Windows and Windows environments, as detailed here.

Desktop window

The X Windows system automatically creates the desktop window. This is a system-defined window that is the base for all windows displayed by all applications. In X Windows, it can be thought of in the same general terms as the root window.

A Win32-based application can retrieve a handle to this window by using the **GetDesktopWindow()** function.

Application window

The Application Window is the interface between the user and the application. Elements such as a menu bar, window menu, minimize, maximize, close button, title bar, sizing border, client area, and scroll bars typically appear in the Application Window.

Dialog boxes

A user typically accesses a dialog box as a temporary window used to create some additional input. A

dialog box contains one or more controls, such as buttons and check boxes, to elicit user input. A developer can build an entire Win32-based program by using dialog box functionality.

Modeless dialog box

When the system creates a modeless dialog box, it becomes the active window. The modeless dialog box does not disable its parent window nor send messages to its parent window. However, it stays at the top of the Z-order even if its parent window becomes the active window.

Applications can create a modeless dialog box by using the **CreateDialog()** function, with arguments to specify the identifier of a dialog box template and the pointer to the callback procedure that handles messages for the window.

Modal dialog box

A modal dialog box becomes the active window when the system creates it. Until a call to **EndDialog()**, the dialog box remains the active window. Neither the application nor the user can make the parent window active. **EndDialog()** must be called.

An application uses the **DialogBox()** function with a resource identifier to create a modal dialog box. Use a modal dialog box when it is desirable to force user input before proceeding.

Message box

A message box is a special dialog box that displays a note, caution or warning to the user. For example, a message box can inform the user of a problem the application has encountered while performing a task.

Reference Material

Table 1 compares popular subject matter for X/Motif and Microsoft Windows. All of the Microsoft documents are accessible from the MSDN Web® site.

Table 1. References for X/Motif and Microsoft Windows

X Windows Reference	Microsoft Windows Reference
Motif Style Guide	Official Guidelines for User Interface Developers and Designers
Motif Programming Manual	Platform SDK: Windows User Interface
Motif Reference Manual	Platform SDK: Windows API

To download the Microsoft Platform SDK, go to the Microsoft Web page for downloads. From the list of available downloads, type "Platform SDK" in the Product Name box, and then follow the instructions to download it.

User Interface Programming In X Windows and Microsoft Windows

The basics of getting a Microsoft Windows-based application and an X/Motif application started are very similar conceptually. Also, the libraries and core functions are similar in both environments. This section

discusses the programming principles for developing user interfaces in the two environments.

Programming Principles

The basic structure of an X Windows application that uses Motif is quite similar to the structure of a Microsoft Windows-based application.

To initiate an X Windows-based interface

1. Initialize the toolkit.
2. Create widgets.
3. Manage widgets.
4. Set up callbacks.
5. Display widgets.
6. Enter the main program event handler.

The following code illustrates these steps:

```
topWidget = XtVaAppInitialize();
frame = XtVaCreateManagedwidget("frame",xmFrameWidgetFrams,
topWidget,,,);
button = XmCreatePushButton( frame, "EXIT", NULL, 0 );
XtManageChild(button)
XtAddCallback( button, XmNactivateCallback, myCallback, NULL );
XtRealizeWidget( topWidget );
XtAppMainLoop();
```

To initiate a Microsoft Windows-based application

1. Initialize an instance and register the Window class.
2. Set up callbacks.
3. Create the window.
4. Display the window.
5. Enter the main program message loop.

The following code illustrates these steps:

```
windowClassStruct.hInstance = thisApplicationInstance;
windowClassStruct.lpfnWndProc = (WNDPROC)myCallback;
RegisterClass( &windowClassStruct );
myWindow = CreateWindow();
ShowWindow(myWindow);
while( GetMessage(,,,) ) {…}
```

To deal with input devices, X clients and Win32-based applications rely on events and messages from the outside. X and Win32 use a similar method for this, a message loop where a callback or inline code executes based on the nature of the event or message.

The following code snippet shows a simple Win32 message loop:

```
while ( GetMessage( &msg, NULL, 0, 0 ) ) {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
```

```
}
```

The **GetMessage()** API returns an MSG structure (**&msg**). Right now, only the **message** member of this structure (**UINT message;** in the following listing) is of interest. Windows places the message identifier in this field. The developer can use this in the message loop to capture device events.

```
Typedef struct tagMSG {
    HWND  hwnd;
    UINT  message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD  time;
    POINT  py;
} MSG, *PMSG;
```

Libraries and Include Files

Despite differences in their underlying architectures, many of the graphical functions used in X Windows and Microsoft Windows perform similar tasks. These include the core libraries and the Motif and Win32 common dialog boxes.

Core libraries

A number of functions exist to support the core API used in a graphical user interface. X Windows includes the libraries X and Xlib and the X Windows Intrinsics toolkit. The Win32 equivalent is Windows.h, which includes a great number of additional header files.

The Microsoft compiler includes Win32 USER32.LIB and GDI32.LIB import libraries, which can be roughly compared to X Library and X Toolkit Intrinsics because they provide nearly all of the basic window management and 2D Graphics APIs.

These Win32 libraries are called *import* libraries because they provide information to the linker. When a Win32-based program references the **CreateWindow()** function, User32.lib tells the linker that this function is in the User32.dll dynamic-link library. This information goes into the .exe file, which enables Windows to perform dynamic linking by using the User32.dll and Gdi32.dll dynamic-link libraries when the program is executed.

Motif and Win32 common dialog boxes

Dialog box functionality is provided by Motif and by the Windows common dialog box library. If the code migrates from Motif, there is probably an equivalent Win32 common dialog box for each Motif function.

Win32 provides a set of functions to create commonly used windows. If Commdlg.h is included in a project, the project has access to the Win32 common dialog box functions. The Comdlg32.dll library stores templates for these dialog boxes, along with the code to drive them. By using these, a developer can save time and provide a consistent look and feel in the application being migrated.

For example, the Motif function **XmCreateFileSelectionDialog()** is very similar to the Win32 function **GetOpenFileName()**. The X/Motif code must include the Xm/FileSB.h header file. The Win32-based applicationmust include Commdlg.h and link to Comdlg32.lib. Calling the **GetOpenFileName()** API to displays the Open File dialog box, as shown in Figure 5.

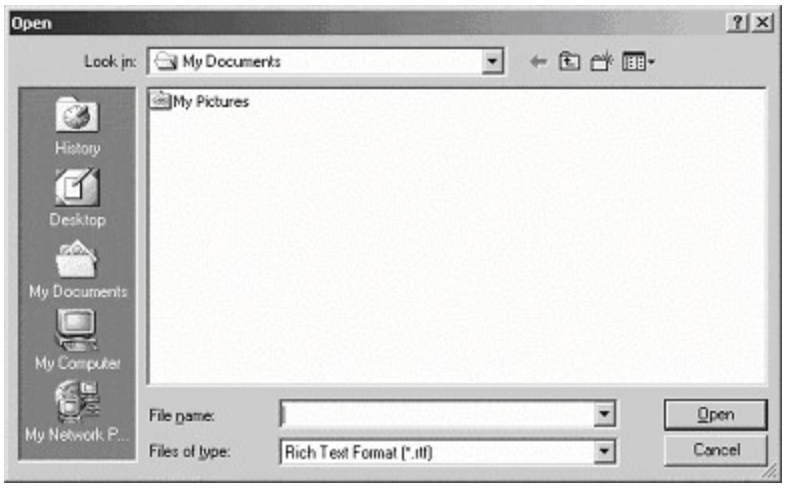

Figure 5. The GetOpenFileName common dialog box

Table 2 lists the available common dialog box functions.

Table 2. Common dialog box functions

Function	Description
ChooseColor()	Creates a Color dialog box that enables the user to select a color.
ChooseFont()	Creates a Font dialog box that enables the user to choose attributes for a logical font.
CommDlgExtendedError()	Returns a common dialog box error code.
FindText()	Creates a system-defined modeless Find dialog box that lets the user specify a string to search for and specify options to use when searching for text in a document.
GetFileTitle()	Retrieves the name of the specified file.
GetOpenFileName()	Creates an Open dialog box that lets the user specify the drive, the directory, and the name of a file or set of files to open.
GetSaveFileName()	Creates a Save dialog box that lets the user specify the drive, the directory, and the name of a file to save.
PageSetupDlg()	Creates a Page Setup dialog box that enables the user to specify the attributes of a printed page.
PrintDlg()	Displays a Print dialog box.
PrintDlgEx()	Displays a Print property sheet that enables the user to specify the properties of a particular print job.
ReplaceText()	Creates a system-defined modeless dialog box that lets the user specify a string to search for and a replacement string, as well as options to control the find and replace operations.

In addition to Windows.h, the Windowsx.h library has roots in Windows version 3.1 and provides many useful macros that can be used in code migration. These macros are not used as often or in the same ways now, but they can be helpful. For example, the **SelectPen()** and **DeletePen()** macros can be more intuitive

than calling **SelectObject()** and **DeleteObject()** with all the required type specifications.

```
#define SelectPen(hdc, hpen)((HPEN)SelectObject((hdc),
(HGDIOBJ)(HPEN)(hpen)))
#define DeletePen(hpen) DeleteObject((HGDIOBJ)(HPEN)(hpen))
```

Writing X Windows APIs for Win32

It might be easier to migrate by creating X Windows-like APIs in the Win32 environment. Creating these APIs can make the original X Windows code easier to migrate to native Win32 and perhaps make the code more portable. The following examples demonstrate this approach.

The following are examples of macro-based APIs:

```
#define DefaultRootWindow(display) GetDesktopWindow()
#define RootWindow(display,screen) GetDesktopWindow()
#define DefaultScreen( display )   ((int)(0))
#define DefaultGC(display,screen)  GetDC(display)
```

The following code demonstrates the use of **DisplayWidth** and **DisplayHeight**:

```
#define DisplayWidth(d,s)  ((unsigned int)GetSystemMetrics(
SM_CXSCREEN ))
#define DisplayHeight(d,s) ((unsigned int)GetSystemMetrics(
SM_CYSCREEN ))
unsigned int maxWIDE;
unsigned int maxHIGH;
maxWIDE = DisplayWidth(x,x);
maxHIGH = DisplayHeight(x,x);
```

The following code demonstrates the use of **XtMessage**:

```
//+
//  XtMessage( TCHAR *message )
//
//  This function requires the following on Win32:
//    1. A global variable HANDLE stdOut;
//    2. The following initialization code in WinProc()
//
//         switch(message) {
//         case WM_CREATE :
//    --<     AllocConsole();
//    --<     stdOut = GetStdHandle( STD_OUTPUT_HANDLE );
//           (. . .)
//
//
//    3. The following cleanup code in WinProc()
//
//         switch(message) {
//         case WM_CLOSE :
//    -->     FreeConsole();
//        (. . .)
//-
```

The following code demonstrates the use of **XtWarning**:

```
void XtWarning( TCHAR *message )
{
  DWORD charsWritten;
  WriteConsole( stdOut, message, _tcsclen(message) , &charsWritten,
NULL );
}
```

Window Management

Window management functions cover the creation, initialization, management, and eventual destruction of dialog boxes and other window types.

Creating Windows

The code listings in this section show some X Windows and Win32 implementations of window management. It is not likely that any large-scale X Windows client or Win32-based application would actually be implemented as these short code snippets are. However, it is easy to see the conceptual similarities and some differences as well.

An X Windows X11 client might use **XtAppInitialize()**, **XtVaAppInitialize()**, **XtOpenApplication()** or **XtVaOpenApplication** to get a top-level widget to create a window, as shown in the following code:

```
main (int argc, char *argv[] )
{
  Widget toplevel;     /* Conceptual Application Window */
  XtAppContext  app;   /* context of the app */

  toplevel = XtVaAppInitialize( &app,
                                "myClassName",
                                NULL,0,&argc,argv,NULL,NULL );

           OR

  toplevel = XtOpenApplication( &app,
                                "myClassName",
                                NULL,0,&argc,argv,NULL,
                                whateverWidgetClass, NULL,0);

  ...
  ...
```

In the following code, a Win32-based graphical application creates a main window:

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;  // handle to the Application Window

   hWnd = CreateWindow( "myClassName",
                        "myWindowsName",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT,
                         0,
                         CW_USEDEFAULT,
                         0,
                         NULL,
                         NULL,
```

```
                          hInstance,   // context of the app
                          NULL);
  …
  …
```

An X Windows client can create a control or widget as follows:

```
main (int argc, char *argv[] )
{
  Widget toplevel;     /* Conceptual Application Window */
  Widget button;
  XtAppContext  app;   /* context of the app */

  toplevel = XtVaAppInitialize( &app, "Example",
NULL,0,&argc,argv,NULL,NULL );
  button   = XtVaCreateManagedWidget( "command",
                                      commandWidgetClass,  /* class
*/
                                      toplevel,           /* parent
*/
                                      XtNheight, 50,
                                      XtNwidth, 100,
                                      XtNlabel, "Press To Exit",
                                      NULL );
```

A Win32-based application can create a control or widget as follows:

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;  // handle to the Application Window
   HWND hButton;

   hWnd = CreateWindow("myClassName",
                       "Example",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        0,
                        CW_USEDEFAULT,
                        0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);


   hButton = CreateWindow( "BUTTON",        // class
                           "Press To Exit",
                           WS_CHILD | BS_PUSHBUTTON,
                           Xcoordinate,
                           Ycoordinate,
                           Width,
                           Height,
                           hWnd,           // parent
                           (HMENU)idNumberOfThisControl,
                           hInstance,
                           NULL
                           );
```

Creating Controls

Controls—such as X Windows widgets—come in all shapes, sizes, colors and functions. There are two ways to create controls in a Win32 environment. The first and simplest method is by using the resource editor in Microsoft Visual Studio® or one of the many dialog box editors. Use these tools to drag and drop controls onto a window or dialog box, which in X Windows is a widget itself. In Win32, controls are also windows in every respect.

Using a dialog editor produces a resource file with the extension .rc, for example MyProgram.rc.

The following code is a small part of a resource file, and shows the definition of a dialog box with several controls:

```
//////////////////////////////////////////////////////////////////
//
// Dialog
//
EXAMPLE_DIALOG DIALOG DISCARDABLE  0, 0, 267, 161
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "  Mission Setup "
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON      "&Quit",IDC_EXIT,210,135,50,15
    GROUPBOX        "  Select Mission ",IDC_STATIC,5,5,100,55
    CONTROL         "Destroy Planet",IDC_DESTROY_PLANET,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP,10,20,62,10
    CONTROL         "Destroy Star",IDC_DESTROY_STAR,"Button",
                    BS_AUTORADIOBUTTON,10,35,55,10
    EDITTEXT
IDC_AUTHORIZATION_CODE,190,85,60,14,ES_AUTOHSCROLL
    LTEXT           "Authorization Code
:",IDC_TARGET_TEXT,120,85,65,13
    PUSHBUTTON      "&Attack",IDC_ATTACK,155,135,50,14
    LISTBOX         IDC_TARGET_LIST,5,90,100,60,LBS_NOINTEGRALHEIGHT
|
                    WS_VSCROLL | WS_TABSTOP
    LTEXT           "Select Target",IDC_STATIC,5,75,100,15
    GROUPBOX        " Selected Mission Parameters
      ",IDC_STATIC,115,10,145,
                    110
    LTEXT           "Mission :",IDC_STATIC,120,35,60,8
    LTEXT           "Target :",IDC_STATIC,120,60,60,8
    EDITTEXT        IDC_MISSION_VALUE,190,35,60,14,ES_AUTOHSCROLL |
                    ES_READONLY
    EDITTEXT        IDC_TARGET_VALUE,190,60,60,14,ES_AUTOHSCROLL |
                    ES_READONLY
END
```

The **Rc.exe** utility compiles the resource file that contains this definition, and the results are linked together with the executable file.

