QuickTime Guide for Windows

QuickTime > QuickTime for Windows

2006-01-10

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Tables and Listings

Introduction to QuickTime Guide For Windows

QuickTime is perfectly at home working with Windows code. But because it grew up in the Macintosh world, QuickTime uses some Mac OS features that don't exist in Windows.

The event-loop structure of Windows programming is remarkably similar to that of the Macintosh, however. The differences lie in the implementation details, not in the basic approach. The major differences that affect QuickTime are discussed in this document, along with sample code illustrating usage.

> **Note:** This document was originally titled *QuickTime for Windows*.

If you are a Windows programmer, you'll want to read this document to understand what you need to add to your Windows code to make it fully compatible with QuickTime.

Organization of This Document

This document is divided into three chapters:

- QuickTime For Windows (page 9) discusses the basic concepts you need to understand in building QuickTime applications for Windows, as well as the fundamental differences between both platforms.
- Building QuickTime Capability Into a Windows Application (page 13) describes how to add QuickTime capability to your Windows program, along with the underlying QuickTime Media Layer (QTML) concepts they're based on.
- Windows Utility Routines (page 39) discusses the "glue" routines that will help you write code to run on both Mac OS and Windows platforms.

See Also

The following Apple books cover the basics of QuickTime programming:

- *QuickTime Overview* gives you the starting information you need to do QuickTime programming.
- *QuickTime Movie Basics* introduces you to some of the basic concepts you need to understand when working with QuickTime movies.
- *QuickTime Movie Creation Guide* describes some of the different ways your application can create a new QuickTime movie.
- *QuickTime API Reference* provides encyclopedic details of all the functions, callbacks, data types and structures, atom types, and constants in the QuickTime API.

QuickTime For Windows

If you are a Windows developer, the QuickTime Software Development Kit (SDK) for Windows allows you to incorporate QuickTime capabilities into your applications developed directly for the Windows platform. If you are a Macintosh developer, the SDK provides you with the tools you need to port the QuickTime-based functionality of your application to Windows. This chapter discusses some of the fundamental concepts you need to understand in order to work with both QuickTime and Windows.

Overview

The core of the QuickTime SDK for Windows is a Windows dynamic link library (DLL) that implements the behavior of QuickTime and a few Macintosh Toolbox routines on the Windows platform. This DLL is intended only for QuickTime cross-platform support, not as a general tool for porting Macintosh code to Windows.

Because the QuickTime routines were originally designed for the Mac OS, they operate on Mac OS data structures and assume certain features of the Mac OS operating environment. For example, QuickTime routines are driven by Mac OS-style events rather than Windows-style messages, and do their drawing in a Mac OS graphics port instead of a Windows device context. To use them in the Windows environment, you have to do a little extra work to mediate between the two platforms.

Table 1-1 lists the basic QuickTime Media Layer (QTML) concepts and their Windows counterparts.

Table 1-1 Windows and QTML concepts compared

Windows concept	QTML equivalent
Message (`MSG`)	Event (`EventRecord`)
Graphics Device Interface (GDI)	QuickDraw
Device context (DC)	Graphics port (`CGrafPort`)
Window handle (`HWND`)	Window pointer (`CWindowPtr`)
Common Dialog Box Library	Standard File Package

The goal here is simply to show how QuickTime fits into the structure of a typical Windows application and to provide Windows developers with the minimum conceptual foundation needed to read and understand the existing QuickTime documentation.

With those objectives in mind, the programming examples in this document have deliberately been kept simple and straightforward. The code samples are limited to the most basic QuickTime functionality: presenting a movie and allowing the user to manipulate and control its presentation through a standard QuickTime movie controller.

Mac OS and Windows Differences

The event-loop structure of Windows programming is remarkably similar to that of the Macintosh. The differences lie in the implementation details, not in the basic approach. The major differences that affect QuickTime are as follows:

- Graphics environments. QuickTime draws to the user's screen by using Mac OS QuickDraw, and QuickDraw depends on a data structure called a graphics port to define such characteristics as pen width, background color, clipping boundaries, and current text font. Windows uses the device context for similar purposes, but a device context is specific to a particular device (such as a window or printer), whereas the Mac OS graphics port is global to all drawing operations at a given time.
- Data containers. Mac OS resources are items of structured data that are stored in files and can be loaded on demand to help determine a program's behavior. Resources in application files usually contain descriptions of such user interface items as menus and dialog boxes, but even the executable code of an application is stored as a resource in the application's file. Although Windows uses the concept of resources as well, they're far less important to the system's software architecture.
- Data type codes. QuickTime uses four-character codes to identify such items as track types, media types, and component types. Internally, such codes are simply 32-bit integers; in source code, they are typically represented by four-character strings in single quotation marks, such as `'abcd'`.
- File forks. Every Macintosh file consists of two separate sections, logically joined under a single file name. The data fork consists of a single stream of data bytes intended to be read sequentially; it corresponds to what DOS and Windows generally treat as a file. The Macintosh resource fork contains a heap of individual resources that are accessed by means of four-character resource types and integer resource IDs. Because DOS/Windows files aren't forked, information that would normally go into Macintosh files must be restructured to fit them.
- Messaging. Mac OS events are similar in concept to Windows messages and carry essentially the same information. But a Windows message is directed to a specific window, whereas a Mac OS event is addressed globally to the currently running program.
- Window registration. QuickTime designates a window by a pointer to a Mac OS window record, which contains its own graphics port. On the Windows platform, however, a window is normally designated by an `HWND` handle. Before QuickTime for Windows can draw in a window, your code must register the window with QuickTime by calling `CreatePortAssociation`. This function creates a graphics port and associates it with the window in a data structure that is internal to QuickTime.
- File typing. Every Macintosh file is stamped with a four-character file type and a four-character creator signature, which identifies the application program to which the file belongs. This technique achieves many of the same goals as that of three-character file name extensions in DOS/Windows.
- String formats. DOS/Windows uses C-style strings (terminated by a null character), whereas QuickTime routines use strings in Pascal form (preceded by a 1-byte length count). The QuickTime utility functions `c2pstr` and `p2cstr` convert strings from one format to the other.
- File structures. DOS/Windows files don't have a counterpart to the Macintosh resource fork; they have only the equivalent of the data fork. This leads to problems when you store QuickTime movies or applications using the Windows file system.
- Movie files. To store a QuickTime movie in a Windows file, you need some way to store the media that the movie will present separately from the movie structure.

- Application files. When porting existing QuickTime applications from Macintosh to Windows, the problem arises of how to transport resources belonging to the application program itself. Such resources are typically stored in a separate 'qtr' file while they are being developed in the Macintosh environment; they ultimately wind up in the resource fork of the Mac application ('APPL') file.

A utility named RezWack, available in the QuickTime Windows SDK at http://developer.apple.com/sdk/index.html, moves these resources into an executable (.exe) or dynamic-link library (.dll) file in the Windows environment. QuickTime's resource management routines will correctly locate and load movie resources when they are stored this way in a Windows application.

Building QuickTime Capability Into a Windows Application

Incorporating the QuickTime routines into the structure of a Windows application program is relatively straightforward. You need to follow the basic steps outlined here to build a simple QuickTime capability into your Windows program. Names in parentheses are those of the relevant QTML routines.

1. Initialize the QuickTime Media Layer (`InitializeQTML`) and QuickTime (`EnterMovies`) at the start of your program.

2. Associate a QuickDraw graphics port with your movie window (`CreatePortAssociation`).

3. Open a movie file (`OpenMovieFile`) and extract the movie from it (`NewMovieFromFile`).

4. Create a movie controller for displaying the movie on the screen (`NewMovieController`).

5. In your window procedure, convert incoming messages to QTML events (`WinEventToMacEvent`) and pass them to the movie controller for processing (`MCIsPlayerEvent`).

6. Dispose of the movie (`DisposeMovie`) and its controller (`DisposeMovieController`) when they're no longer needed.

7. Dispose of your movie window's graphics port when the window is destroyed (`DestroyPortAssociation`).

8. Terminate QuickTime (`ExitMovies`) and the QuickTime Media Layer (`TerminateQTML`) at the end of your program.

Listing 2-1 illustrates, in skeletal form, how these steps fit into the structure of a typical Windows application program.

Listing 2-1 Skeleton of a Windows program using QuickTime

```
// Resource identifiers
             .
             .
#define  IDM_OPEN  101
             .
             .
// Global variables
    char            movieFile[255];                // Name of movie file
    Movie           theMovie;                      // Movie object
    MovieController theMC;                         // Movie controller
/////////////////////////////////////////////////////////////////////////////
int CALLBACK WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow)

    {
            .
            .
```

```
        InitializeQTML(0);                            // Initialize QTML
        EnterMovies();                                // Initialize QuickTime
            .
            .
        ////////////////////////////////////////////////////////////////////
        // Main message loop
                .
                .
        //
        ////////////////////////////////////////////////////////////////////
            .
            .
        ExitMovies();                                 // Terminate QuickTime
        TerminateQTML();                              // Terminate QTML

    } /* end WinMain */
////////////////////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

    {
        MSG          winMsg;
        EventRecord  qtmlEvent;
        int          wmEvent, wmId;

        // Fill in contents of MSG structure
                .
                .
        // Convert message to a QTML event
        NativeEventToMacEvent (&winMsg, &qtmlEvent);

        // Pass event to movie controller
        MCIsPlayerEvent (theMC, (const EventRecord *) &qtmlEvent);

        switch ( message )
            {
                case WM_CREATE:
                    // Register window with QTML
                    CreatePortAssociation (hWnd, NULL);
                    break;

                case WM_COMMAND:
                    wmEvent = HIWORD(wParam);  // Parse menu selection
                    wmId   = LOWORD(wParam);

                    switch ( wmId )
                        {
                            case IDM_OPEN:
                                // Close previous movie, if any
                                CloseMovie ();

                                // Get file name from user
                                if ( GetFile (movieFile) )

                                    // Open the movie
                                    OpenMovie (hWnd, movieFile);

                                break;
```

```
                                    .
                                    .
                              default:

                                  return DefWindowProc (hWnd, message,
                                                         wParam, lParam);

                          } /* end switch ( wmId ) */

                      break;

                  case WM_CLOSE:

                      // Unregister window with QTML
                      DestroyPortAssociation (hWnd);
                      break;

                          .
                          .

                  default:
                      return DefWindowProc (hWnd, message, wParam, lParam);

              } /* end switch ( message ) */

          return 0;

      } /* end WndProc */
//////////////////////////////////////////////////////////////////////////////
BOOL GetFile (char *movieFile)

    {
      OPENFILENAME  ofn;

      // Fill in contents of OPENFILENAME structure
                .
                .

      if ( GetOpenFileName(&ofn) )      // Let user select file
        return TRUE;
        else
        return FALSE;

    } /* end GetFile */

//////////////////////////////////////////////////////////////////////////////
void OpenMovie (HWND hwnd, char fileName[255])
    {
        short  theFile = 0;
        FSSpec  sfFile;
        char   fullPath[255];

        // Set graphics port
        SetGWorld ( (CGrafPtr)GetNativeWindowPort( hwnd ), nil);

        strcpy (fullPath, fileName);                  // Copy full pathname
```

```
        c2pstr (fullPath);                          // Convert to Pascal string

        FSMakeFSSpec (0, 0L, fullPath, &sfFile);      // Make file-system
                                                      //  specification record

        OpenMovieFile (&sfFile, &theFile, fsRdPerm);  // Open movie file
        NewMovieFromFile (&theMovie, theFile, nil,    // Get movie from file
                          nil, newMovieActive, nil);

        CloseMovieFile (theFile);                     // Close movie file

        theMC = NewMovieController (theMovie, ... );  // Make movie controller
            .
            .

    } /* end OpenMovie */
/////////////////////////////////////////////////////////////////////////////
void CloseMovie (void)
    {
        if ( theMC )                        // Destroy movie controller, if any
            DisposeMovieController (theMC);

        if ( theMovie )                     // Destroy movie object, if any
            DisposeMovie (theMovie);

    } /* end CloseMovie */
```

Basic QTML Routines

This section discusses the basic QuickTime Media Layer (QTML) routines for building QuickTime capabilities into your Windows application, along with the underlying QTML concepts they're based on.

Initializing and Terminating QTML and QuickTime

Before your program can perform any QuickTime operations, you must initialize the QuickTime Media Layer and then QuickTime itself. The first is accomplished by calling a routine named `InitializeQTML`, the second with `EnterMovies`.

`InitializeQTML` must be called at the very beginning of your program, before any other QuickTime call. The recommended place to call it is in your `WinMain` function, before creating your main window. The function is defined as follows:

```
OSErr InitializeQTML (long flag);
```

The `flag` parameter allows you to specify certain options for the way QuickTime will behave:

Term	Definition
`kInitQTMLUseDefault`	Use standard behavior.