The following line of code creates the dialog box and all the controls that it contains. Notice that the second parameter is the name of the dialog box as it appears in the first line of the resource text.

```
  /*
  ** Create modeless dialog box.
  */
  hExampleDlg = CreateDialog( hInstance,
                              MAKEINTRESOURCE(EXAMPLE_DIALOG),
```

```
                                (HWND)NULL,
                                (DLGPROC)ExampleDlgProc );
```

In the Microsoft Platform SDK, the **RC** utility and the help file for it are found in the Microsoft Platform SDK\Bin directory. The help file Rc.hlp describes all the necessary parts of the resource file, which is officially referred to as the "resource definition script."

In Visual Studio, the Rc.exe and Rc.hlp files are found in the Microsoft Visual Studio\Common\MSDev98\Bin directory.

The second method is to call **CreateWindow()** with the necessary parameters to produce the desired control at the desired location inside a parent window.

An X Windows example is as follows:

```
thisButton =
    XtVaCreateManagedWidget("Fire Phasers", <-- button text
                            commandWidgetClass, <-- the type of widget
                            parentWidget, <-- parent widget
                            NULL );
```

A Win32 example using GDI is as follows:

```
HWND handleToThisButton;

handleToThisButton =
  CreateWindow( "BUTTON",    <-- the type of control
                "Fire Phasers", <-- button text
                WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, <-- the button style
                XpositionInParent,
                yPositionInnParent,
                BUTTONWIDTH,
                BUTTONHEIGHT,
                handleOfParentWindow, <-- parent window
               (HMENU)NUMBER_USED_TO_ID_THIS_CONTROL,
                hInst,
                NULL );
```

Identifying a Control

To communicate with or respond to a control, it is necessary to identify it. This is done using the window handle and a unique ID associated with the control. The handle to a control can be used just as a handle to any window. For example, calling **SetWindowPos()** with the control's window handle can move or make the control larger. Use the ID in the **WindProc()** switch statement to send or capture messages to and from the control.

The following API calls use the ID of the control along with the handle of the parent window:

- **SetDlgItemText()**
- **GetDlgItemText()**
- **GetDlgItemInt()**
- **SetDlgItemInt()**

If the developer uses **CreateWindow()** to build the control, both identification pieces are known.

CreateWindow() returns the handle to the control, and the ninth parameter in the call to **CreateWindow ()** is the unique ID the application associates with that control.

If the developer uses a resource editor to put the control in a containing window, the editor assigns a resource ID to the control or the developer can enter this manually. In either case, as shown below, the Resource.h file that corresponds to the .rc file contains the ID number.

- **GetDlgCtrlID()** returns the ID if it is passed the handle to the control.
- **GetDlgItem()** returns the handle if it is passed the ID of the control.

Communicating with a Control

After the application identifies a control, it can communicate with it. The following are some examples of sending and receiving command/messages from controls:

```
//+
//  programmatically add strings to the list box
//-
// Win32
//
SendMessage(handleToThisListBox, LB_ADDSTRING, 0, TEXT("String One")
);
SendMessage(handleToThisListBox, LB_ADDSTRING, 0, TEXT("String Two")
);
SendMessage(handleToThisListBox, LB_ADDSTRING, 0, TEXT("String
  Three") );

// X11/Motif
//
XmString newString;

      newString = XmStringCreateLocalized("String One");
      XmListAddItem( listWidget, newString, 0);
      XmStringFree(newString);

      newString = XmStringCreateLocalized("String Two");
      XmListAddItem( listWidget, newString, 0);
      XmStringFree(newString);

      newString = XmStringCreateLocalized("String Three");
      XmListAddItem( listWidget, newString, 0);
      XmStringFree(newString);

//+
//  programmatically force the current focus
//  to this control, i.e., make the user select
//  something now!
//-
// Win32
//
SetFocus(  handleToThisListBox );

// Xlib
//
XSetInputFocus( display, listWidget, RevertToParent, timeNow );

//+
```

```
//  programmatically pick a string for them!
//-
// Win32
//
SendMessage(handleToThisListBox, LB_SETCURSEL, 2, 0 );

// X11/Moif
//
XmListSelectPos( listWidget, 2, 0 );

//+
//  Controls can be managed in the WndProc() as follows
//
//  Using the button created in the example above
//-
LRESULT CALLBACK WndProc( HWND hWnd,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
{
  switch (message) {

  case WM_COMMAND:

    switch ( wParam ) {

    //+
    //  this message is received because
    //  the button was clicked by
    //  the user
    //-
    case NUMBER_USED_TO_ID_THIS_CONTROL :
      doSomethingWhenButtonIsPressed();
      break;
```

Device Management

User interface devices include keyboards and mouse devices, tablets, touchpads and other devices. The most used devices in many applications are the mouse and the keyboard.

Capturing Mouse Events

There are more than 30 mouse input messages. These are broken into two cases: client-area mouse messages and nonclient-area mouse messages. A window receives client-area mouse messages when a mouse event occurs in that window's client area. The file Winuser.h (included by Windows.h) defines these message values, as shown in Table 3.

Table 3. Mouse event definitions

Message	Meaning
WM_LBUTTONDBLCLK	The left mouse button was double-clicked.
WM_LBUTTONDOWN	The left mouse button was pressed.
WM_LBUTTONUP	The left mouse button was released.
WM_MBUTTONDBLCLK	The middle mouse button was double-clicked.
WM_MBUTTONDOWN	The middle mouse button was pressed.

WM_MBUTTONUP	The middle mouse button was released.
WM_RBUTTONDBLCLK	The right mouse button was double-clicked.
WM_RBUTTONDOWN	The right mouse button was pressed.
WM_RBUTTONUP	The right mouse button was released.
WM_XBUTTONDBLCLK	Windows 2000 or Windows XP: An X mouse button was double-clicked.
WM_XBUTTONDOWN	Windows 2000 or Windows XP: An X mouse button was pressed.
WM_XBUTTONUP	Windows 2000 or Windows XP: An X mouse button was released.

Nonclient-Area Mouse Messages

A window receives nonclient-area mouse messages when a mouse event occurs in the window but outside the client area. The nonclient area includes the border, the title bar, any scroll bars, a menu and minimize or maximize buttons. Each client-area message has a corresponding nonclient-area message. Nonclient-area messages are defined by including NC in its name, for example, WM_NCLBUTTONUP.

The **lParam** member of the **MSG** structure consists of two SHORT values representing the POINTS structure shown in the following listing. This can give the current location of the mouse pointer. For client-area messages, the (x,y) pair is relative to the window's client area. For non-client–area messages, the (x,y) pair is relative to the upper left corner of the screen.

```
typedef struct tagPOINTS {
  SHORT x;
  SHORT y;
} POINTS, *PPOINTS;
```

The following code shows a message loop in **WinMain()** that captures client-area mouse messages and then determines the (x,y) coordinates of the mouse pointer. (The following example shows only the code relevant to the message loop.)

```
int APIENTRY WinMain( HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR     lpCmdLine,
                      int       nCmdShow)
{

  MSG    msg;
  POINT mouseXY;

...

  while (GetMessage(&msg, NULL, 0, 0))  {

       switch (msg.message) {

       case WM_LBUTTONDOWN:
       case WM_LBUTTONUP:

         mouseXY.x = (SHORT) (LOWORD(msg.lParam));
         mouseXY.y = (SHORT) (HIWORD(msg.lParam));
         break;
```

```
      default :
        break;
      }

      TranslateMessage(&msg);
      DispatchMessage(&msg);
  }

  ...

}
```

The following code shows how to process client-area mouse messages in the **WndProc()** function. (This snippet includes only code that shows where to retrieve the mouse event information.)

```
LRESULT CALLBACK WndProc( HWND hWnd,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
{
  POINT mouseXY;

  switch (message) {

  case WM_LBUTTONDOWN:
  case WM_LBUTTONUP:

    //+
    //   The windowsx.h header must be included to use the
    //   GET_X_LPARAM and GET_Y_LPARAM
    //   macros
    //-
    mouseXY.x = GET_X_LPARAM(lParam);
    mouseXY.y = GET_Y_LPARAM(lParam);

    break;

  ...
}
```

This is a very similar process in Xll as shown in this code:

```
void main() {

  Display *xdisplay;
  XEvent   xEvent;
  int mouseX;
  int mousey;

  while (1) {

    /* wait for next event */
    XNextEvent (xdisplay, &xevent);

    switch (xevent.type) {

    case ButtonPress:
      mouseX = xevent.xbutton.x;
      mousey = xevent.xbutton.y;
```

```
      break;

 }
```

Capturing Keyboard Events

A scan code identifies each physical key on the keyboard. The device driver responsible for servicing the keyboard maps this number to a virtual-key code. The include file Winuser.h defines these virtual key codes. After mapping the scan code, the system places a message that includes the scan code and virtual key code along with other information in the system message queue. Some additional system processing takes place, then the system sends the keyboard message to the process that has the keyboard focus.

Pressing a key causes a WM_KEYDOWN or WM_SYSKEYDOWN message to be placed in the thread message queue attached to the window that has the keyboard focus. Releasing a key causes a WM_KEYUP or WM_SYSKEYUP message to be placed in the queue.

The system posts a WM_CHAR message to the window with the keyboard focus when the **TranslateMessage()** function translates a WM_KEYDOWN message. The WM_CHAR message contains the character code of the key that was pressed.

The following code snippet shows how to manually catch and process keystrokes:

```
//+
//  contrived union used only to show how the bits of the
//  lParam parameter are arranged
//  when handling WM_KEYDOWN messages
//-
typedef union {

  struct {
    unsigned long repeatCount   :16;
    unsigned long scanCode      :8;
    unsigned long extendedChar  :1;
    unsigned long reserved      :4;
    unsigned long altKeyDown    :1;
    unsigned long previousState :1;
    unsigned long transition    :1;
  }bits;

  LPARAM lParam;

}tyKeyData;

//+
//  Win32 Application Window Proc
//-
LRESULT CALLBACK WndProc( HWND   hWnd,
                          UINT   message,
                          WPARAM wParam,
                          LPARAM lParam )
{

  tyKeyData keyData;
  TCHAR     characterCode;

  switch ( message ) {
```

```
case WM_SYSKEYDOWN :
case WM_KEYDOWN    :

  //+
  // just for clarity showing what is in
  // the wParam parameter when WM_KEYDOWN is
  // sent to the window proc
  //-
  characterCode  = ((TCHAR)(wParam));

  //+
  //  the tyKeyData union is defined above
  //  this union displays how the bits are defined
  //-
  keyData.lParam = lParam;


  if ( keyData.bits.altKeyDown ) {

    //+
    //  using the keyboard hardware scan code
    //  to determine what key was pressed
    //-
    switch ( keyData.bits.scanCode ) {
    case 0x3B : // <Alt-F1>
      break;

    case 0x3C : // <Alt-F2>
      break;

    default :
      break;
    }

  }
  else {

    //+
    //  VK_XX Key Codes are found in winuser.h
    //
    //  These are not the keyboard hardware scan codes!!!
    //
    //  using the wParam to determine
    //  what key was pressed
    //-
    switch ( characterCode ) {
    case VK_F1 : // <F1>
      break;

    case VK_F2 : // <F2>
      break;

    default :
      break;
    }

  }

  break;

case WM_CHAR    :
```

```
    characterCode  = ((TCHAR)(wParam));

    switch ( characterCode ) {

    //+
    //  VK_XX codes can be used here
    //  VK_XX Key Codes are found in winuser.h
    //-
    case 0x08:  // backspace
    case 0x0A:  // linefeed
    case 0x1B:  // escape
      break;

    case VK_LEFT   :  // left arrow
    case VK_UP     :  // up arrow
    case VK_INSERT :  // the insert key
      break;



    //+
    //    convert TAB to Spaces
    //-
    case 0x09:  // tab

      for ( int i = 0; i < 4; i++)
        SendMessage(hWnd, WM_CHAR, 0x20, 0);

      return 0;
```

Keyboard Focus

Keyboard focus is a temporary property of a window or widget. At any given time, only one component can listen to the keyboard for events. The window or widget that is listening is said to have the current focus, keyboard focus, or just focus.

Processing focus in a Win32-based application involves processing the WM_KILLFOCUS and WM_SETFOCUS Windows Messages. This is similar to using **XmNfocusCallback** and **XmNlosingFocusCallback** for focus callbacks setup with in X/Motif.

The following code fragment shows the window procedure for a subclassed "button" that is handling focus messages:

```
LRESULT CALLBACK CSoftKeyProc(HWND hWnd,UINT iMsg, WPARAM
wParam,LPARAM lParam)
{
   LRESULT   lResult    = FALSE;

  //+
  //    this is a trick to retrieve data (in this case a pointer )
    that is attached to this window object.
  //    SetWindowLongPtr() was used to initially attach this data to
    the window. The reason for this
  //    being used here is that this function may be the callback for
    any number of these type
  //    objects and this data is "state" for this particular instance
  //-
```

```
  CvtSoftKey *pSoftKey   = (CvtSoftKey *)GetWindowLongPtr( hWnd, 0 );

  switch (iMsg) {
  default :
    break;

  //+
  //  this window (button in this case) is receiving the focus
  //  so we can do whatever processing we like
  //  Draw a new border -- Highlight the text -- whatever!
  //-
  case WM_SETFOCUS   :
    lResult  = pSoftKey->OnSetFocus( hWnd,iMsg,wParam,lParam);
    break;

  //+
  //  this window ( button in this case ) is losing focus
  //-
  case WM_KILLFOCUS  :
    lResult  = pSoftKey->OnKillFocus( hWnd,iMsg,wParam,lParam);
    break;
  }
   return DefWindowProc(hWnd,iMsg,wParam,lParam);
}
```

Table 4 shows functions to use for getting current focus.

Table 4. Functions for getting current focus

X Windows	Win32
No equivalent	**GetActiveWindow()**
XGetInputFocus()	**GetForegroundWindow()**
XmGetFocusWidget()	**GetFocus()**

Table 5 shows functions to use for setting current focus.

Table 5. Functions for setting current focus

X Windows	Win32
No equivalent	**SetFocus()**
No equivalent	**SetActiveWindow()**
XSetInputFocus()	**SetForegroundWindow()**

Creating Keystrokes, Mouse Motions, and Button Click

A developer can simulate keystrokes, mouse motions, or button clicks by using the **SendInput()** function to serially insert events into the mouse or keyboard stream.

For additional information about handling the keyboard, search the MSDN Web site for **keybd_event**, **GetRawInputData**, **GetKeyState**, **GetAsyncKeyState**, **GetKeyboardState** and **MapVirtualKey()**.

Displaying Text

Text can be surprisingly complex. To make the visual display of the text as readable as possible requires creation and use of fonts, then decisions about mapping modes, kerning and more.