Term	Definition
`kInitQTMLUseGDIFlag`	Use the Windows Graphics Device Interface (GDI) for all drawing, rather than the DirectDraw or DCI services.
`kInitQTMLNoSoundFlag`	Don't initialize the Sound Manager; disable sound for all movies.
`kInitializeQTMLDisableDirectSound`	Disable QTML's use of DirectSound.
`kInitializeQTMLUse-ExclusiveFullScreenModeFlag`	Operate exclusively in full screen mode, in versions of QuickTime later than 3.0.

In most cases, you'll just want to set this parameter to `kInitQTMLUseDefault`, but other options are also available for unusual cases, either singly or in combination.

The function returns an error code indicating success (zero) or failure (nonzero). You can test this result and take appropriate action in case of failure, such as displaying a message box to inform the user that QuickTime is not available. Depending on the nature of your program, you might then choose either to terminate the program or to continue with QuickTime-related features disabled.

If you are writing a routine that does not know from context whether `InitializeQTML` has already been called, add a call to `InitializeQTML` at the beginning of the routine and a call to `TerminateQTML` at the end. It does no harm to call `InitializeQTML` more than once, as long as each call is nested with a matching call to `TerminateQTML`. If `InitializeQTML` has already been called, subsequent calls do nothing except increment a counter. Calls to `TerminateQTML` just decrement the counter (if it is nonzero). Only the first nested call to `InitializeQTML` and the last nested call to `TerminateQTML` do any actual work, so there is no penalty for having nested calls.

The `EnterMovies` function allocates space for QuickTime's internal data structures and initializes their contents. Your program should call this function immediately after calling `InitializeQTML`. The function takes no parameters and returns an error code:

```
OSErr EnterMovies (void);
```

Again, you can test the result and do whatever is appropriate in case of failure.

At the end of the program, your initialization calls to `InitializeQTML` and `EnterMovies` should be balanced by corresponding calls to the termination routines `ExitMovies` and `TerminateQTML`. Both of these functions take no parameters and return no result.

```
void ExitMovies (void)
void TerminateQTML (void)
```

Listing 2-2 shows how these initialization and termination calls fit into the structure of a typical `WinMain` routine.

Listing 2-2 Main routine of a Windows program using QuickTime

```
int CALLBACK WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
    {
        MSG   msg;
        HANDLE hAccelTable;
```

```
    if ( !hPrevInstance )                // Is there a previous instance?
        if ( !(InitApplication(hInstance)) )   // Register window class
            return (FALSE);                    // Report failure

    if ( InitializeQTML(0) != noErr )          // Initialize QTML
        {
            MessageBox (hWnd, "QuickTime not available", // Notify user
                            ", MB_OK);
            return (FALSE);                    // Report failure

        } /* end if ( InitializeQTML(0) != noErr ) */

    if ( EnterMovies() != noErr)              // Initialize QuickTime
        {
            MessageBox (hWnd, "QuickTime not available", // Notify user
                            ", MB_OK);
            return (FALSE);                   // Report failure

        } /* end if ( EnterMovies() != noErr ) */

    if ( !(InitInstance(hInstance, nCmdShow)) )  // Create main window
        return (FALSE);                          // Report failure

    hAccelTable = LoadAccelerators(hInstance,  // Load accelerator table
                    MAKEINTRESOURCE(IDR_ACCELSIMPLESDI));


    ////////////////////////////////////////////////////////////////////
    // Main message loop
    ////////////////////////////////////////////////////////////////////

    while ( GetMessage(&msg, NULL, 0, 0) )      // Retrieve next message

        // Check for keyboard accelerator
        if ( !TranslateAccelerator (msg.hwnd,
                        hAccelTable, &msg) )
            {
                // Convert virtual key to character
                TranslateMessage(&msg);

                // Send message to window procedure
                DispatchMessage(&msg);

            } /* end if ( !TranslateAccelerator
                              (msg.hwnd, hAccelTable, &msg) ) */

    ////////////////////////////////////////////////////////////////////


    ExitMovies();                  // Terminate QuickTime
    TerminateQTML();               // Terminate QTML

    return (msg.wParam);

} /* end WinMain */
```

Using Graphics Ports

Because of its Mac OS origins, QuickTime uses the QuickDraw graphics routines to draw to the screen. QuickDraw consitutes the Macintosh counterpart to the Windows Graphics Device Interface, or GDI. Even when running under Windows, the QuickTime Media Layer compatibility interface allows the QuickTime routines to use QuickDraw calls internally for their drawing operations. So in order to use QuickTime properly, you have to understand a little about QuickDraw.

The fundamental QuickDraw data structure is the **graphics port** (analogous to a Windows device context). This is a complete drawing environment that specifies all of the parameters needed to control QuickDraw's drawing operations. The port includes such things as the size and location of the line-drawing pen; colors and patterns (like brushes in Windows) for drawing, area fill, and background; the font, size, and style for text display; clipping boundaries; and so forth. All of this information is held in a data structure of type `CGrafPort`, pointed to by a pointer of type `CGrafPtr`.

The `C` in `CGrafPort` and `CGrafPtr` stands for "color," to distinguish these from the "classic" black-and-white versions of these structures (`GrafPort` and `GrafPtr`), which are now obsolete. Any QTML routine that nominally expects a `GrafPort` or `GrafPtr` will accept a `CGrafPort` or `CGrafPtr` instead.

The main purpose of a graphics port is to serve as the environment in which to perform QuickDraw graphics operations. Unlike the Windows GDI routines, which always accept a device context as an explicit parameter, most QTML QuickDraw routines operate implicitly on the **current port.** At any given time, exactly one graphics port is current. The QTML routine `GetPort`

```
void GetPort (GrafPtr *port)
```

returns a pointer to the current port, and `MacSetPort`

```
void MacSetPort (GrafPtr  port)
```

changes it.

The original Mac OS name of this routine, `SetPort`, conflicts with an existing name in the Windows API and had to be changed to `MacSetPort`.

Graphics ports are closely associated with windows on the screen; the current port for QuickDraw drawing operations is typically a window. When running in the Windows environment, you have to associate a Mac OS-style graphics port with your movie window for the QuickTime routines to use in displaying a movie. The next section discusses how to accomplish this.

Redefined API Names

Some names defined in the Macintosh application programming interfaces conflict with identical names in the Windows API. In these cases, the QTML header file `QTMLMapNames`.h avoids these conflicts by redefining the affected names with the prefix `Mac` added.

In Table 2-2, names listed in the first column refer to the original Macintosh function or data structure name; the second column gives the redefined or newly mapped names.

Table 2-1 Redefined API names

Original Macintosh API name	Mapped name
AnimatePalette	MacAnimatePalette
AppendMenu	MacAppendMenu
CloseDriver	MacCloseDriver
CloseWindow	MacCloseWindow
CompareString	MacCompareString
CopyRgn	MacCopyRgn
DeleteMenu	MacDeleteMenu
DrawMenuBar	MacDrawMenuBar
DrawText	MacDrawText
EqualRect	MacEqualRect
EqualRgn	MacEqualRgn
FillRect	MacFillRect
FillRgn	MacFillRgn
FindWindow	MacFindWindow
FlushInstructionCache	MacFlushInstructionCache
FrameRect	MacFrameRect
FrameRgn	MacFrameRgn
GetClassInfo	MacGetClassInfo
GetCurrentThread	MacGetCurrentThread
GetCursor	MacGetCursor
GetDoubleClickTime	MacGetDoubleClickTime
GetFileSize	MacGetFileSize
GetItem	MacGetItem
GetMenu	MacGetMenu
GetNextWindow	MacGetNextWindow
GetParent	MacGetParent

Original Macintosh API name	Mapped name
GetPath	MacGetPath
GetPixel	MacGetPixel
InsertMenu	MacInsertMenu
InsertMenuItem	MacInsertMenuItem
InsetRect	MacInsetRect
InvertRect	MacInvertRect
InvertRgn	MacInvertRgn
IsWindowVisible	MacIsWindowVisible
LineTo	MacLineTo
LoadResource	MacLoadResource
MoveWindow	MacMoveWindow
OffsetRect	MacOffsetRect
OffsetRgn	MacOffsetRgn
OpenDriver	MacOpenDriver
PaintRgn	MacPaintRgn
Polygon	MacPolygon
PtInRect	MacPtInRect
Region	MacRegion
ReplaceText	MacReplaceText
ResizePalette	MacResizePalette
SendMessage	MacSendMessage
SetCursor	MacSetCursor
SetItem	MacSetItem
SetPort	MacSetPort
SetRect	MacSetRect
SetRectRgn	MacSetRectRgn
ShowCursor	MacShowCursor

Original Macintosh API name	Mapped name
`ShowWindow`	`MacShowWindow`
`StartSound`	`MacStartSound`
`StopSound`	`MacStopSound`
`TokenType`	`MacTokenType`
`UnionRect`	`MacUnionRect`
`UnionRgn`	`MacUnionRgn`
`XorRgn`	`MacXorRgn`

Window Records

Because most drawing on the screen takes place in a window, graphics ports are also the basis of the QTML **window record** (`CWindowRecord`).

The only point to notice here is that its first field (`port`) holds not a pointer to a graphics port, but actually a complete graphics port structure embedded directly in the window record. At the machine level, this means that the window record is simply an extended graphics port with some additional, window-specific information appended at the end. In fact, the pointer to a color window (`CWindowPtr`) is directly equated to the corresponding graphics port pointer (`CGrafPtr`):

```
typedef  CGrafPtr  CWindowPtr;
```

This allows a window to be used in place of a graphics port in any context in which a port would be valid. Any QuickDraw routine that expects a pointer to a graphics port as a parameter will accept a window pointer in its place, since the two pointers are really the same data type. In particular, the QuickTime routines can pass your window pointer to the `MacSetPort` function discussed in the preceding section, making the window the current port in which to display the contents of a movie.

On the Windows platform, however, your window is normally designated by a Windows-style handle (`HWND`) rather than a QTML pointer (`CWindowPtr`). To allow QuickTime to draw into the window, you must first **register** it with QTML by calling the QTML routine `CreatePortAssociation`.

```
void CreatePortAssociation
        (void  *theWnd,
         Ptr   storage
         long  flags);
```

This creates a graphics port and associates it with this window in an internal data structure maintained by QTML. The first parameter (`theWnd`) is your Windows-style window handle, of type `HWND`. The second parameter (`storage`) allows you to supply your own storage for the `CGrafPort` record, if you wish. Generally, you will always pass `nil`, allowing the call to allocate memory. (If you leave this parameter `null`, QTML will allocate the space for you.)

Typically, you'll want to register your movie window at the time it is created by calling `CreatePortAssociation` from your window procedure in response to the `WM_CREATE` message, as shown in Listing 2-3.

Listing 2-3 Creating a port association

```
LRESULT CALLBACK WinProc
        (HWND   thisWindow,                        // Handle to window
         UINT   msgType,                           // Message type
         WPARAM  wParam,                          // Message-dependent parameter

         LPARAM  lParam)                          // Message-dependent parameter

    {
            .
            .

        switch ( msgType )
            {
                case WM_CREATE:
                    CreatePortAssociation (thisWindow, NULL);
                                                   // Register window with QTML

                    break;

                        .
                        .

            } /* end switch ( msgType ) */
    } /* end WinProc */
```

Once you've registered your window, you can use the conversion routine `GetHWNDPort` to obtain a QTML-style window pointer for it:

```
WindowPtr GetNativeWindowPort (void *h)
```

There's also a reverse conversion function for recovering the window handle associated with a given window pointer.

```
void* GetPortNativeWindow (WindowPtr  wptr)
```

When you're through with a particular window, you can deregister it and dispose of its graphics port with `DestroyPortAssociation`:

```
void DestroyPortAssociation (CGrafPtr  cgp)
```

A good place to do this is in your window procedure's response to the `WM_CLOSE` or `WM_DESTROY` message. Listing 2-4 shows an example of how to destroy a port association.

Listing 2-4 Destroying a port association

```
LRESULT CALLBACK WinProc
        (HWND    thisWindow,                        // Handle to window
         UINT    msgType,                           // Message type
         WPARAM  wParam,                          // Message-dependent parameter
```

```
    LPARAM  lParam)                         // Message-dependent parameter
{
            .
            .
    switch ( msgType )
        {
            case WM_CLOSE:
                CWindowPtr  qtmlPtr;            // Macintosh window pointer

                // Convert to window pointer
                qtmlPtr = GetHWNDPort(thisWindow);

                DestroyPortAssociation (qtmlPtr);   // Deregister window
                break;

                    .
                    .
        } /* end switch ( msgType ) */
            .
            .
} /* end WinProc */
```

Graphics Worlds (GWorlds) and How To Work With Them

Another aspect of the graphics environment that affects the way QuickTime displays images on the screen is the characteristics of the graphics device on which they're being presented. These include such things as the device's pixel resolution, its color depth, and the capacity of its color table. The device's characteristics are summarized in a **graphics device record.**

Ordinarily, the results of a program's drawing operations depend on the graphical capabilities of the display device that happens to be connected to the user's computer at run time. There can even be more than one such device attached to the same system: QTML will figure out which screen is being drawn to and will display all results correctly according to the characteristics of each device. All of this normally happens automatically, and is transparent to the running program.