Using Fonts

Fonts control the display characteristics of text. An X Windows client application can use the **XLoadQueryFont()** and **XSetFont()** functions to apply a font to a given graphics context (GC), as shown in the following code:

```
#define FONT1 "-*-lucida-medium-r-*-*-12-*-*-*-*-*-*-*"

Font           font1;
XFontStruct   *font1Info;

main() {

  Display  *pDisplay;
  int       iScreen;
  GC        gc;

  pDisplay  = XOpenDisplay("myDisplay");
  iScreen   = DefaultScreen(pDisplay);

  //+
  //  get the Graphics Context
  //-
  gc        = DefaultGC(pDisplay,iScreen);

  //+
  //  attempt to load the font
  //-
  font1Info = XLoadQueryFont( pDisplay,FONT1 );
  font1     = font1Info->fid;

  //+
  //  Set the font in the GC
  //-
  XSetFont( pDisplay, gc,  font1 );

   ...
   ...
```

A Win32-based application follows much the same logic. That is, it creates or selects a font, retrieves a device context (DC), and then selects the font object to that DC.

```
#define FONT1  TEXT("Lucida Console");

HFONT hFont1;

void fontDemo(  HWND hWnd  )
{
  HDC   hDC;
  HFONT hOldFont;

  //+
  //  get the Device Context
  //-
  hDC    = GetDC( hWnd );
```

```
//+
//  attempt to load the system font
//-
hFont1 = (HFONT)GetStockObject (SYSTEM_FONT);

//+
//  Set the font in the GC
//-
hOldFont = (HFONT)SelectObject( hDC, hFont1 );

…
```

Table 6 shows how to reference fixed font types.

Table 6. Fixed font references

Font Reference	Font Type
ANSI_FIXED_FONT	Windows fixed-pitch (monospace) system font.
ANSI_VAR_FONT	Windows variable-pitch (proportional space) system font.
DEVICE_DEFAULT_FONT	Microsoft Windows NT®, Windows 2000, or Windows XP operating system: Device-dependent font.
DEFAULT_GUI_FONT	Default font for user interface objects such as menus and dialog boxes. This is Microsoft Sans Serif. Compare this with SYSTEM_FONT.
OEM_FIXED_FONT	Original equipment manufacturer (OEM)-dependent fixed-pitch (monospace) font.
SYSTEM_FONT	System font. By default, the system uses the system font to draw menus, dialog box controls, and text. Windows 95/98 or Windows NT: The system font is Microsoft Sans Serif. Windows 2000 or Windows XP: The system font is Tahoma.
SYSTEM_FIXED_FONT	Fixed-pitch (monospace) system font. This stock object is provided only for compatibility with 16-bit Windows earlier than version 3.0.

Note Win32 provides several utilities for adding and editing fonts.

The **Eudcedit.exe** utility, which comes with the operating system, allows the user to create unique characters such as logos and special characters. Eudcedit Help describes how to create, store, and use these characters in the font library.

The **Charmap.exe** utility, which comes with the operating systems, allows the user to view, find, and copy characters from the Windows, MS-DOS and Unicode character sets. Charmap Help describes how to do this.

The **Fontedit.exe** utility, which comes with Visual Studio, allows the user to create and edit raster fonts.

Creating Fonts

Developers might want to create fonts to use in an application. The short example in this section uses the font specified by SYSTEM_FONT, which duplicates the default, although developers are likely to use something more creative. The Win32 **CreateFont()**, **CreateFontIndirect()** and **CreateFontIndirectEx()** functions provide the ability to create logical fonts based on the fonts loaded on the system.

```
#define MY_FONT_FACE TEXT("Lucida Console")
//+
//  fontAttribute Option Bits
//-
#define fontAttribute_BOLD                      0x01
#define fontAttribute_CROSSED_OUT               0x02
#define fontAttribute_UNDERLINED                0x04
#define fontAttribute_ITALIC                    0x08

typedef struct {
  unsigned char   fontSize;
  unsigned char   fontStyle;
  TCHAR          *fontFace;
} tyFONT_ATTRIBUTE;

HFONT createFont( tyFONT_ATTRIBUTE *fontAttributeObject )
{

  HFONT   hFont;
  LOGFONT lf;

  //+
  //  these are completely arbitrary values for this example code.
  //  they simply associate a width and height with a
  //  font size number found in the tyFONT_ATTRIBUTE struct.
  //
  //  For example fontSize == 2 (used to index these two arrays)
  //  will produce a 12x8 font
  //-
  int fontHeight[] = {8,8,12,16,16,24,32,
    32,48,64,64,96,128,128,192};
  int fontWidth [] = {6,8, 8,12,16,16,24, 32,32,48,64,64,
    96,128,128};

  //+
  //  pick a font face
  //-
  lstrcpy( lf.lfFaceName, fontAttributeObject->fontFace );

  //+
  //  protect against running out of the arrays above
  //  and pick a default behavior of "2"
  //-
  if ( fontAttributeObject->fontSize  > 14 )
    fontAttributeObject->fontSize = 2;


  if ( fontAttributeObject->fontStyle & fontAttribute_BOLD )
    lf.lfWeight  = FW_MEDIUM;
  else
    lf.lfWeight  = FW_LIGHT;
```

```
  lf.lfItalic         = (unsigned char)( fontAttributeObject-
    >fontStyle &
                                         fontAttribute_ITALIC      );

  lf.lfUnderline      = (unsigned char)( fontAttributeObject-
    >fontStyle &
                                         fontAttribute_UNDERLINED  );

  lf.lfStrikeOut      = (unsigned char)( fontAttributeObject-
    >fontStyle &
                                         fontAttribute_CROSSED_OUT );

  lf.lfEscapement     = 0;
  lf.lfOrientation    = 0;
  lf.lfCharSet        = ANSI_CHARSET;
  lf.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  lf.lfQuality        = DRAFT_QUALITY;
  lf.lfPitchAndFamily = FF_MODERN | FIXED_PITCH;

  lf.lfHeight         = fontHeight [ fontAttributeObject->fontSize ];
  lf.lfWidth          = fontWidth  [ fontAttributeObject->fontSize ];

  hFont               = CreateFontIndirect(&lf);

  return( hFont );
}

//+
//  example using createFont()
//-
void fontDemo(  HWND hWnd  )
{

  HDC              hDC;
  HFONT            hOldFont;
  HFONT            hFont1;
  tyFONT_ATTRIBUTE fontAttribute;

  //+
  //  get the Device Context
  //-
  hDC    = GetDC( hWnd );

  //+
  //   attempt to create a font
  //-
  fontAttribute.fontSize  = 2;
  fontAttribute.fontStyle = (fontAttribute_BOLD |
    fontAttribute_ITALIC );
  lstrcpy( fontAttribute.fontFace, MY_FONT_FACE );

  hFont1 = createFont( &fontAttribute );

  //+
  //  Set the font in the GC
  //-
  hOldFont = (HFONT)SelectObject( hDC, hFont1 );

  …
  …
```

For more information about creating and using logical fonts in a Win32-based application, see The Logical Font.

Device vs. Design Units

An application can retrieve font metrics for a physical font only after the font has been selected in a device context. When a user selects a font in a device context, the system scales the font for the device. The font metrics specific to the device are known as *device units*.

Portable metrics in fonts are known as *design units*. To apply to a specified device, convert design units to device units by using the following formula:

```
DeviceUnits = (DesignUnits/unitsPerEm) * (PointSize/72) *
  DeviceResolution
```

For a full explanation of device units, design units and pixels, see the operating system Help or search the MSDN Web site.

Windows Character Data Types

Table 7 lists the Windows character types. Most of the pointer-type names begin with a prefix of P or LP. For additional information about character sets used by fonts, see the operating system Help or search the MSDN Web site.

Table 7. Windows character types

Reference	Character Type
CHAR	An 8-bit Windows (ANSI) character
LPCSTR	Pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters
LPCTSTR	LPCWSTR if UNICODE is defined, LPCSTR otherwise
LPCWSTR	Pointer to a constant null-terminated string of 16-bit Unicode characters
LPSTR	Pointer to a null-terminated string of 8-bit Windows (ANSI) characters
LPTSTR	LPWSTR if UNICODE is defined, LPSTR otherwise
PCHAR	Pointer to CHAR
PCSTR	Pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters
PCTSTR	PCWSTR if UNICODE is defined, PCSTR otherwise
PCWCH	Pointer to a constant WCHAR
PCWSTR	Pointer to a constant null-terminated string of 16-bit Unicode characters
PSTR	Pointer to a null-terminated string of 8-bit Windows (ANSI) characters
PTCHAR	Pointer to TCHAR
PTSTR	PWSTR if UNICODE is defined, PSTR otherwise
PWSTR	Pointer to a null-terminated string of 16-bit Unicode characters

TBYTE	WCHAR if UNICODE is defined, CHAR otherwise
TCHAR	WCHAR if UNICODE is defined, CHAR otherwise
WCHAR	A 16-bit Unicode character

A best practice with characters is to declare all characters and strings as TCHAR and use the TEXT() macro to declare static strings. For more information about wsprintf() and the rest of the string functions, see the operating system Help or search the MSDN Web site.

```
TCHAR myString[255];

wsprintf( myString,
          TEXT("This is a good example %d is a %s \n" ),
          1950,
          TEXT("Year")
        );
```

Drawing Text

Drawing text can be simple or complicated. Because simple is often better, this discussion starts with the X Windows **XDrawString()** function and the Win32 **TextOut()** function. Both functions require a context to draw on, the x and y coordinates, the string and the string length, in characters. The examples in this section draw the string "Hello World" in the current font and colors at the specified coordinates.

It is often desirable to set a particular font or color before writing the text. These examples show how the two systems perform these tasks.

A programmer can code font and text display in Win32 as follows:

```
#define rgbBlack (COLORREF)RGB( 0x00,0x00,0x00 )
#define rgbWhite (COLORREF)RGB( 0xFF,0xFF,0xFF )
font = (HFONT)GetStockObject(OEM_FIXED_FONT);
oldFont = (HFONT)SelectObject( hdc, font ); // save old font
SetTextColor (hdc, rgbBlack);
SetBkColor   (hdc, rgbWhite);
TextOut( hdc, x, y, "Hello World", 11);
```

The Win32 code example above uses the COLORREF type, the **RGB()** macro, and the **GetStockObject ()** function. These are explained in the following list:

- The COLORREF value used in the Win32 example code is used to specify an RGB color and is defined as shown here:

  ```
  typedef DWORD COLORREF;
  typedef DWORD *LPCOLORREF;
  ```

- The **RGB** macro selects a red, green, blue (RGB) color based on the arguments supplied and the color capabilities of the output device, as shown here:

  ```
  COLORREF RGB(
    BYTE Error! Hyperlink reference not valid.,    // red component
  of color
    BYTE Error! Hyperlink reference not valid.,  // green component
  of color
    BYTE Error! Hyperlink reference not valid.    // blue component
  ```

```
of color
);
```

- The **GetStockObject(int objectType)** function retrieves a handle to one of the stock pens, brushes, fonts, or palettes. The return value must be cast to the expected type, as shown here:

```
void foo() {
  HFONT hFont;
  HBRUSH hBrush;
  hfont   = (HFONT)GetStockObject(DEFAULT_GUI_FONT);
  hBrush  = (HBRUSH)GetStockObject(BLACK_BRUSH);
}
```

A programmer can code font and text display in X Windows as follows:

```
font = XLoadQueryFont (display, "fixed");
XSetFont (display, gc, font->fid);

XSetBackground(display, gc, WhitePixel(display, screen));
XSetForeground(display, gc, BlackPixel(display, screen));
XDrawString( display, d, gc, x, y, "Hello World", 11 );
```

X Windows provides explicit definitions of 8-bit and 16-bit character functions with **XDrawString()** and **XDrawString16()**. Likewise, Win32 provides **TextOutA()** for ASCII (8-bit characters) and **TextOutW()** for Wide Char (16-bit UNICODE characters). The **TextOut()** function is actually a macro that resolves correctly to **TextOutA()** or **TextOutW()** based on the status of the UNICODE definition, as follows:

```
#define UNICODE
#define _UNICODE

TextOut()…  // this will result in TextOutW()

#undef UNICODE
#undef _UNICODE

TextOut()…  // this will result in TextOutA()
```

One drawback of using **XDrawString()** and **TextOut()** is that nothing is done about erasing the background. Continually outputting strings to the same x and y coordinates results in a jumble of unreadable text strings one upon the other. The X Windows library provides the **DrawImageString()** function, which calculates a rectangle containing the string and fills it with the background pixel color before drawing the text in the foreground pixel color. Win32 supports the **ExtTextOut()** function to provide this capability. Using the Win32 **ExtTextOut()** function requires the bounding rectangle to be calculated and passed into the function. This requires knowledge about the current font and logical display units.

Calculating Text Metrics

The X Windows programmer can rely on **XTextWidth()** to get the length of a character string in pixels. The Win32 programmer must work a little harder to get this number.

Understanding mapping mode

First, it is necessary to understand mapping mode. The mapping mode defines the unit of measure used to

transform page-space units into device-space units. It also defines the orientation of the device's x and y axes.

A mapping mode is a scaling transformation that specifies the size of the units used for drawing operations. The mapping mode can also perform translation. In some cases, the mapping mode alters the orientation of the x and y axes in device space.

The default mapping mode is MM_TEXT. One logical unit equals one pixel. Positive x is to the right, and positive y is down. This mode maps directly to the device's coordinate system.

The Win32 **SetMapMode** function sets the mapping mode of the specified device context, as shown in the following code:

```
int SetMapMode(
  HDC hdc,        // handle to device context
  int fnMapMode   // new mapping mode
);
```

Ultimately, to calculate the size of a string in pixels it is necessary for the current mapping mode to be MM_TEXT. The Win32 programmer can assume the current mapping mode is the default MM_TEXT, set it to MM_TEXT by calling **SetMapMode()**, or make sure it is MM_TEXT by using **GetMapMode()** to retrieve it. (For more information, search for MM_TEXT on the MSDN Web site, http://msdn.microsoft.com/.)

Calculating character size and string length

The Win32 **GetTextExtentPoint32()** function returns the width and height of a string of text in logical units, as shown in the following lines of code. (Recall that setting the mapping mode to MM_TEXT returns logical units as pixels.)

```
BOOL GetTextExtentPoint32(
  HDC     hdc,       // handle to DC
  LPCTSTR lpString,  // text string
  int     cbString,  // characters in string
  LPSIZE  lpSize     // string size
);
```

The size structure looks like the following and is defined in Windef.h:

```
typedef struct tagSIZE {
  LONG Error! Hyperlink reference not valid.;
  LONG Error! Hyperlink reference not valid.;
} SIZE, *PSIZE, *LPSIZE;
```

The Win32 **GetTextMetrics()** function fills a TEXTMETRIC structure with all the information about the device context's currently selected font, as shown in the following lines of code. The programmer can use this information to perform any number of scaling or text size calculations.

```
BOOL GetTextMetrics(

  HDC          hdc,   // handle to DC
  LPTEXTMETRIC lptm   // text metrics
);
```

The TEXTMETRIC structure contains basic information about a physical font, as shown in the following example. All sizes are specified in logical units; that is, they depend on the current mapping mode of the display context.

```
typedef struct tagTEXTMETRIC {
  LONG  tmHeight;
  LONG  tmAscent;
  LONG  tmDescent;
  LONG  tmInternalLeading;
  LONG  tmExternalLeading;
  LONG  tmAveCharWidth;
  LONG  tmMaxCharWidth;
  LONG  tmWeight;
  LONG  tmOverhang;
  LONG  tmDigitizedAspectX;
  LONG  tmDigitizedAspectY;
  TCHAR tmFirstChar;
  TCHAR tmLastChar;
  TCHAR tmDefaultChar;
  TCHAR tmBreakChar;
  BYTE  tmItalic;
  BYTE  tmUnderlined;
  BYTE  tmStruckOut;
  BYTE  tmPitchAndFamily;
  BYTE  tmCharSet;
} TEXTMETRIC, *PTEXTMETRIC;
```

More Win32 Text Functions

The following Win32 functions are also useful for working with text:

- **DrawText()**
- **CreateSolidBrush()**
- **GetSysColor()**
- **SetTextColor()**
- **GrayString()**

This section discusses and shows examples for these functions.

The **DrawText()** function draws formatted text in the specified rectangle, as shown in the following example. It formats the text according to the specified method, expanding tabs, justifying characters, breaking lines, and so forth.

```
int DrawText(
  HDC     hDC,      // handle to DC
  LPCTSTR lpString, // text to draw
  int     nCount,   // text length
  LPRECT  lpRect,   // formatting dimensions
  UINT    uFormat   // text-drawing options
);
```

The **CreateSolidBrush()** function creates a logical brush that has the specified solid color, as shown in the following example:

```
HBRUSH CreateSolidBrush(
```

```
  COLORREF crColor // brush color value
);
```

The **GetSysColor()** function retrieves the current color of the specified display element, as shown in the following example. Display elements are the parts of a window and the Windows display that appear on the system display screen.

```
DWORD GetSysColor( int nIndex );
```

The **SetTextColor()** function sets the text color for the specified device context to the specified color, as shown in the following example:

```
COLORREF SetTextColor(
  HDC      hdc,           // handle to DC
  COLORREF crColor        // text color
);
```

The following example incorporates the use of **DrawText()**, **CreateSolidBrush()**, **GetSysColor()** and **SetTextColor()**:

```
RECT myRectangle;
//+
//   create a brush
//-
HBRUSH myBackgroundBrush =
  CreateSolidBrush(
    GetSysColor(COLOR_BACKGROUND) // color of system background
  );

//+
//  set the text color to the system's button text color
//-
SetTextColor(
  hdc,
  GetSysColor(COLOR_BTNTEXT) //color of text on buttons
);

// calculate myRectangle

//+
//  fill in (erase) the area inside the rectangle with the
//  system's background color
//-
FillRect( hdc, &myRectangle, myBackgroundBrush);

//+
//  The DrawText function uses the device context's selected font,
text
//  color, and background color to draw the text. Unless the
  DT_NOCLIP
//  format is used, DrawText clips the text so that it does not
appear
//  outside the specified rectangle.
//-
DrawText( hdc,
          myString,
          _tcsclen(myString), // use _tcsclen() vs. strlen()
          &myRectangle,
```

```
          (DT_CENTER | DT_SINGLELINE )
        );
```

The **GrayString()** function draws gray text at the specified location, as shown in the following example. The function draws the text by copying it into a memory bitmap, graying the bitmap and then copying the bitmap to the screen. The function grays the text regardless of the selected brush and background. **GrayString()** uses the font currently selected for the specified device context.

```
BOOL GrayString(
  HDC              hDC,           // handle to DC
  HBRUSH           hBrush,        // handle to the brush
  GRAYSTRINGPROC   lpOutputFunc,  // callback function
  LPARAM           lpData,        // application-defined data
  int              nCount,        // number of characters
  int              X,             // horizontal position
  int              Y,             // vertical position
  int              nWidth,        // width
  int              nHeight        // height
);
```

Text Widgets and Controls

A text widget or control is used to display, enter, and edit text. The exact functionality of a text widget or control depends upon how its resources are set.

In X Windows, the widget functionality is set as shown in the following example:

```
text = XtVaCreateManagedWidget ( "myTextWidget",
                                 asciiTextWidgetClass,
                                 parentWidget,
                                 XtNfromHoriz,
                                 quit,
                                 XtNresize,
                                 XawtextResizeBoth,
                                 XtNresizable,
                                 True,
                                 NULL);
```

In Motif, the widget functionality is set as shown in the following example:

```
main (int argc, char *argv[])
{
  Widget       mainWidget;
  Widget       textWidget;
  XtAppContext appContext;

  mainWidget =

    XtVaOpenApplication ( &appContext,
                          "TextExample",
                          NULL,
                          0,
                          &argc,
                          argv,
                          NULL,
                          sessionShellWidgetClass,
                          NULL);
```

```
  (…)

  textWidget =

    XmCreateText ( mainWidget,"textWidget",NULL,0);

  (…)

  XtAppMainLoop( appContext );

}
```

In Win32 and GDI, the control functionality is set as shown in the following example:

```
//+
// Create an edit Control.
//-
HWND handleToThisEditControl;

handleToThisEditControl =

  CreateWindow( TEXT("EDIT"),      //<-- the type of
control
                TEXT("Some Text"), //<-- edit control text

                (WS_CHILD |
                   WS_VISIBLE |
                     ES_READONLY |
                       ES_LEFT |
                          ES_UPPERCASE), //<-- the control
style

                XpositionInParent,
                yPositionInnParent,
                CONTROL_WIDTH_IN_DEVICE_UNITS,
                CONTROL_HEIGHT_IN_DEVICE_UNITS,
                handleOfParentWindow, //<-- parent window
                (HMENU)NUMBER_USED_TO_ID_THIS_EDIT_CONTROL,
                appContext,
                NULL );


//+
//  Turn off Read Only
//-
SendMessage( handleToThisEditControl ,
             EM_SETREADOINLY,
             (WPARAM)FALSE,   //<-- set read only false
             (LPARAM)NULL );

//+
//  set the edit control's text
//-
SetWindowText( handleToThisEditControl, TEXT("Some New Text") );

//+
//  retrieve the edit control's text as text
//-
GetWindowText( handleToThisEditControl,
               myStringBuffer,
```

```
                myStringBufferMaxSize );

//+
//  retrieve the edit control's text as an integer
//-
myIntegerValue =
  GetDlgItemInt( handleOfParentWindow,
                 NUMBER_USED_TO_ID_THIS_EDIT_CONTROL,
                 &resultFlag,   // did the translation succeed ?
                 FALSE );       // no this is an unsigned number
```

Drawing

Drawing functions set the way to present graphical information on the screen. These range from primitive functions such as turning a pixel on or off, to complex 2-D and 3-D drawing functions. There are some notable differences in how drawing works on X Windows and Win32 platforms.

Device Context

Applications on both platforms use a context to control how drawing functions behave. On X Windows systems, this context is known as the graphics context (GC). On Win32-based GDI systems, this context is known as the device context (DC).

The first difference is in where the operating system stores and manages drawing attributes such as the width of lines or the current font.

In X Windows, these values belong to the graphics context. When using **XCreateGC()** or **XtGetGC()**, it is necessary to provide a values mask and values structure. These values are used to store settings such as line width, foreground color, background color, and font style.

The following code is an example of the process of setting the foreground and background colors:

```
GC              gcRedBlue;
XGCValues       gcValues;
unsigned long   gcColorRed;
unsigned long   gcColorBlue;
unsigned long   gcColorWhite;
Widget          myWidget;

int main (int args, char **argv)
{
  // initialize colors - widget - etc.

  gcValues.foreground = gcColorRed;
  gcValues.background = gcColorBlue;

  gcRedBlue = XtGetGC ( myWidget, GCForeground | GCBackground,
    &gcValues);
}
```

Win32-based applications use a different approach. The device context is a structure that defines a set of graphic objects and their associated attributes, as well as the graphic modes that affect output. To keep this simple, the graphic objects include a font for displaying text, a pen for line drawing, and a brush for painting and filling. To draw lines, rectangles, text, and so on, it is necessary to get or create one of these

objects and select it into the desired DC.

Rather than create several specialized GC objects like X Windows does, Win32-based applications create several drawing objects and then select them into the DC as required. This methodology is similar to what an X Windows client application could do by getting a single GC and then repeatedly calling **XChangeGC()**.

The following code snippet shows a Win32-based application that creates several pens and then uses them to draw lines and rectangles:

```
#define onePixel    1
#define threePixels 3

#define thinLine    onePixel
#define thickLine  threePixels

COLORREF colorRed;
COLORREF colorBlue;

void drawSomthing( HDC hDC )
{

  HPEN thinRedPen;
  HPEN thinBluePen;

  HPEN oldPen;
  int  x;
  int  y;

  // initialize colors - etc.

  //+
  //  create two pens.
  //  this could be done more statically somewhere so that
  //  it would not be necessary to create them each time
  //  this method is called.
  //-
  thinRedPen  = CreatePen( PS_SOLID,  thinLine, colorRed );
  thinBluePen = CreatePen( PS_SOLID, thinLine, colorBlue );
  x = 100;
  y = 200;

  //+
  //  draw a line with the current pen,
  //  whatever it is at this time for this DC
  //-
  LineTo( hDC, x,y );

  //+
  //  make our pen the current pen for the DC
  //  and save the existing one so we can put it back
  //-
  oldPen = (HPEN)SelectObject(  hDC, thinRedPen );

  //+
  //  draw a line with our pen
  //-
  LineTo( hDC, x,y );
```

```
  //+
  //  make our other pen current in the DC.
  //  we are not saving the old one.
  //-
  SelectObject ( hDC, thinBluePen );

  //+
  //  draw a line using our second pen
  //-
  LineTo( hDC, x, y );

  //+
  //  put back the original pen
  //-
  SelectObject( hDC, oldPen );

  //+
  //  get rid of our pen resources
  //-
  DeleteObject( thinRedPen );
  DeleteObject( thinBluePen );
}
```

Getting Win32 GDI Device Context (DC)

Win32-based applications can retrieve the device context from the window handle, as shown in the following example:

```
void myFunction (  HWND hWnd  )
{
  //+
  //  retrieve the DC of the
  //  window referenced by hWnd
  //-
  hDC = GetDC( hWnd );

  // draw using the device context hDC

  //+
  // release the DC
  //-
  ReleaseDC( hWnd, hDC );
```

Win32-based applications can also retrieve the device context in the Windows Proc by using **BeginPaint ()** and **EndPaint()**, as shown in the following code:

```
LRESULT CALLBACK WndProc( HWND   hWnd,
                          UINT   message,
                          WPARAM wParam,
                          LPARAM lParam)
{

  HDC            hDC;
  PAINTSTRUCT    ps;

  switch ( message ) {
```

```
case WM_PAINT:

  //+
  //   Retrieve the device context (DC)
  //   for the window referenced by hWnd
  //-
  hDC = BeginPaint( hWnd, &ps );

  //+
  //  draw with hDC or ps.hdc
  //-

  //+
  // always follow BeginPaint() with EndPaint()
  //-
  EndPaint( hWnd, &ps );
```

Creating Win32 GDI Device Context (DC)

It is often useful to draw in an off-screen buffer and then move that buffer into the display memory. This hides the live drawing function calls from the user and eliminates "flicker" in the window.

To create this off-screen context

1. Calculate the width and height that are needed.
2. Get the DC of the target (dialog box, button, or any other window object).
3. Call CreateCompatibleDC.
4. Call CreateCompatibleBitmap.

```
// m_hButton is the window handle to a button.
// m_clientRect is a RECT structure.

// Step 1. Calculate the size.
GetClientRect( m_hButton, &m_clientRect );
m_width       = ((int)( m_clientRect.right  - m_clientRect.left ));
m_height      = ((int)( m_clientRect.bottom - m_clientRect.top  ));

// Step 2. Get the DC of the target window.
hdc           = GetDC( m_hButton );

// Step 3. Create a compatible device context.
m_hdcMem      = CreateCompatibleDC(hdc);

// Step 4. Create a compatible bitmap - our X Windows drawable.
m_hbmpMem     = CreateCompatibleBitmap( hdc,m_width,m_height );
```

To use and display this off-screen bitmap

1. Select the compatible bitmap into the compatible device context.
2. Draw on that DC.
3. Get the target window DC.
4. Transfer the compatible memory image to the screen.
5. Select the old bitmap into the DC.

```
// Step 1. Select the compatible bitmap into the compatible DC.
// hbmpOld is a handle to a bitmap
// m_hdcMem is the compatible device context
// m_hbmpMem is the compatible bitmap
hbmpOld = (HBITMAP)SelectObject( m_hdcMem, m_hbmpMem );

// Step 2. Draw on that DC.
// FillRect() cleans out the rectangle
FillRect( m_hdcMem, &m_clientRect, hBackgroundBrush );

// Draw a line
LineTo( m_hdcMem, x,y );


// Step 3. Get the target DC.
targetDC =  GetDC( hTargetWindow );

// Step 4. Transfer the compatible image to the screen.
// transfer everything to the screen
// hdcMem is what we drew on
//-
BitBlt( targetDC,
        0,
        0,
        m_width,
        m_height,
        m_hdcMem,
        0,
        0,
        SRCCOPY );

// Step 5. Put the old bitmap back into the compatible DC.
SelectObject( m_hdcMem, hbmpOld  );

// based on program logic - Release the DC of the target window
ReleaseDC( hTargetWindow, targetDC );
```

For additional information about Win32-based GDI device context, search the MSDN Web site (http://msdn.microsoft.com/) for **GetDC**, **CreateDC**, **CreateCompatibleDC**, and **DeleteDC**.

Display and Color Management

X Windows and Win32-based GDI are both constrained by the physical limitations of the available display hardware. One such limitation is the number of colors a display adapter is capable of showing.

All X Windows applications use a color map. This map can be shared or private. A shared color map is used by all other applications that are not using a private map. Using a private map gives an application better color control and potentially a greater number of colors. There is one problem with private maps: When the mouse moves on or off the client by using a private map, the screen colors change.

Win32-based applications typically use color with no regard for the display device. If the application uses a color that is beyond the capabilities of the display device, the system approximates that color within the limits of the hardware. On display devices that support a color palette, applications sensitive to color quality can create and manage one or more logical palettes.

A palette is conceptually similar to an X Windows color map. Both of these methodologies are used to

"map" some desired colors onto the physical capabilities of the display hardware. For example, if a Win32-based program needs more than 16 colors and is running on an 8-bits-per-pixel (bpp) display adapter, the program must create and use a palette.

The Win32 system palette can be thought of as similar to an X Windows shared color map. A logical palette created and realized by an application can be thought of as an X Windows private color map.

A Win32-based application that uses a logical palette exhibits some of the same behaviors as an X Windows application that uses a private color map. The application that gets priority in color selection is the one with the current focus. When the application that has the current focus calls **RealizePalette()**, the system palette changes and the WM_PALETTECHANGED message is sent to all top-level and overlapped windows. This message enables a window that uses a color palette but does not have the keyboard focus to realize its logical palette and update its client area. The wParam parameter identifies the owner window. Inspecting this value prevents the originating window from realizing a logical palette over and over again upon receipt of this message.

Today, most display hardware is capable of 24-bit or better color depth. For palette examples, see the many samples both on the MSDN Web site and in Microsoft Windows Platform SDK.

To create a logical color palette

1. Allocate a LOGPALETTE structure and assign values to it.
2. Call **CreatePalette()** with a pointer to the LOGPALETTE structure.
3. Call **SelectPalette()** by using the pointer returned from **CreatePalette()**.
4. Call **RealizePalette()** to make the system palette the same as the DC.
5. Call **UnrealizeObject()** when finished with the palette.

To determine the capabilities of the hardware and calculate the best possible behaviors of the display, an X Windows program can use functions such as **DefaultColorMap()**, **DefaultVisual()**, **DisplayCells()**, **DisplayPlanes()**, **XGetVisualInfo()** or **XGetWindowAttributes()**.