Sometimes, however, a program may need to take a more active role in controlling the graphics environment for its drawing operations. If you're creating a QuickTime movie, for instance, you probably don't want to define the movie's appearance in terms of the display characteristics of a particular graphics device. Rather, you want the movie's content to be device-independent, with its own inherent dimensions, pixel depth, colors, and so on. Then, when the movie is displayed on a user's computer, QuickTime will automatically adapt its graphical characteristics to those of the available display device, and will present the movie as faithfully as it can on the given device.

The way to accomplish this is to define the movie with respect to a device-independent **graphics world.** This combines a graphics port and a device record, which together completely determine the graphics environment in which QuickTime does its drawing. Like the window record we discussed in the preceding section, the data structure representing a graphics world is an extended graphics port with some additional fields appended at the end. The exact details are private to QTML; the graphics world is always referred to by means of an opaque pointer of type `GWorldPtr`. Because the underlying structure is based on a graphics port, however, this pointer is equated to a graphics port pointer:

```
typedef  CGrafPtr  GWorldPtr;
```

This means that (again, like a window record), a graphics world can be used anywhere a graphics port would be expected: for instance, as an argument to the `MacSetPort` function that sets the current port for subsequent drawing operations.

A graphics world's device record can represent an existing physical graphics device, but it need not: it can also describe a fictitious "offscreen" device with any graphical characteristics you choose. You create such an offscreen graphics world by specifying the desired characteristics as parameters to the QTML function `NewGWorld`:

```
QDErr NewGWorld
        (GWorldPtr    *offscreenGWorld,   // Returns pointer to GWorld
         short        pixelDepth,         // Color depth in bits per pixel
         const Rect   *boundsRect,        // Boundary rectangle
         CTabHandle   cTable,             // Handle to color table
         GDHandle     aGDevice,           // Set to null for offscreen
         GWorldFlags  flags);             // Option flags
```

Notably, if the `noNewDevice` flag in the `flags` parameter is clear, the function will ignore the parameter `aGDevice` and create a new, device-independent device record with the specified characteristics. It will then combine this device record with a graphics port for drawing into a memory-based image buffer (rather than directly to the screen), and will return a pointer to the resulting graphics world via the `offscreenGWorld` parameter.

When you use `NewGWorld` to create your graphics world, it will be set up to draw into a Macintosh-style `bitmap` as its image buffer. If you want to work with a Windows-style `bitmap` instead, you can use an alternate function available only in the Windows version of the QuickTime API.

```
QDErr NewGWorldFromHBITMAP
        (GWorldPtr    *offscreenGWorld,   // Returns pointer to GWorld
         CTabHandle   cTable,             // Handle to color table
         GDHandle     aGDevice,           // Set to null for offscreen
         GWorldFlags  flags,              // Option flags
         void         *newHBITMAP,        // Handle to bitmap
         void         *newHDC)            // Handle to device context
         long         rowBytes)           // number of bytes in a scanline
```

The parameters `newHBITMAP` and `newHDC` must either both be `null` or handles to a Windows `bitmap` and device context, respectively. If they're null, the function will allocate a complete graphics world for you; otherwise, it will simply wrap one around the specified structures. This allows you to use the native Windows drawing environment as the source for QuickTime operations such as image compression or `CopyBits`. If you do supply a Windows `bitmap`, it must be a device-independent `bitmap` (DIB) created with the Windows function `CreateDIBSection`.

```
    0                          // Default
    k1MonochromePixelFormat
    k2IndexedPixelFormat
    k4IndexedPixelFormat
    k8IndexedPixelFormat
    k1IndexedGrayPixelFormat
    k2IndexedGrayPixelFormat
    k4IndexedGrayPixelFormat
```

```
    k8IndexedGrayPixelFormat
    k16BE555PixelFormat
    k32ARGBPixelFormat
    k16LE555PixelFormat
    k16LE565PixelFormat
    k24BGRPixelFormat
    k24RGBPixelFormat
    k32BGRAPixelFormat
    k32ABGRPixelFormat
    k32RGBAPixelFormat
```

Once you've created a graphics world to your specifications, you can use it to set the current graphics port and device, then you can proceed to create your movie. The QTML function `SetGWorld`

```
void SetGWorld
        (CGrafPtr  port,       // Port or graphics world to make current
         GDHandle  gdh)        // Device to make current
```

nominally accepts a graphics port and device record and makes them the current port and current device. However, if the `port` parameter actually points to a graphics world (remember that data types `GWorldPtr` and `CGrafPtr` are equivalent), then the function ignores parameter `gdh` and uses the port and device from the given graphics world instead. A companion function, `GetGWorld`

```
void GetGWorld
        (CGrafPtr *port,       // Returns current port
         GDHandle *gdh)        // Returns current device
```

returns a pointer to the current port and a handle to the current device record. You can use this function, for example, to save the previous current port and device and restore them again after you're finished creating your movie. Listing 2-5 shows an example of how to use an offscreen graphics world.

Listing 2-5 Using an offscreen graphics world

```
CGrafPtr   oldPort;                   // Previous current port
GDHandle   oldDevice;                 // Previous current device
GWorldPtr  movieGWorld = nil;         // Movie's graphics world
Rect       movieFrame;                // Boundary rectangle for movie images
OSErr      errCode;                   // Result code
        .
        .
errCode = NewGWorld (&movieGWorld,    // Return result in movieGWorld
                    16,               // Pixel depth
                    &movieFrame,      // Boundary rectangle
                    nil,              // Use default color table
                    nil,              // No preexisting device record
                    0 );              // No flags to pass
if ( errCode != noErr )               // Was there an error?
   MessageBox (hWnd, "Error creating graphics world", ", MB_OK); // Notify user

else
    {
        GetGWorld (&oldPort, &oldDevice); // Save previous graphics world
        SetGWorld (movieGWorld, nil);     // Set movie's graphics world

            /* Here...you would draw images */

        SetGWorld (oldPort, oldDevice);   // Restore previous graphics world
        DisposeGWorld (movieGWorld);      // Dispose of movie's graphics world
```

```
    } /* end else */
```

Besides the general SetGWorld and GetGWorld functions, the QuickTime Movie Toolbox also provides a pair of functions for setting and retrieving a movie's graphics world directly:

```
void SetMovieGWorld
        (Movie     theMovie,
         CGrafPtr  port,
         GDHandle  gdh)

void GetMovieGWorld
        (Movie     theMovie,
         CGrafPtr  *port,
         GDHandle  *gdh)
```

They are useful for drawing offscreen because you can create GWorlds and then direct the movie to draw them there.

Similar to SetGWorld, the SetMovieGWorld function will accept a graphics world as its first parameter in place of a graphics port; it will then ignore the second parameter and use the device record from the graphics world instead.

Mixing QuickDraw and Win32 Drawing

The Win32 implementation of QuickDraw incorporates Win32 graphics elements, thus enabling much easier integration. Resources are allocated using native Win32 elements. If you need these resources, there are accessor functions provided for their retrieval.

By default, QuickDraw allocates GWorlds by creating DIB sections. The DIB's memory (where the actual pixel values are stored) is shared by the GWorld. The GWorld's PixMap base address points to the location of the HBITMAP's pixel values. In this way, both QuickDraw and Win32 can draw into the same graphic environment.

An HDC is created and the HBITMAP is selected into it for the duration of the GWorld. Any changes in the GWorld drawing environment are reflected in the corresponding HDC and HBITMAP. If QuickDraw is unable to allocate the GWorld as a DIB section, it falls back to allocating the offscreen drawing pixels in memory.

You can access a GWorld's HDC and HBITMAP by using QuickTime's GetPortHDC and GetPortHBITMAP functions. Your application must not dispose of the HBITMAP or HDC returned from these calls. They are owned by QuickTime and will automatically be disposed of when the GWorld is disposed.

You may also request that the offscreen GWorld is allocated as a DirectDraw surface by passing the kAllocDirectDrawSurface flag to the NewGWorld family of calls.

Rendering into an HBITMAP

A common question is, how to use QuickTime or QuickDraw to render into an HBITMAP? Since most GWorlds are simply wrappers for HBITMAPs, this is straightforward enough to answer.

The RenderIntoHBITMAPExample function shown below demonstrates the relationship of GWorlds and HBITMAPs, as well as the use of QTNewGWorld, SetGWorld, GetPortHDC, and GetPortHBITMAP.

The sample code in listing below first creates a GWorld in a Win32 compatible pixel format (in our example code we use k32BGRAPixelFormat).

Once created the GWorld's HDC and HBITMAP are retrieved using GetPortHDC and GetPortHBITMAP. The active port is selected by calling SetGWorld, and graphics rendered using the QuickDraw APIs RGBForeColor and PaintRect. Additionally, native Win32 GDI calls are used to render graphics into the same GWorld. Finally, the contents of the port are copied to a secondary HDC via ScaleBlt, using the GWorld's HDC as a source. Note that the HBITMAP and HDC are owned and managed by the GWorld, and are not disposed of.

```
OSErr RenderIntoHBITMAPExample(HDC hdcDest, RECT *rectDest)
{
        GWorldPtr gw = nil;
        CGrafPtr savedPort;
        GDHandle savedGD;
        HDC hdcSrc;
        HBITMAP hbitmapSrc;
        Rect bounds;
        OSErr result = noErr;

        // Create a 256 x 256 32BGRA GWorld
        bounds.top = bounds.left = 0;
        bounds.bottom = bounds.right = 256;

        result = QTNewGWorld(&gw,k32BGRAPixelFormat,&bounds,NULL,NULL,NULL);

        // check for errors
        if (result != noErr)
                goto bail;

        // retrieve the associated HDC and HBITMAP
        hdcSrc = GetPortHDC((GrafPtr)gw);
        hbitmapSrc = GetPortHBITMAP((GrafPtr)gw);

        // bail if DIB allocation failed.
        if ((hdcSrc == 0) || (hbitmapSrc == 0)) {
                result = memFullErr;
                goto bail;
        }

        // save current port and GDevice, set current port to new GWorld
        GetGWorld(&savedPort,&savedGD);
        SetGWorld(gw,NULL);
// Render graphics into GWorld
{
        Rect macRect;
        RGBColor color;
        RECT winRect;
        HBRUSH hBrush, hBrushOld;

        // red
        color.red = 0xffff; color.green = 0; color.blue = 0;
        RGBForeColor(&color);
        macRect = bounds;
        PaintRect(&macRect);
```

```
        // green
        color.red = 0; color.green = 0xffff; color.blue = 0;
        RGBForeColor(&color);
        MacInsetRect(&macRect, 20, 20);
        PaintRect(&macRect);

        // blue. just for kicks lets use GDI
        // to render graphics into the same GWorld
        MacInsetRect(&macRect, 20, 20);
        winRect.top = macRect.top;
        winRect.left = macRect.left;
        winRect.bottom = macRect.bottom;
        winRect.right = macRect.right;
        hBrush = CreateSolidBrush(RGB(0,0,0xff));
        hBrushOld = SelectObject(hdcSrc, hBrush);
        FillRect(hdcSrc, &winRect, hBrush);
        GdiFlush();
        DeleteObject(SelectObject(hdcSrc, hBrushOld));
}

        // copy contents of GWorld to dstHDC.
      StretchBlt(hdcDest,rectDest->left,rectDest->top,rectDest->right-rectDest->left,
                rectDest->bottom-rectDest->top,hdcSrc,bounds.left,bounds.top,
                bounds.right-bounds.left,bounds.bottom-bounds.top,SRCCOPY);

        // reset port
        SetGWorld(savedPort,savedGD);

bail:
        // dispose of gworld
        if( gw ) DisposeGWorld(gw);

        return result;
}
```

In some cases, you might want to use QuickTime or QuickDraw to render into an existing DIB section. The `NewGWorldFromHBITMAP` call allows you to wrap an existing DIB section with a `GWorld`. After creating the `DIBSection`, call `NewGWorldFromHBITMAP`, passing in the HDC and HBITMAP, to create a `GWorld` that shares the pixels of the DIB section.