A Win32-based application can rely on **GetDeviceCaps()** for this information. The **GetDeviceCaps()** function retrieves device-specific information for the specified device. The following code example shows a few examples of device information that can be retrieved by using **GetDeviceCaps()**. For a full list of the possible values of the nIndex parameter, see the operating system Help or search the MSDN Web site (http://msdn.microsoft.com/).

```
int GetDeviceCaps(
  HDC hdc,      // handle to DC
  int nIndex    // index of capability
);

void myFunction( HWND hThisWindow )
{
  HDC hDC;

  hDC = GetDC( hThisWindow );

  widthOfScreenInPixels  = GetDeviceCaps( hDC, HORZRES );
  numberOfColorPlanes  = GetDeviceCaps( hDC, PLANES );
  numberOfColors     = GetDeviceCaps( hDC, NUMCOLORS );
  numberOfFonts     = GetDeviceCaps( hDC, NUMFONTS );

}
```

Drawing 2-D Lines and Shapes

The device context of a drawing surface contains attributes that directly affect how lines, curves, and rectangles are drawn. These attributes include the current brush and pen and current position.

The default current position for any given DC is (0,0) in logical (world) 2-D space. The value of the current position can be changed by calling **MoveToEx()**, as shown in the following code example. The **MoveToEx()** function updates the current position to the specified point and optionally returns the previous position. This function affects all drawing functions.

```
BOOL MoveToEx(
  HDC hdc,          // handle to device context
  int X,            // x-coordinate of new current position
  int Y,            // y-coordinate of new current position
  LPPOINT lpPoint   // old current position
);
```

The POINT structure defines the x and y coordinates of a point, as shown in the following code example:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT;
```

Drawing lines

Two sets of line and curve drawing functions are provided in the Win32 GDI API. These two sets of functions are identified by the letters, "To" at the end of the function name. Functions ending with "To" use and set the current position. Those that do not end with "To" leave the current position as it was.

The **LineTo** function draws a line from the current position up to, but not including, the specified point, as shown in the following code example:

```
BOOL LineTo(
  HDC hdc,    // device context handle
  int nXEnd,  // x-coordinate of ending point
  int nYEnd   // y-coordinate of ending point
);
```

The **PolylineTo** function draws one or more straight lines that use and update the current position. A line is drawn from the current position to the first point specified by the lppt parameter by using the current pen. For each additional line, the function draws from the ending point of the previous line to the next point specified by lppt, as shown in the following code example:

```
BOOL PolylineTo(
  HDC          hdc,   // handle to device context
  CONST POINT *lppt,  // array of points
  DWORD        cCount // number of points in array
);
```

The **Polyline** function draws a series of line segments by connecting the points in the specified array, as shown in the following code example. The lines are drawn from the first point through subsequent points by using the current pen. Unlike the **LineTo()** or **PolylineTo()** functions, the **Polyline()** function neither uses nor updates the current position.

```
BOOL Polyline(
  HDC          hdc,    // handle to device context
  CONST POINT *lppt,   // array of endpoints
  int          cPoints // number of points in array
);
```

The following X Windows example shows the use of **XDrawLine()**:

```
int main (int argc, char **argv)
{
  XtToolkitInitialize ();

  myApplication = XtCreateApplicationContext ();

  myDisplay     = XtOpenDisplay( myApplication,
                                 NULL,
                                 NULL,
                                 "XBlaat",
                                 NULL,
                                 0,
                                 &argc,
                                 argv);

  myWindow = RootWindowOfScreen(DefaultScreenOfDisplay (mydisplay));

  //+
  // now we need a surface to draw on
  //-
  myMap = XCreatePixmap ( myDisplay,myWindow,64,64, 1 );

  values.foreground =
    BlackPixel (myDisplay, DefaultScreen (myDisplay));

  myGC = XCreateGC (myDisplay, mySurface, GCForeground, &values);

  //+
  //  draw two diagonal lines across the 64x64 surface
  //
  XDrawLine( myDisplay,mySurface,myGC,0,0,63,63 );

  XDrawLine( myDisplay,mySurface,myGC,0,63,63,0 );

  …

}
```

The following Win32 example shows the use of **MoveToEx()** and **LineTo()**:

```
void lineExampleIn64X64Window( HDC hDC, HPEN myBlackPen )
{
  HPEN oldPen;

  //+
  //  use a black pen
  //-
  oldPen = (HPEN)SelectObject( hDC, myBlackPen );

  //+
  //  set the current position.
```

```
  //  this would not be necessary if it was
  //  known that the current position
  //  was already at (0,0).
  //-
  MoveToEx( hDC, 0,0, NULL );

  //+
  //  now the current position is at (63,63)
  //-
  LineTo( hDC, 63,63 );

  //+
  //  set the current position to the lower left corner
  //-
  MoveToEx( hDC, 0,63, NULL );

  //+
  //  draw the second diagonal
  //  and make the current position (63,0)
  //
  //-
  LineTo( hDC, 63, 0 );

  //+
  // put back the old pen
  //-
  SelectObject( hDC, oldPen );

}
```

Drawing rectangles

In Win32, Rectangle shape is a filled shape. Filled shapes are geometric forms that the current pen can outline and the current brush can fill.

There are five filled shapes:

- Ellipse
- Chord
- Pie
- Polygon
- Rectangle

In X Windows, the XRectangle shape is quite different from the Win32 equivalent. When porting between the two, it is necessary to understand the conceptual difference. The X Windows version uses an upper-left-corner point and the width and height. The Win32 version uses the upper left and lower right points. This difference is also true for the **XDrawRectangle()** and Win32 **Rectangle()** functions.

The X Windows structure is as follows:

```
typedef struct {
  short x,y;
  unsigned short width,height;
} XRectangle;
```

Its Win32 equivalent is as follows:

```
typedef struct _RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT;
```

The **Rectangle()** function draws a rectangle. The rectangle is outlined by using the current pen and filled by using the current brush. Because it does not fill the rectangle, this is quite different from the **XDrawRectangle()** function.

```
BOOL Rectangle(
  HDC hdc,          // handle to DC
  int nLeftRect,    // x-coord of upper-left corner of rectangle
  int nTopRect,     // y-coord of upper-left corner of rectangle
  int nRightRect,   // x-coord of lower-right corner of rectangle
  int nBottomRect   // y-coord of lower-right corner of rectangle
);
```

Rectangle functions that fill the rectangle are:

- X Windows: XFillRectangle()
- Win32: Rectangle()
- Win32: FillRect()

Rectangle functions that draw the outline only are:

- X Windows: XDrawRectangle()
- Win32: FrameRect()

Note The Win32 functions **Rectangle()** and **FillRect()** differ in the parameters they take. For additional information, see Visual Studio Help or the MSDN Web site, http://msdn.microsoft.com/.

The following X Windows example demonstrates rectangle functions:

```
void drawSomeRectangles()
{

  //+
  // fill the rectangle and then draw a black border around it
  //-
  XFillRectangle (myDisplay, mySurface, myWhiteGC, 0, 0, 31, 31);
  XDrawRectangle (myDisplay, mySurface, myBlackGC, 0, 0, 31, 31);


  //+
  //  draw an empty rectangle ten pixels square
  //-
  XDrawRectangle( myDislay, mySurface, myBlackGC, 0,0, 10,10 );
}
```

The following Win32 example demonstrates rectangle functions:

```
void drawSomeRectangles( )
```

```
{
  RECT myRectangle;

  //+
  // fill the rectangle and then draw a black border around it
  // assume that the current pen in this DC is black and the
  // current brush is not
  //-
  Rectangle( hDC, 0,0,31,31 );



  //+
  //  draw an empty rectangle ten pixels square
  //  The FrameRect() function draws a border around the specified
  //  rectangle by using the specified brush rather than the current
  //  pen.
  //
  //  The width and height of the border are always one logical unit.
  //-
  myRectangle.Top     = 0;
  myRectangle.Left    = 0;
  myRectangle.Bottom  = 10;
  myRectangle.Right   = 10;

    FrameRect( hDC, &myRectangle, myBlackBrush  );

}
```

Timeouts and Timers

Timers are required to determine and act on delays in user input. The functionality and differences between X Windows timeouts and Win32 timers is discussed in the following sections.

X Windows Timeouts

An X Windows client program can use the **XtAddAppTimeOut()** and **XtRemoveTimeOut()** functions with a callback to perform processing based an interval specified in milliseconds. The following code shows an example of this:

```
//+
// perform task every one second
//-
void myTimerProc( Widget    w,
                  XEvent   *event,
                  String   *pars,
                  Cardinal *npars)
{
  …
  …
}

//+
// start a 1 second timer
//-
void startTimer( XtIntervalId *timer )
{
```

```
  (*timer) = XtAppAddTimeOut( gContext, 1000, myTimerProc, NULL);
}

//+
// stop the timer
//-
void stopTimer( XtIntervalId * timer )
{
  if( *timer ) {
    XtRemoveTimeOut( *timer );
    (*timer) = NULL;
  }
}
```

Win32 Timers

Windows timers can be used in two scenarios. First, as in the X Windows approach, a callback function can be identified to execute at each timer interval. Second, a message (WM_TIMER) can be used to process timer intervals.

The following example shows the callback version using Win32 timers:

```
//+
// perform task every 1 second
//-
void CALLBACK myTimerProc( HWND      w,
                  UINT      timerMessage,
                  UINT_PTR  timerID,
                  DWORD     systemTime )
{
  …
  …
}

//+
// start a 1 second timer
//-
void startTimer( UINT_PTR *timer, UINT_PTR timerID )
{
  (*timer) = SetTimer( hParentWindow, timerID, 1000, myTimerProc );
}

//+
// stop the timer
//-
void stopTimer( UINT_PTR *timer )
{
  if( *timer ) {
    KillTimer( hParentWindow, (*timer) );
  }
}
```

The following example shows Win32 timers using WM_TIMER:

```
#define timer1 1
#define timer2 2

LRESULT CALLBACK WndProc( HWND hWnd,
```

```
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
{

  switch (message) {

  case WM_CREATE :

    //+
    //  create two timers
    //-
    SetTimer( hWnd, timer1, 1000, (TIMERPROC) NULL );
    SetTimer( hWnd, timer2, 5000, (TIMERPROC) NULL );
    break;

  //+
  //  wParam specifies that the timer identifier
  //  returns zero (0) if this message is handled
  //-
  case WM_TIMER

    switch ( wParam ) {

    case timer1 :

      // do timer 1 stuff

      break;

    case timer2 :

      // do timer 2 stuff

      break;

    default :
      break;
    }

    return ( FALSE );

  case WM_DESTROY :

    //+
    //  stop the two timers
    //-
    KillTimer( hWnd, timer1 );
    KillTimer( hWnd, timer2 );

    PostQuitMessage(0);
    break;

  …
```

Migrating Character-Based User Interfaces

Not all UNIX-based interfaces are graphical. Character-based interfaces were the original mainstay of UNIX computing long before the graphical workstation was conceived. There are two options for

character-based interfaces. Either a migration to the Interix environment can take place with minimal change, or a graphical interface (Windows-based or HTML) can replace the character-based interface. A preliminary port to POSIX smoothes the migration to Interix.

POSIX Terminal I/O

The POSIX termios structure and a new set of access calls replace the two traditional terminal hardware interfaces, namely termio structures in System V and stty structures in BSD.

The POSIX input/output (I/O) model is very similar to the System V model. Two modes exist: canonical and noncanonical. Canonical input is line-based, like BSD cooked mode. Noncanonical mode is character-based, like BSD raw or cbreak mode. The Interix subsystem includes a true, noncanonical mode, with support for **cc_c[VMIN**] and **cc_c[VTIME]**.

The **termios** structure is defined in Termios.h, as shown in the following listing:

```
struct termios {
   tcflag_t c_iflag;  /* input mode         */
   tcflag_t c_oflag;  /* output mode        */
   tcflag_t c_cflag;  /* control mode       */
   tcflag_t c_lflag;  /* local mode         */
   speed_t c_ispeed;  /* input speed        */
   speed_t c_ospeed;  /* output speed       */
   cc_t c_cc[NCCS];   /* control characters */
};
```

The Interix SDK extends the POSIX.1 set of flags for **c_iflag** to include IMAXBEL and VBELTIME. For **c_cc**, VMIN and VTIME do not have the same values as VEOF and VEOL. For a portable application, however, a developer should take into consideration that VMIN and VTIME can be identical to VEOF and VEOL on a POSIX.1 system.

Table 8 shows the 12 new functions that replace the terminal I/O **ioctl()** calls, which include **ioctl(fd, TIOCSETP, buf)** and **ioctl(fd, TIOCGETP, buf)** or **stty()** and **gtty()**. They were changed because the data type of the final argument for terminal I/O **ioctl()** calls depends on an action that makes type checking impossible.

Table 8. Functions that replace terminal I/O ioctl()

Function	Description
tcgetattr()	Fetches attributes (termios structure)
tcsetattr()	Sets attributes (termios structure)
cfgetispeed()	Gets input speed
cfgetospeed()	Gets output speed
cfsetispeed()	Sets input speed
cfsetospeed()	Sets output speed
tcdrain()	Waits for all output to be transmitted
tcflow()	Suspends transmit or receive
tcflush()	Flushes pending I/O
tcsendbreak()	Sends BREAK character
tcgetpgrp()	Gets foreground process group identifier (ID)
tcsetpgrp()	Sets foreground process group ID

To get the window size, use the **TIOCGWINSZ** command for **ioctl()** or the winsize structure, which are both supported.

Porting Curses and Terminal Routines to Interix

The Interix SDK libraries include Libcurses.a, a port of the ncurses package, and Libtermcap.a, termcap routines. The Interix SDK also supports pseudoterminals. Porting such applications to Interix should be straightforward, but note the following:

- The curses routines make use of the terminfo database, stored in /usr/share/terminfo. This location is different from the location used in traditional systems. To link with the terminfo routines, link with the curses library.
- Interix supports both the BSD /dev/ptynn and the System V /dev/ptmx methods for opening the master side of a pseudoterminal. The System V method is slightly faster because the search for an available master device is handled in the subsystem. Currently, the Interix subsystem supports 265 ptys named /dev/pty[p–zA–E][0–9a–f] on the master side. The corresponding subordinate side names are /dev/tty[p–zA–E][0–9a–f].
- When using /dev/ptmx, the subordinate (slave) tty name can be obtained with **ptsname()**. BSD-based **ioctl()** calls can be used with the pty master side.
- Provided that it is a session leader, a process without a controlling tty acquires a controlling terminal on the first **open()** call to a tty, unless NOCTTY is specified in the **open()** call.

Older character-based terminal applications placed the cursor on the physical display screen based on the capabilities of the terminal. These capabilities were typically stored in the /etc/termcap file with around 15,000 lines of terminal capabilities.

The Console APIs can be used to create character-based applications with an addressable cursor. **SetConsoleCursorPosition()**, **WriteConsole()**, and **ReadConsole()** are three functions among the many available for use in the Console API.

Porting OpenGL Applications

OpenGL was originally developed by Silicon Graphics as a platform-independent set of graphics APIs. This has made OpenGL an attractive option for developers who wish to target multiple platforms. Very little, if any, platform-specific code is necessary to move a graphics application from one platform to another. OpenGL extensions enable the segregation and handling of platform-specific code.

OpenGL is not, however, a set of windowing libraries. Most platform-specific code is encountered by using windows that in a UNIX to Windows migration. An OpenGL application with windows uses either the windowing system of the target platform (X Windows or Win32), or a cross-platform library such as the OpenGL Graphics Library Utility Kit (GLUT). Because of licensing concerns, however, most commercial applications incorporate the target platform windowing system. Therefore, when moving a UNIX application that uses OpenGL to Windows, migration considerations similar to those for a non-OpenGL application are likely to apply. The remainder of this section covers additional GUI considerations for the migration of OpenGL applications.

In addition to the windowing system itself, OpenGL applications require a "context" to the host windowing system. A special set of OpenGL extensions for window context has been developed. UNIX applications typically use the GLX OpenGL extensions for X Windows. Microsoft Windows-based applications typically use the WGL (wiggle) OpenGL extensions. In either case, these three main

functions are required:

- Create context.
  - X Windows: glXCreateContext
  - Win32: wglCreateContext
- Make context current.
  - X Windows: glXMakeCurrent
  - Win32: wglMakeCurrent
- Delete context.
  - X Windows: glXDeleteContext
  - Win32: wglDeleteContext

If, after the migration, the application still needs support for UNIX, C/C++ pre-compiler directives (**#ifdef**) can be used to target the appropriate platform.

The OpenGL API is a C library on both UNIX and Windows. FORTRAN applications can also use OpenGL. To make it easier for FORTRAN applications to use OpenGL, a FORTRAN 90 Module often exists to handle the translation between FORTRAN and C calling conventions. Most FORTRAN compilers on Windows provide an optional FORTRAN module for OpenGL.

For additional information about OpenGL and platform-specific examples, see the SGI OpenGL Web site or the opengl.org Web site.

GDI+

GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on both the video display and the printer. Win32-based applications do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications. GDI+ is also supported by the 64-bit Windows operating system.

To use GDI+, the developer must copy the Gdiplus.dll library to the system directory of the user's computer. For information about which operating systems are required to use a particular class or method, see the Requirements section of the documentation for the class or method.

All Windows-based applications can use GDI+, a new technology in the Windows XP and the Windows Server 2003 family of operating systems. It is available for applications that run on the Windows NT 4.0 SP6, Windows 2000, Windows 98, and Windows Millennium Edition operating systems.

To download the latest redistributable, see Platform SDK Redistributables.

Mapping X Windows Terminology to Microsoft Windows

The graphical models of UNIX and Microsoft Windows are very different. There are conceptual similarities, but little side-by-side mapping is possible. This section, however, describes as many connections as possible.

In the headings of this section, the Win32 GDI term is followed by the corresponding X Windows term in the following format:

```
X Windows term - Win32 term
```

Callback vs. WindowProc

Win32 uses the **WindowProc** function in the same capacity as **Callback** in X Windows. An X widget can have a list of callbacks associated with it, but a Win32 window has a single entry point for handling messages sent to it.

Client vs. Client Window

X Windows comprises a protocol that describes how a client interacts with a server that could be running on a remote computer. How objects are drawn is the responsibility of the server. This provides device independence for the client application because it is not responsible for knowing anything about the physical hardware.

In the Microsoft Windows environment, the graphics device interface (GDI) API provides this layer of device independence. Windows-based applications, like X clients, are not required to access graphics hardware directly. GDI interacts with the hardware by using device drivers on behalf of the application.

A single Windows-based application can contain any number of separate windows. Each of these can have a window frame, caption bar, system menu, minimize and maximize buttons, and its own main display area, which is referred to as the client window.

Windows MDI (multiple document interface) applications have three kinds of windows: a frame window, an MDI client window, and a number of child windows. The term "client window" takes on a special meaning in this case. For additional information on MDI, see the MDI documentation on the MSDN Web site or the Platform SDK.

Console Mode vs. Command Window

If X Windows or some other graphical user interface is not running on a UNIX system ,then a user must work in text only or in *console mode*.

Think of Microsoft Windows as exactly the opposite. If a console is not running, the user must work in GUI mode. Windows text-based mode is provided by running the **Cmd.exe** utility. This environment is also referred to as a command window or the MS-DOS prompt. To run the **Cmd.exe** utility, click **Start** and then click **Run**. Type **cmd** and then click **OK**. Or, on the keyboard, press the Windows key and then press**R**. The Run dialog box appears.

Developers can also use the Console API to build native Win32-based console applications. See Console Functions

DPI vs. Screen Resolution

When starting an X Windows session, using the **-dpi** (dots per inch) option can improve appearance on displays with larger resolutions, such as 1600x1200. The **-dpi** option also helps to work around possible font issues.

A Windows-based application is usually built with no assumptions about the capabilities of the system it will run on. System APIs are used to calculate proper scaling and other characteristics **GetDeviceCaps()** is used to obtain the DPI of the system. **GetSystemMetrics()** and **SystemParametersInfo()** provide information about practically every graphical element needed to calculate sizes for fonts and other

graphical elements. for more information, search for "dots per inch" on the MSDN Web site.

Graphics Context vs. Device Context

The X Windows graphics context (GC) contains required information about how drawing functions are to be executed. Win32 device context (DC) provides similar information. The functions used in each are summarized in Table 9.

Table 9. X Windows GC and Win32 DC Comparable Functions

Xlib	Win32
XtGetGC	**GetDC**
XtReleaseGC	**ReleaseDC**
XCreateGC	**CreateDC**
XFreeGC	**DeleteDC**

For more information about using DC and GC, see the sections under "Drawing" earlier in this chapter.

Resources vs. Properties

In X Windows terminology, a widget is defined by its *resources*. Width, height, color, and font are examples of resources. Resources can be managed by using the **XtVaCreateManagedWidget()** method, or by using resource files or **XtVaGetValues()** and **XtVaSetValues()**.

In Win32 terminology, a control is defined by its *properties*. For example, a text control has the following properties: Center Vertically, No Wrap, Transparent, Right Aligned Text, and Visible.

Resource Files vs. Registry

X Windows systems use configuration files referred to as *resource files* to store information about system settings or preferences for a particular X Windows client.

In a Windows-based system, this type of information is stored in the *registry*. The registry stores data in a hierarchically structured tree. There is a Win32 API with more than 40 functions to help access the registry. For more information, search for "registry" or "registry functions" on the MSDN Web site.

Resource file can take on another meaning in Windows-based application development. Resources are objects (like widgets), such as menus, dialog boxes, cursors, strings, bitmaps and user-defined data that are used in an application. Resource files have an extension of .rc and contain a special resource language (script) that is compiled by the resource compiler. The resulting file (.res) is then linked with the containing application.

For more information, search for "using resources" on the MSDN Web site.

Root Window vs. Desktop Window

All X Windows windows are descendents of the root window. In the Windows environment, the desktop window is a system-defined window that is the base for all windows displayed by all applications.

/bin vs. System32

In Windows, the /System32 directory is roughly equivalent to the /bin directory on a UNIX system. This is where the system executable files are located. The /System32 directory is located in the system root directory. To find system root, at a command prompt, type **set** and press **Return**. This displays a listing of the current environment. In the list, locate SYSTEMROOT. Under SYSTEMROOT, there is an entry similar to SYSTEMROOT=C:\WINNT. This is the system directory, and under that directory is the /System32 directory.

/usr/bin vs. Program Files

The Program Files directory on a Windows-based system is similar to the /usr/bin directory on a UNIX system. This is a default location for user applications. In Windows, a user can create more than one. Each drive, for example, might have a Program Files directory. The system environment variable ProgramFiles contains the path of one default location, for example, ProgramFiles=C:\ProgramFiles.

/usr/lib vs. LIB Environment Variable

In Windows, the path to user libraries can be anywhere. To manage this relationship, retrieve or set the system environment variable LIB.

/usr/include vs. INCLUDE Environment Variable

In Windows, the path to user include files may be anywhere. To manage this relationship, retrieve or set the system environment variable INCLUDE.

Pixmap or Bitmap vs. Bitmap

In X Windows, bitmap and pixmap have the same usage as Win32 bitmaps. For example, they can be used as pictures, fill patterns, icons, and cursors. They are, however, very different in form.

The following X Windows example represents a simple 16x16 "X" figure:

```
#define simple_width 16
#define simple_height 16
static unsigned char simple_bits[] = {
  0x01, 0x80, 0x02, 0x04, 0x20, 0x08, 0x10, 0x10, 0x08, 0x20, 0x04,
  0x40, 0x02, 0x80, 0x01, 0x80, 0x01, 0x02, 0x20, 0x04, 0x10, 0x08,
  0x08, 0x10, 0x04, 0x20, 0x02, 0x40, 0x01, 0x80
};
```

The following Win32 example also represents a simple 16x16 "X" figure:

```
000000  42  4D  7E  00  00  00 00  00  00  00  3E  00  00  00  28  00
000010  00  00  10  00  00  00  10  00 00  00  01  00  01  00  00  00
000020  00  00  40  00  00  00  CA  0E  00  00  C4  0E  00  00  00 00
000030  00  00  00  00  00  00  00  00  00  00  FF  FF  FF  00  7F  FE
000040  00  00  BF  FD  00  00  DF  FB  00  00  EF  F7  00  00  F7 EF
000050  00  00  FB  DF  00  00  FD  BF  00  00  FE  7F  00  00  FE 7F
000060  00  00  FD  BF  00  00  FB  DF  00  00  F7  EF  00  00  EF F7
000070  00  00  DF  FB  00  00  BF  FD  00  00  7F  FE  00  00
```

Window Manager vs. Windows 2000 and Windows XP

A special kind of X Windows client called the Window Manager provides a consistent working environment in the root window.

In a Microsoft Windows environment, the operating system itself is the window manager and it provides the desktop window. When a user logs on, the system creates three desktops within the WinSta0 windows station. For more information, search for "WinSta0" on the MSDN Web site. Widgets or Gadgets—Controls

Widgets are usually represented as controls in Win32-based applications. Like the X Windows environment, Win32 offers many to choose from and there are a great number of third-party versions available.

Sometimes deciding exactly what to call which is difficult. For example, X Windows dialog boxes are widgets. In Win32, however, dialog boxes are not considered to be controls, although objects such as dialog boxes, buttons, scroll bars and so on, are all windows.

X Library [Xlib] [X11] vs. Gdi32.lib

The X Windows library [Xlib][X11] is the lowest level library. Like Gdi32.lib, it provides all the basic drawing functions.

X Toolkit [Intrinsics] [Xt] vs. User32.lib

The X Toolkit (Xt) is a library that accesses the lower-level graphics functionality of Xlib (X Windows) and provides user interface elements such as menus, buttons, and scroll bars. It is similar to User32.lib except that in the Win32 environment the look and feel of widgets or controls is provided in User32.lib rather than by higher-level libraries.

Mapping X Windows Tools to Microsoft Windows

The primary tools for Win32 development are the Microsoft Platform SDK and Microsoft Visual Studio. If the Microsoft Platform SDK is installed in the default location, tools are found in *DriveLetter*:\Program Files\Microsoft Platform SDK\Bin. Also look for tools in the SDK Help or open Tools Help in *DriveLetter*:\Program Files\Microsoft Platform SDK\Help directory. SDK Help and Tools Help provide descriptions of all the SDK tools. In Microsoft Visual Studio, tools are found in *DriveLetter*:\Program Files\Microsoft Visual Studio\Common\Tools. Online Help provides information on these tools.

> **Note** In some cases, the X Windows tool and the Win32 tool have the same name but do not perform the same function. The **bitmap** tool is one example of this.

Bitmap vs. Imagedit.exe and Shed.exe

The **Imagedit.exe** tool in the Platform SDK supports the same functionality as **bitmap**. In Visual Studio, use the resource editor to create and edit bitmaps and icons. In MSDN, look for resource editors under Visual Tools and Languages\Visual Studio 6.0 Documentation\Visual C++ Documentation\Using Visual C++\Visual C++ User's Guide\Resource Editors.

Shed.exe is a Win32 tool included in the Platform SDK. It is used to edit hotspots. It is found in

DriveLetter:\Program Files\Microsoft Platform SDK\Bin\WinNT. This Hotspot Editor is used to create and edit hypergraphics. A hypergraphic is a bitmap that includes one or more hotspots.

Mspaint.exe can also be used to edit bitmaps. This utility is found in the /System32 directory.

Kodakimg.exe can be used to open, view, and edit a large variety of graphic files. This utility is found in *DriveLetter*:\Program File\Windows NT\Accessories\ImageVue.

Manual Pages vs. Help

UNIX provides online documentation, which explains commands and procedures, in the form of *manual pages*. To access a particular manual page, at the shell prompt, type **man command_name**.

Windows systems use the commands **help** and **help** *CommandName*. These provide a similar look and feel to man on UNIX systems. However, most of Windows Help is found on the Start Menu under Help. Additionally most if not all of the Microsoft development environment (MSDN, compilers, Visual Studio, WORD) provide topical help.

At a command prompt, type **help** and press **Return** to see a list of available commands. Typing **help** followed by the name of the command will provide information about the specified command. For example, **help setlocal**.

In the system directory you will find a help directory. Here you will find compiled HTML format help on all aspects of the Windows environment. Windows Advanced Server installations provide ntbooks.exe in the system32 directory. This is an excellent help resource for all windows server commands.

grep vs. Qgrep.exe

The **Qgrep** utility can be found in the Platform SDK/Bin directory. It performs much like the UNIX **grep** family of commands.

xcalc vs. Calc.exe

The **Calc.exe** utility is the Windows calculator program. It is located in the /System32 directory and provides number base conversion between decimal, hexadecimal, and binary.

xclipboard vs. Clipbrd.exe

The **Clipbrd.exe** utility is found in the /System32 directory and provides the Win32 Clipboard viewing, sharing, and saving functions.

xedit vs. Notepad.exe

The **Notepad.exe** utility is a simple text editor that can be used like **xedit**. **Notepad** is located in the system root directory.

xev vs. Spy.exe or Spyxx.exe

The **Spy.exe** and **Spyxx.exe** utilities provide functionality like **xev**. These utilities allow selection of a window and filtering of desired events and messages. The **Spy.exe** utility is provided in the Platform

SDK and **Spyxx.exe** (also called **spy++**) is provided in Visual Studio.

xfd vs. Fontview.exe

The **Fontview.exe** command-line utility provides a view of fonts. For example, the following command displays the Modern fonts:

```
fontview modern.fon
```

However, the **Charmap.exe** utility, a GUI utility found in the /System32 directory, is a much better choice for viewing and manipulating fonts in a graphical manner.

xkill vs. Kill.exe

The Win32 **Kill.exe** utility provides the same functionality as the X Windows **xkill** command. The **Kill.exe** utility is found in the /System32 directory.

When a user presses **Ctrl+Alt+Del** on a Windows-based system, a dialog box appears. Click the **Task Manager** button to display the **Task Manager** dialog box. To display the (PID) process ID of the current running tasks, click the **Processes** tab in the **Task Manager** dialog box. Locate the errant process in the list and use that PID in the **kill** command, or simply click the process and then click **End Process**.

On a machine withPlatform SDK is installed, the user can track down and kill a troublesome process by using the **Pview.exe** utility. This utility is found in the directory /Program Files/Microsoft Platform SDK/Bin/WinNT.

xlsclients vs. Pview.exe

Like **xlsclients**, the **Pview.exe** utility lists the current running applications. **Pview** is a GUI application that can be used to select the name of a computer to view.

xlsfonts vs. Fonts Control Panel Item

The Windows Control Panel Fonts item provides all font management functionality. For additional information, see Fonts Help, in the /System/Help Fonts.chm file.

xmag vs. Magnify.exe or Zoomin.exe

The Win32 **Magnify.exe** utility is equivalent to X Windows **xmag**. **Magnify** is found in the /System32 directory on Windows 2000 and Windows XP.

On a machine with Microsoft Visual Studio or the Platform SDK installed, the user also has access to the **Zoomin.exe** utility, which is found in Visual Studio at *DriveLetter*:\Program Files\Microsoft Visual Studio\Common\Tools or in the Platform SDK at *DriveLetter*:\Program Files\Microsoft Platform SDK\Bin.

xon vs. Start.exe or Remote.exe

Like the X Windows **xon** command, the Windows **Start.exe** utility starts a new command window to run a specified program or command.

The Platform SDK provides the **Remote.exe** utility. **Remote** is a debugging utility, but proves to be useful for much more than debugging. Use this application to start a server end and a client end, which allows commands to be executed on a remote system. The **Remote** utility is found at *DriveLetter*:/Program Files/Microsoft Platform SDK/Bin/Dbg.

xset client vs. Control Panel Items

Microsoft Windows provides a GUI interface for managing the keyboard, the mouse, and the video display. The Control Panel includes an item for managing each of these devices.

The **Mode**, **Color**, and **Graftabl** commands can be used to perform some device management. To see a list of features for these three commands, at a command prompt, type **help mode**, **help color**, or **help graftabl**.

If a particular application requires advanced device control, that application must provide code to perform this required functionality. Use the **SystemParametersInfo()** function to set or retrieve systemwide parameters.

xterm vs. Hyperterm.exe

The Windows utility **Hyperterm.exe**, found in the /Program Files/Windows NT directory, is a terminal emulator similar to the X Windows **xterm** command.

User Interface Coding Examples

The examples in this section show how to port an X Windows-based application to Microsoft Windows:

- X Windows "Hello World" example (including xHello.mak)
- Win32 "Hello World" example
- Win32 DialogWindow example

X Windows "Hello World" Example

The following example demonstrates the Hello World code for X Windows.