As in the `QTNewGWorld` case, the HBITMAP is selected into the HDC for the life of the `GWorld`. Note that in this case, QuickDraw does not dispose of the HDC or HBITMAP in `DisposeGWorld`, as it did not allocate them. The creator of the HDC and HBITMAP is responsible for disposing of them. Also note that the HBITMAP passed to `NewGWorldFromHBITMAP` must be created using the `CreateDIBSection` API, old style DDBs are not allowed.

```
OSErr RenderIntoExistingDIBExample(HDC hdcDest, RECT *rectDest)
{
        GWorldPtr gw = nil;
        CGrafPtr savedPort;
        GDHandle savedGD;
        HDC hdcSrc,hdcTemp;
        HBITMAP hbitmapSrc;
        BITMAPINFO *bitmapInfo = nil;
        OSErr result = noErr;
        void *baseaddr;
```

```
// create an HDC
hdcTemp = GetDC(NULL);
hdcSrc = CreateCompatibleDC(hdcTemp);
ReleaseDC(NULL,hdcTemp);

// create a DIB section
bitmapInfo = (BITMAPINFO *)NewPtrClear(sizeof (BITMAPINFOHEADER)
              + (3 * sizeof (DWORD)));

bitmapInfo->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
bitmapInfo->bmiHeader.biWidth = 256;
bitmapInfo->bmiHeader.biHeight =  -1 * 256;     // top down dib
bitmapInfo->bmiHeader.biPlanes = 1;
bitmapInfo->bmiHeader.biBitCount = 32;
bitmapInfo->bmiHeader.biCompression = BI_RGB;
bitmapInfo->bmiHeader.biSizeImage = 0;
bitmapInfo->bmiHeader.biXPelsPerMeter = 0;
bitmapInfo->bmiHeader.biYPelsPerMeter = 0;
bitmapInfo->bmiHeader.biClrUsed = 0;
bitmapInfo->bmiHeader.biClrImportant = 0;
bitmapInfo->bmiColors[0].rgbBlue = 0;
bitmapInfo->bmiColors[0].rgbGreen = 0;
bitmapInfo->bmiColors[0].rgbRed = 0;
bitmapInfo->bmiColors[0].rgbReserved = 0;

hbitmapSrc = CreateDIBSection(hdcSrc, bitmapInfo, DIB_RGB_COLORS,
                              &baseaddr, NULL, NULL);

// bail if DIB creation failed
if (hbitmapSrc == 0) {
        result = memFullErr;
        goto bail;
}

DisposePtr((Ptr)bitmapInfo);

// wrap the DIB with a GWorld
result = NewGWorldFromHBITMAP(&gw,NULL,NULL,NULL,hbitmapSrc,hdcSrc);

// check for errors
if (result != noErr)
        goto bail;

// save current port and GDevice, set current port to new GWorld
GetGWorld(&savedPort,&savedGD);
SetGWorld(gw,NULL);

// Render graphics into GWorld
{
        Rect rect;
        RGBColor color;
        color.red = 0xffff; color.green = 0; color.blue = 0;

        RGBForeColor(&color);
        rect.top = rect.left = 0;
        rect.bottom = rect.right = 256;

        PaintRect(&rect);
```

```
                color.red = 0; color.green = 0xffff; color.blue = 0;

                RGBForeColor(&color);
                rect.top = rect.left = 20;
                rect.bottom = rect.right = 236;

                PaintRect(&rect);
        }

        // copy contents of GWorld to dstHDC.
        StretchBlt(hdcDest,rectDest->left,rectDest->top,
                rectDest->right-rectDest->left,rectDest->bottom-rectDest->top,
                hdcSrc,0,0,256,256,SRCCOPY);

        // reset port
        SetGWorld(savedPort,savedGD);

bail:
        // dispose of gworld
        if( gw ) DisposeGWorld(gw);

        DeleteObject(hbitmapSrc);
        DeleteDC(hdcSrc);

        return result;
}
```

File Selection Dialogs

When the user chooses the Open command from your File menu, you'll want to present a dialog box that allows the user to select the file to be opened. In Windows, this is normally done with the function `GetOpenFileName`, part of the Common Dialog Box Library. This function displays the standard Windows Open File dialog box on the screen, handles all interactions with the mouse and keyboard until the dialog is dismissed, and then returns a data structure of type `OPENFILENAME` identifying the file the user has selected. One of the members of this structure, `lpstrFile`, points to a string buffer in which to return the pathname of the file the user has selected. Typically, a Windows program would simply pass this string to the appropriate Windows function, such as `CreateFile`, to open the designated file.

As we'll see in the next section, however, the QuickTime function `OpenMovieFile` instead expects to receive an analogous data structure from the Macintosh Standard File dialog package, a **file-system specification record** (Listing 2-6).

Listing 2-6 File-system specification record

```
struct FSSpec
    {
        short   vRefNum;                // Volume reference number
        long    parID;                  // Directory ID of parent directory
        Str255  name;                   // File name
    }; /* end FSSpec */
```

So before calling `OpenMovieFile` from a Windows program, you have to create a specification record to pass to it. The QTML function `FSMakeFSSpec` can be used to accomplish this.

```
OSErr FSMakeFSSpec
        (short              vRefNum,   // Volume reference number
         long               dirID,     // ID of parent directory
         ConstStr255Param   fileName,  // File name
         FSSpec             *spec)     // Returns a specification record
```

On the Macintosh, files are normally identified by giving a directory ID and a local file name within the directory. In Windows code, you set the directory ID and volume reference number to `0` and supply a full pathname instead; `FSMakeFSSpec` will interpret this correctly and initialize the specification record accordingly. Listing 2-7 shows how to use this function to mediate between the Windows common dialog box and the QTML `OpenMovieFile` function.

Another point to keep in mind is that the Windows `GetOpenFileName` function returns the file's pathname as a C-style string (terminated by a null character), whereas `FSMakeFSSpec`, like all QTML routines, expects it in Pascal form (preceded by a 1-byte length count).

QTML provides a pair of utility functions, `c2pstr` and `p2cstr`, for converting strings from one format to the other in place. You don't want to pass a string constant; the buffer needs to be modifiable.

Listing 2-7 Opening a user-selected movie file

```
OPENFILENAME  ofn;                        // Parameters to Common Dialog Box
char      pathName[255];                  // Buffer for pathname
BOOL      confirmed;                      // Did user confirm file selection?
FSSpec    fileSpec;                       // File-system specification record
short     theFile;                        // Reference number of movie file
HWND      hwnd;                           // Handle to movie window
CGrafPtr  windowPort;                     // Window's graphics port
OSErr     errCode;                        // Result code
        .
        .
memset (&ofn, 0, sizeof(OPENFILENAME);    // Clear to zero
fileName[0] = '\0';                       // No default file name
ofn.lStructSize   = sizeof(OPENFILENAME);    // Size of structure
ofn.hwndOwner    = GetActiveWindow();     // Active window owns dialog
ofn.lpstrFile    = LPSTR(pathName);       // Point to pathname buffer
ofn.nMaxFile    = 255;                    // Size of buffer
ofn.lpstrFilter   = "QuickTime Movies (*.mov;*.avi) \0 *.mov;*.avi\0";
                                          // Filter string
ofn.nFilterIndex  = 1;                    // Index of default filter
ofn.lpstrInitialDir = NULL;               // Use current directory
confirmed = GetOpenFileName (&ofn);       // Let user select file
if ( confirmed )                          // Did user confirm selection?
    {
        c2pstr (pathName);                // Convert to Pascal string

        // Make specification record
        FSMakeFSSpec (0, 0L, pathName, &fileSpec);

        // Get window's graphics port
        windowPort = GetNativeWindowPort( hwnd );
        SetGWorld (windowPort, nil);      // Make it the graphics world

        // Open the movie file
        errCode = OpenMovieFile (&fileSpec, &theFile, fsRdPerm);

    } /* end if ( confirmed ) */
```

Movies and Movie Files

QuickTime movies reside in **movie files.** On the Mac OS platform, such files carry the file type `'MooV'` (defined in the QuickTime interface as a constant named `MovieFileType`); on the Windows platform, they are identified by the file-name extension .mov.