```
/*
** xHello.c
**
** One possible "Hello World" according to X11
**
*/
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Intrinsic.h>

char helloString[] = "Hello World";

int main( int argc, char **argv )
{
  int           iScreen;
  unsigned long ulForeground;
  unsigned long ulBackground;
```

```
  Display      *pDisplay;
  Window        exampleWindow;
  GC            gc;
  XSizeHints    sizeHints;
  XWMHints      wmHints;
  XTextProperty textProperty;
  XEvent        xEvent;

  pDisplay         = XOpenDisplay( NULL );
  iScreen          = DefaultScreen( pDisplay );

  ulBackground     = WhitePixel( pDisplay, iScreen );
  ulForeground     = BlackPixel( pDisplay, iScreen );

  sizeHints.x      = 0;
  sizeHints.y      = 0;
  sizeHints.width  = 250;
  sizeHints.height = 30;
  sizeHints.flags  = ( PPosition | PSize );

  wmHints.flags    = InputHint;
  wmHints.input    = True;

  exampleWindow    = XCreateSimpleWindow( pDisplay,
                                          DefaultRootWindow( pDisplay
),
                                          sizeHints.x,
                                          sizeHints.y,
                                          sizeHints.width,
                                          sizeHints.height,
                                          2,
                                          ulForeground,
                                          ulBackground );


  XStringListToTextProperty( &argv[0],1,&textProperty );

  XSetWMName( pDisplay, exampleWindow ,&textProperty );

  XSetWMProperties( pDisplay,
                    exampleWindow,
                    &textProperty,
                    NULL,
                    NULL,
                    0,
                    &sizeHints,
                    &wmHints,
                    NULL );

  gc = XCreateGC( pDisplay, exampleWindow, 0,0 );

  XSetBackground( pDisplay, gc, ulBackground );
  XSetForeground( pDisplay, gc, ulForeground );

  XSelectInput( pDisplay, exampleWindow, ( KeyPressMask |
    ExposureMask ) );

  XMapWindow( pDisplay, exampleWindow );

  do {
```

```
    XNextEvent( pDisplay, &xEvent );

    if ( xEvent.type == Expose ) {

      if ( xEvent.xexpose.count == 0 ) {

        XClearWindow( pDisplay, exampleWindow );

        XDrawImageString( pDisplay,
                          exampleWindow,
                          gc,
                          (sizeHints.width/10),
                          (sizeHints.height/2),
                          helloString,
                          (strlen( helloString )));

      }

    }

  } while (1);

  exit( 0 ) ;

}
```

The xHello.mak file

The following example shows the X Windows xHello.mak file used with the previous Hello World code.

```
CC = cc
INSTALL = ./
INCLUDES = -I/usr/X11R6/include
LIBS = -L/usr/X11R6/lib -lX11 -lXaw -lXt -lXext
OBJS = xHello.o
xHello : ${OBJS}

  ${CC} -o xHello ${OBJS} ${INCLUDES} ${LIBS}

clean:
  rm -fr *.o xHello
```

Win32 "Hello World" Example

The following example demonstrates the corresponding Hello World code for Win32.

```
#include <windows.h>   // basic windows functionality
#include <tchar.h>     // required for _tcsclen() in WndProc

//+
//  type TCHAR
//
//  If the symbol _UNICODE is defined for your program,
//    TCHAR is defined as type wchar_t, a 16-bit character type.
//  Otherwise,
//    TCHAR is defined as char, the normal 8-bit character type.
//
//
```

```
//  If you have MSDN installed, look for these and other
//  papers and help files about UNICODE:
//
//    "International Support in Microsoft Windows 2000"
//    "Defining a Character Set"
//    "Unicode Programming Summary"
//    "Unicode and Character Sets"
//
//
//  The TEXT macro identifies a string as
//    Unicode when UNICODE or _UNICODE is defined during compilation.
//  Otherwise, it identifies a string as an ANSI string.
//
//-
TCHAR     *szAppName     = TEXT("HelloWorld");
TCHAR     *szTitle       = TEXT("Win32 - Hello World");
TCHAR     *szMessage     = TEXT("Hello World!");


HINSTANCE  hInst; // A sometimes useful global copy of our instance
handle

//+
//
//  The WindowProc function is an application-defined
//  function that processes messages sent to a window.
//
//  The WNDPROC type defines a pointer to this callback function.
//  WindowProc is a placeholder for the application-defined function
name.
//
//  *** In the WinMain() function
//  *** notice this line of code   --> wc.lpfnWndProc =
  (WNDPROC)WndProc;
//
//-
LRESULT CALLBACK WndProc( HWND hWnd,
                          UINT message,
                          WPARAM uParam,
                          LPARAM lParam )
{
  //+
  //  A device context is a structure that defines a set of graphic
objects
  //  and their associated attributes, as well as the
  //  graphic modes that affect output.
  //
  //  The graphic objects include:
  //    a pen for line drawing,
  //    a brush for painting and filling,
  //    a bitmap for copying or scrolling parts of the screen,
  //    a palette for defining the set of available colors,
  //    a region for clipping and other operations,
  //    a path for painting and drawing operations.
  //-
  HDC hDC; // Handle to the device context, such as the GC in an X11
app

  //+
  //  The PAINTSTRUCT structure contains information for an
application.
```

```
  //  This information can be used to paint the client area of a
  //  window owned by that application.
  //
  //
  //  typedef struct tagPAINTSTRUCT {
  //    HDC  hdc;
  //    BOOL fErase;
  //    RECT rcPaint;
  //    BOOL fRestore;
  //    BOOL fIncUpdate;
  //    BYTE rgbReserved[32];
  //  } PAINTSTRUCT, *PPAINTSTRUCT;
  //
  //-
  PAINTSTRUCT    ps;

  //+
  //  The RECT structure defines the coordinates of the
  //  upper-left and lower-right corners of a rectangle.
  //
  //  typedef struct _RECT {
  //    LONG left;
  //    LONG top;
  //    LONG right;
  //    LONG bottom;
  //  } RECT, *PRECT;
  //-
  RECT           rect;

  //+
  //   This switch is used to catch and respond to messages sent to
    this window
  //
  //   An X Windows program using XNextEvent() might use a do { ... }
    while(1); loop
  //   to process events retrieved by XNextEvent().
  //
  //   A Win32 program uses this function WndProc() and
  //   this sort of switch( message ){ .. }
  //   architecture to asynchronously  process "window Messages"
  //
  //-
  switch (message) {

  //+
  //
  //    The WM_PAINT message is sent when the system or another
  //    application makes a request to paint a portion
  //    of an application's window.
  //
  //    The message is sent when the UpdateWindow() or
  //    RedrawWindow() function is called, or by the
DispatchMessage() function
  //    when the application obtains a WM_PAINT message
  //    by using the GetMessage() or PeekMessage() function.
  //
  //-
  case WM_PAINT:

    //+
    //  The BeginPaint function prepares the specified window for
```

```
painting
    //  and fills a PAINTSTRUCT structure with information about the
painting.
    //-
    hDC = BeginPaint ( hWnd, &ps );

    //+
    //  NOTE ***
    //
    //  More complex applications could/should call a painting
      function here
    //  passing the HDC (hDC) or the PAINTSTRUCT (ps) as parameters.
    //  The HDC (hDC) is a member of the PAINTSTRUCT (ps)
    //-
    //+
    //  The GetClientRect function retrieves the coordinates of a
    //  window's client area. The client coordinates specify the
    //  upper-left and lower-right corners of the client area.
    //
    //  Because client coordinates are relative to the
    //  upper-left corner of a window's client area,
    //  the coordinates of the upper-left corner are (0,0).
    //-
    GetClientRect( hWnd, &rect );

    //+
    // The DrawText function draws formatted text in the specified
rectangle.
    //
    // It formats the text according to the specified method
    //  (expanding tabs, justifying characters, breaking lines, and
so forth).
    //
    //-
    DrawText( hDC,
              szMessage,
              _tcsclen(szMessage)  ,
              &rect,
              DT_CENTER | DT_VCENTER | DT_SINGLELINE );

    //+
    // The EndPaint function marks the end of painting in the
specified window.
    //
    // This function is required for each call to the BeginPaint
function,
    //   but only after painting is complete.
    //
    EndPaint( hWnd, &ps);

    break;

  //+
  //  The WM_DESTROY message is sent when a window is being
destroyed.
  //
  //  It is sent to the window procedure of the window being
destroyed
  //  after the window is removed from the screen.
  //
  //  This message is sent first to the window being destroyed and
```

```
then
  //  to the child windows (if any) as they are destroyed.
  //
  //  During the processing of the message,
  //  it can be assumed that all child windows still exist.
  //-
  case WM_DESTROY:

    //+
    //  The PostQuitMessage function indicates to the system that a
    //  thread has made a request to terminate (quit).
    //  It is typically used in response to a WM_DESTROY message
    //-
    PostQuitMessage(0);
    break;

  //+
  //  The WM_CLOSE message is sent as a signal that a
  //  window or an application should terminate
  //-
  case WM_CLOSE:


    //+
    //   The DestroyWindow function destroys the specified window.
    //   The function sends WM_DESTROY and WM_NCDESTROY messages to
    //   the window to deactivate it and remove the keyboard focus
from it.
    //
    //   The function also destroys the window's menu,
    //   flushes the thread message queue,
    //   destroys timers, removes clipboard ownership,
    //   and breaks the clipboard viewer chain
    //   (if the window is at the top of the viewer chain).
    //
    //   If the specified window is a parent or owner window,
    //   DestroyWindow automatically destroys the associated
    //   child or owned windows
    //
    //   when it destroys the parent or owner window.
    //
    //   The function first destroys child or owned windows,
    //   and then it destroys the parent or owner window.
    //-
    DestroyWindow (hWnd);
    break;

  //+
  // Pass message on if unprocessed
  //-
  default:

    //+
    //   The DefWindowProc function calls the default window
procedure
    //   to provide default processing for any window messages
    //   that an application does not process.
    //
    //   This function ensures that every message is processed.
    //   DefWindowProc is called with the same parameters
    //   received by the window procedure
```

```
    //-
    return (DefWindowProc(hWnd, message, uParam, lParam));

  }

  return (0);
}

//+
//  The WinMain function is called by the system as the
//  initial entry point for a Windows-based application.
//
//  Your WinMain should
//
//   1. initialize the application,
//
//   2. display its main window,
//
//   3. enter a message retrieval-and-dispatch loop that is the top-
level
//      control structure for the remainder of the application's
execution.
//-
int APIENTRY WinMain( HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine,
                      int nCmdShow)
{

  HWND      hWnd;


  //+
  //  The MSG structure contains message information
  //  from a thread's message queue.
  //
  //  typedef struct tagMSG {
  //    HWND   hwnd;
  //    UINT   message;
  //    WPARAM wParam;
  //    LPARAM lParam;
  //    DWORD  time;
  //    POINT  pt;
  //  } MSG, *PMSG;
  //
  //-
  MSG msg;

  //+
  //  The WNDCLASS structure contains the window class attributes
that
  //  are registered by the RegisterClass function.
  //
  //  This structure has been superseded by the WNDCLASSEX structure
used
  //  with the RegisterClassEx function.
  //
  //  You can still use WNDCLASS and RegisterClass if you do not need
to set
  //  the small icon associated with the window class.
  //
```

```
  //  typedef struct _WNDCLASS {
  //    UINT       style;
  //    WNDPROC    lpfnWndProc;
  //    int        cbClsExtra;
  //    int        cbWndExtra;
  //    HINSTANCE  hInstance;
  //    HICON      hIcon;
  //    HCURSOR    hCursor;
  //    HBRUSH     hbrBackground;
  //    LPCTSTR    lpszMenuName;
  //    LPCTSTR    lpszClassName;
  //  } WNDCLASS, *PWNDCLASS;
  //
  //-
  WNDCLASS  wc;

  hInst = hInstance; // Store instance handle in our global variable

  //+
  //   1. Initialize
  //
  //       register the window class (required)
  //       Allocate global resources
  //       start threads
  //       connect to hardware etc.
  //  register the window class for this application
  //-
  wc.style         = CS_HREDRAW | CS_VREDRAW;          // Class
style(s).
  wc.lpfnWndProc   = (WNDPROC)WndProc;                 // Window
Procedure
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;

  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wc.lpszMenuName  = NULL;
  wc.lpszClassName = szAppName;
  wc.hIcon         = NULL;

  //+
  //  The RegisterClass function registers a window class
  //  for subsequent use in calls to the CreateWindow or
  //  CreateWindowEx function.
  //
  //  Note  The RegisterClass function has been
  //        superseded by the RegisterClassEx function.
  //        You can still use RegisterClass, however, if you do
  //        not need to set the class small icon.
  //-
  RegisterClass(&wc);

  //+
  //
  //  2. Display the Main Window
  //
  //      a. CreateWindow(), CreateWindowEx() or CreateDialog()
(required)
  //      b. return false on failure
(required)
```

```
  //      c. ShowWindow();   based on program logic
  //      d. UpdateWindow();
  //
  //-
  hWnd = CreateWindowEx(0,
                        szAppName,
                        szTitle,
                        WS_OVERLAPPEDWINDOW,
                        0,
                        0,
                        250,
                        250,
                        NULL,
                        NULL,
                        hInstance,
                        NULL
                    );

  if (!hWnd)
    return( FALSE );

  //+
  //  The ShowWindow() function sets the specified window's show
state.
  //-
  ShowWindow( hWnd, nCmdShow);

  //+
  //  The UpdateWindow() function updates the client area
  //  of the specified window by sending a WM_PAINT message to
  //  the window if the window's update region is not empty.
  //
  //  The function sends a WM_PAINT message directly to
  //  the window procedure of the specified window,
  //  bypassing the application queue.
  //
  //  If the update region is empty, no message is sent.
  //+
  UpdateWindow( hWnd );

  //+
  //
  //   3. Enter the message loop
  //
  //   The GetMessage() function retrieves a message
  //   from the calling thread's message queue.
  //
  //   The function dispatches incoming sent messages
  //   until a posted message is available for retrieval.
  //-
  while ( GetMessage( &msg, NULL, 0, 0 ) ) {

    //+
    //   The TranslateMessage() function translates
    //   virtual-key messages into character messages.
    //
    //   The character messages are posted to the calling
    //   thread's message queue, to be read the next time
    //   the thread calls the GetMessage() or PeekMessage() function.
    //-
    TranslateMessage( &msg );
```

```
    //+
    //   The DispatchMessage() function dispatches a message
    //   to a window procedure. It is typically used to
    //   dispatch a message retrieved by the GetMessage() function.
    //-
    DispatchMessage( &msg );
  }

  return (msg.wParam);

  lpCmdLine; // TRICK ***
             //
             // keep the compiler from complaining about unreferenced
parameter
             //

}
```

Win32 DialogWindow Example

The code in this section is based entirely on a single modeless dialog box. This type of implementation is very quick to put together in Visual Studio by using the resource editor. This example demonstrates how to communicate with many of the typical controls found in a Win32-based application.

```
// resource.h
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by DialogWindow.rc
//
#define EXAMPLE_DIALOG                  103
#define IDC_EXIT                        1000
#define IDC_RADIO1                      1001
#define IDC_DESTROY_PLANET              1001
#define IDC_RADIO2                      1002
#define IDC_DESTROY_STAR                1002
#define IDC_CHECK1                      1003
#define IDC_CHECK2                      1004
#define IDC_AUTHORIZATION_CODE          1005
#define IDC_TARGET_TEXT                 1006
#define IDC_ATTACK                      1007
#define IDC_TARGET_LIST                 1008
#define IDC_MISSION_VALUE               1009
#define IDC_TARGET_VALUE                1010
#define IDC_M_P                         1011

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        106
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1012
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif

// DialogWindow.rc
//
```

```
// Notice that EXAMPLE_DIALOG is the identifier of this template and
// is used in the call to CreateDialog(
  ,MAKEINTRESOURCE(EXAMPLE_DIALOG) ,, )
//
              #include "resource.h"
              #define APSTUDIO_READONLY_SYMBOLS
              #include "afxres.h"
              #undef APSTUDIO_READONLY_SYMBOLS

EXAMPLE_DIALOG DIALOG DISCARDABLE  0, 0, 267, 161
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "  Mission Setup "
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON      "&Quit",IDC_EXIT,210,135,50,15
    GROUPBOX        "  Select Mission ",IDC_STATIC,5,5,100,55
    CONTROL         "Destroy Planet",IDC_DESTROY_PLANET,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP,10,20,62,10
    CONTROL         "Destroy Star",IDC_DESTROY_STAR,"Button",
                     BS_AUTORADIOBUTTON,10,35,55,10
    EDITTEXT
IDC_AUTHORIZATION_CODE,190,85,60,14,ES_AUTOHSCROLL
    LTEXT           "Authorization Code
:",IDC_TARGET_TEXT,120,85,65,13
    PUSHBUTTON      "&Attack",IDC_ATTACK,155,135,50,14
    LISTBOX         IDC_TARGET_LIST,5,90,100,60,LBS_NOINTEGRALHEIGHT
|
                    WS_VSCROLL | WS_TABSTOP
    LTEXT           "Select Target",IDC_STATIC,5,75,100,15
    GROUPBOX        " Selected Mission Parameters
      ",IDC_STATIC,115,10,145,
                    110
    LTEXT           "Mission :",IDC_STATIC,120,35,60,8
    LTEXT           "Target :",IDC_STATIC,120,60,60,8
    EDITTEXT        IDC_MISSION_VALUE,190,35,60,14,ES_AUTOHSCROLL |
                    ES_READONLY
    EDITTEXT        IDC_TARGET_VALUE,190,60,60,14,ES_AUTOHSCROLL |
                    ES_READONLY
END

// DialogWindow.c
//
#include <windows.h>
#include "resource.h"

/*
** Global Variables
*/
HWND            hExampleDlg = 0;   /* Handle to modeless dialog box
*/
HANDLE          hInst;

//+
//  Static strings used in list boxes
//-
TCHAR *PlanetList[] = {
  {TEXT("Mercury")},
  {TEXT("Venus  ")},
  {TEXT("Earth  ")},
  {TEXT("Mars   ")},
  {TEXT("Jupiter")},
```

```
  {TEXT("Saturn ")},
  {TEXT("Neptune")},
  {TEXT("Uranus ")},
  {TEXT("Pluto  ")},
  NULL
};


TCHAR *StarList[] = {
  {TEXT("Andromeda       ")},
  {TEXT("51 Pegasi       ")},
  {TEXT("70 Virginis     ")},
  {TEXT("47 Ursae Majoris")},
  {TEXT("Beta Pictoris   ")},
  {TEXT("G1229           ")},
  {TEXT("Rho 1 Cancri    ")},
  NULL
};

//+
//  The DialogProc function is an application-defined callback
//  function used with the CreateDialog function.
//
//  It processes messages sent to the dialog box.
//
//  The DLGPROC type defines a pointer to this callback function.
//  DialogProc is a placeholder for the application-defined function
name.
//
//-
BOOL APIENTRY ExampleDlgProc( HWND hDlg,unsigned iMessage, WORD
  wParam,LONG lParam )
{
  int     x;
  LRESULT currentTarget;

  switch ( iMessage ) {

  //+
  //  The WM_INITDIALOG message is sent to the dialog box
  //  procedure immediately before a dialog box is displayed.
  //
  //  Dialog box procedures typically use this message
  //  to initialize controls and carry out any other initialization
  //  tasks that affect the appearance of the dialog box.
  //-
  case WM_INITDIALOG  :

    //+
    //  The CheckRadioButton function adds a check mark to
    //  the specified radio button in a group and removes a check
mark from
    //  all other radio buttons in the group.
    //
    //  If DialogWindow.rc is opened as a text file, the following
lines
    //  can be found.
    //
    //  GROUPBOX        "  Select Mission ",IDC_STATIC,5,5,100,55
    //  CONTROL         "Destroy Planet",IDC_DESTROY_PLANET,"Button",
    //                  BS_AUTORADIOBUTTON | WS_GROUP,10,20,62,10
```

```
    //  CONTROL         "Destroy Star",IDC_DESTROY_STAR,"Button",
    //                  BS_AUTORADIOBUTTON,10,35,55,10
    //
    //  The GROUPBOX contains the two BS_AUTORADIOBUTTON controls.
    //  This containment allows the radio buttons to be grouped
together
    //-
    CheckRadioButton( hDlg,
                      IDC_DESTROY_PLANET,
                      IDC_DESTROY_STAR,
                      IDC_DESTROY_PLANET );


    //+
    //  SendDlgItemMessage() sends a message to the designated
control.
    //  In this case it is a listbox control and the LB_INSERTSTRING
message
    //  is used to load a string in the list box.
    //
    //  For details about this message/command look up
      LB_INSERTSTRING
    //  in MSDN or Visual Studio Help
    //-
    for (x=0;PlanetList[x] != NULL;x++ )
      SendDlgItemMessage( hDlg,
                          IDC_TARGET_LIST,
                          LB_INSERTSTRING,
                          x,
                          (LPARAM)PlanetList[x] );

    //+
    //  An application sends an LB_SETCURSEL message to select a
string
    //  and scroll it into view, if necessary.
    //
    //  When the new string is selected, the list box removes
    //  the highlight from the previously selected string.
    //-
    SendDlgItemMessage( hDlg,IDC_TARGET_LIST,LB_SETCURSEL,0,(LPARAM)0
);

    //+
    //  The SetDlgItemText() function sets
    //  the title or text of a control in a dialog box.
    //-
    SetDlgItemText( hDlg, IDC_TARGET_VALUE, PlanetList[0] );
    SetDlgItemText( hDlg, IDC_MISSION_VALUE, TEXT(" Destroy Planet")
);
    SetDlgItemText( hDlg, IDC_AUTHORIZATION_CODE, TEXT("JD12131950")
);

    break;

  //+
  //   The WM_COMMAND message is sent when the user selects
  //   a command item from a menu, when a control sends a
  //   notification message to its parent window,
  //   or when an accelerator keystroke is translated.
  //
  //   In this example case
```

```
  //-
  case WM_COMMAND:

    switch ( wParam ) {

    //+
    //  this message is received each time the user clicks
    //  the IDC_DESTROY_PLANET radio button
    //-
    case IDC_DESTROY_PLANET :
      currentTarget = 0;
      SendDlgItemMessage( hDlg,
                          IDC_TARGET_LIST,
                          LB_RESETCONTENT,
                          (WPARAM)0,
                          (LPARAM)0);

      for (x=0;PlanetList[x] != NULL;x++ )
        SendDlgItemMessage( hDlg,
                            IDC_TARGET_LIST,
                            LB_INSERTSTRING,
                            x,
                           (LPARAM)PlanetList[x] );


      SetDlgItemText( hDlg, IDC_MISSION_VALUE, TEXT(" Destroy
        Planet") );
      SetDlgItemText( hDlg, IDC_TARGET_VALUE,
PlanetList[currentTarget] );
      SendDlgItemMessage( hDlg,
                          IDC_TARGET_LIST,
                          LB_SETCURSEL,
                          currentTarget,(LPARAM)0 );

      break;

    //+
    //  this message is received each time a user clicks
    //  the IDC_DESTROY_STAR radio button
    //-
    case IDC_DESTROY_STAR :

      currentTarget = 0;
      SendDlgItemMessage( hDlg,
                          IDC_TARGET_LIST,
                          LB_RESETCONTENT,
                          (WPARAM)0,
                          (LPARAM)0);


      for (x=0;StarList[x] != NULL;x++ )
        SendDlgItemMessage( hDlg,
                            IDC_TARGET_LIST,
                            LB_INSERTSTRING,
                            x,
                           (LPARAM)StarList[x] );

      SetDlgItemText( hDlg, IDC_MISSION_VALUE, TEXT(" Destroy Star")
);
      SetDlgItemText( hDlg, IDC_TARGET_VALUE,
StarList[currentTarget] );
```

```
      SendDlgItemMessage( hDlg,
                          IDC_TARGET_LIST,
                          LB_SETCURSEL,
                          currentTarget,
                          (LPARAM)0 );

      break;



    //+
    //  make sure a target was picked.
    //  this message will be sent if the user
    //  clicks anywhere in the list box
    //-
    case IDC_TARGET_LIST :


      //+
      //  Send an LB_GETCURSEL message to retrieve the index of the
currently
      //  selected item, if any, in a single-selection list box.
      //-
      currentTarget = SendDlgItemMessage( hDlg,
                                          IDC_TARGET_LIST,
                                          LB_GETCURSEL,
                                          0,
                                          0 );

      if (

        SendDlgItemMessage( hDlg,
                            IDC_DESTROY_PLANET,
                            BM_GETCHECK, 0,0 ) == BST_CHECKED ) {

        SetDlgItemText( hDlg, IDC_TARGET_VALUE,
          PlanetList[currentTarget] );
      }
      else {
        SetDlgItemText( hDlg, IDC_TARGET_VALUE,
          StarList[currentTarget] );
      }

      break;


    //+
    //  this message is received when a user clicks the IDC_ATTACK
button
    //-
    case IDC_ATTACK :
    {
        TCHAR   authorizationCode[255];
        TCHAR   missionString[255];
        TCHAR   targetString[255];
        TCHAR   executionString[255];

        //+
        //  In a single-selection list box,
        //  the return value is the zero-based index
```

```
        //  of the currently selected item.
        //  If there is no selection, the return value is LB_ERR.
        //-
        currentTarget =

        SendDlgItemMessage( hDlg, IDC_TARGET_LIST, LB_GETCURSEL, 0,0
);

        //+
        //  The GetDlgItemText function retrieves the title or
        //  text associated with a control in a dialog box.
        //-
        GetDlgItemText( hDlg,
                        IDC_AUTHORIZATION_CODE,
                        authorizationCode,
                        32 );
         GetDlgItemText( hDlg, IDC_MISSION_VALUE, missionString, 32
);
        GetDlgItemText( hDlg, IDC_TARGET_VALUE,  targetString, 32 );

        //+
        //  The wsprintf function formats and stores a series of
characters
        //  and values in a buffer. Any arguments are converted and
copied

        //  to the output buffer according to the corresponding
format
        //  specification in the format string.
        //
        //  The function appends a terminating null character to
        //  the characters it writes, but the return value does
        //  not include the terminating null character in its
        //  character count.
        //
        //  use this rather than sprintf() in a Win32-based
application
        //-
        wsprintf( executionString,
        TEXT("%s (%s) Authorization Code - %s, Has been carried
out!"),
          missionString,
          targetString,
          authorizationCode );
        //+
        //  The MessageBox function creates, displays, and operates
        //  a message box. The message box contains an application-
defined
        //  message and title, plus any combination of predefined
icons
        //  and push buttons.
        //
        //  MessageBox() returns a value based on the type
        //  indicated in the 4th parameter and user response.
        //
        //  it is used similarly to the X/Motif
XmCreateQuestionDialog()
        //
        //  dialog = XmCreateQuestionDialog (w, "notice", args, n);
        //
        //-
```

```
        MessageBox( hDlg,
                    executionString,
                    TEXT("Mission Execution"),
                    MB_OK );
    }
    break;

    case IDC_EXIT:
      DestroyWindow( hDlg );
      break;

    default:
      return FALSE;
    }

    break;

  case WM_DESTROY:
    PostQuitMessage( 0 );
    break;

  case WM_CLOSE:
      DestroyWindow( hDlg );
      break;



  default:
    return FALSE;
  }

  return FALSE;

}

//+
//  This example application is based on DialogBox.
//
//  WinMain calls CreateDialog(), shows that window and
//  enters the message loop.
//
//  This sort of implementation can be used to quickly
//  develop applications that require a small GUI.
//  There is however, nothing to prevent large applications
//  from using this model.
//-
int WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR  lpCmdLine, int nCmdShow )
{
  MSG       msg;
  hInst     = hInstance;

  //+
  // Create modeless dialog box.
  //
  // The CreateDialog macro creates a modeless dialog box from
  // a dialog box template resource.
  //
  // This resource is found in DialogWindow.rc.
  hExampleDlg =
    CreateDialog( hInstance,
```

```
                 MAKEINTRESOURCE(EXAMPLE_DIALOG),
                (HWND)NULL,
                (DLGPROC)ExampleDlgProc );

  if ( hExampleDlg != NULL ) {
    ShowWindow( hExampleDlg, SW_SHOW);
  }

  while ( GetMessage( &msg, NULL, 0, 0 ) ) {
    if ( hExampleDlg == 0 || !IsDialogMessage( hExampleDlg, &msg)) {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
  }

  return ( msg.wParam );
}
```

Send feedback to Microsoft