Before reading a movie in from its movie file, you must first open the file with the QuickTime function `OpenMovieFile`.

```
OSErr OpenMovieFile
        (const FSSpec  *fileSpec,       // Identifies file to be opened
         short         *resRefNum,      // Returns file reference number
         SInt8         permission)      // Requested permission level
```

The `fileSpec` parameter points to a **file-system specification record** (described in Listing 2-6) telling which movie file to open. The `OpenMovieFile` function returns a **file reference number,** via the `resRefNum` parameter, that uniquely identifies this movie file. You'll use this reference number to refer to the file when calling other QuickTime routines, such as `CloseMovieFile` and `NewMovieFromFile`. The `permission` parameter specifies the level of access permission requested for the file, such as `fsRdPerm` (read-only), `fsWrPerm` (write-only), or `fsRdWrPerm` (read-write).

After opening the movie file, you can read the movie's contents into a **movie record,** an opaque data structure in which QuickTime reads some information into memory about the movie's contents. The movie record is referred to by a **movie identifier** of type `Movie`.

```
typedef MovieRecord* Movie;
```

The QuickTime function `NewMovieFromFile` creates movie record in memory for the specified file.

```
OSErr NewMovieFromFile
        (Movie      *theMovie,            // Returns movie identifier
         short      resRefNum,            // File reference number
         short      *resID,               // Unused in Windows; set to nil
         StringPtr  resName,              // Unused in Windows; set to nil
         short      newMovieFlags,        // Option flags
         Boolean    *dataRefWasChanged)   // Unused in Windows; set to nil
```

You identify the movie file by supplying the file reference number (`resRefNum`) that you got back from your call to `OpenMovieFile`. Parameter `theMovie` returns a movie identifier for the movie retrieved from the file. Of the possible option flags that you can set in the `newMovieFlags` parameter, the only one of interest on the Windows platform is `newMovieActive`, which controls whether the movie will initially be active or inactive when you read it in; you can later control this setting dynamically with the QuickTime function `SetMovieActive`. The remaining parameters refer to Macintosh-style resources, and are not relevant in the Windows context.

Once you've read a movie in from its file to a movie record and obtained a movie identifier for it, there's no need to keep the movie file open any longer. In the movie record, there are pointers to the file and QuickTime will automatically reopen it to retrieve data, if needed. It's considered good practice to close the file immediately, using the QuickTime function `CloseMovieFile`:

```
OSErr CloseMovieFile (short resRefNum)   // File reference number
```

Once again, you identify the file by using the file reference number you received when you first opened it. After closing the file, the file reference number is invalid. Therefore, passing the reference to another file manager call is not a good idea and should be avoided.

Listing 2-8 illustrates how to combine these QuickTime calls to read a movie in from its movie file.

Listing 2-8 Reading a movie from a file

```
FSSpec  fileSpec;               // Descriptive information on file to open
short   theFile;                // Reference number of movie file
Movie   theMovie;               // Movie identifier
HWND    hWnd;                   // Handle to window
OSErr   errCode;                // Result code
        .
        .
// Open the movie file
errCode = OpenMovieFile (&fileSpec, &theFile, fsRdPerm);
if ( errCode != noErr )                        // Was there an error?
    {
        MessageBox (hWnd, "Error opening movie file", // Notify user
                        ", MB_OK);
        return (FALSE);                        // Report failure

    } /* end if ( errCode != noErr ) */

errCode = NewMovieFromFile (&theMovie, theFile, // Get movie from file
                            nil, nil,
                            newMovieActive, nil);
CloseMovieFile (theFile);                      // Close the file
if ( errCode != noErr)                         // Was there an error?
    {
        MessageBox (hWnd, "Error reading movie from file", // Notify user
                        ", MB_OK);
        return (FALSE);                        // Report failure

    } /* end if ( errCode != noErr ) */
```

Movie Controllers

The preferred way to present a movie is with a **movie controller.** This is a QuickTime component that presents the user with a standard set of controls for running the movie and controlling its direction, speed, and so on.

You create a movie controller with the QuickTime function `NewMovieController`.

```
MovieController NewMovieController
        (Movie        theMovie,             // Movie to be displayed
         const Rect   *movieRect,           // Rectangle to display it in
         long         someFlags)            // Option flags
```

Parameter `theMovie` is the movie identifier you received when you read the movie in with `NewMovieFromFile`. The second parameter, `movieRect`, specifies the rectangle in which to display the movie on the screen. The parameter `someFlags` specifies various options, such as whether to display the movie with a frame around it, how to position it within the specified rectangle, and whether to scale it to fit the rectangle. (If you want it to fit the rectangle exactly, you can get the dimensions of the movie's boundary rectangle with the QuickTime function `GetMovieBox`.)

Because of its Mac OS origins, a movie controller is driven by **events** rather than messages. Events are similar in concept to Windows-style messages, though different in detail. As you can see in Listing 2-9, the QTML **event record** closely resembles the Windows message structure (`MSG`) and contains essentially the same information. (One difference is that unlike a Windows message, the event doesn't identify a particular window to which it applies; this is because all Macintosh events are addressed globally to the program itself, rather than to an individual window.)

Listing 2-9 Event record

```
struct EventRecord
    {
        EventKind       what;         // Event type
        UInt32          message;      // Additional parametric information
        UInt32          when;         // Time event occurred
        Point           where;        // Mouse position at time of event
        EventModifiers  modifiers;    // State of keyboard modifier keys
    };
```

The QTML utility function `NativeEventToMacEvent` converts a Windows message into an equivalent QTML event:

```
int NativeEventToMacEvent
        (void          *winMsg,       // Windows message to be converted
         EventRecord   *macEvent)     // Equivalent Macintosh event
```

The first parameter points to a Windows `MSG` structure describing the message received by your window procedure; the second points to a QTML event record for the function to fill in to represent an equivalent event, if any. (A nonzero function result indicates that the conversion took place successfully; if the given message doesn't correspond to a Mac OS-style event, the function simply converts it to a **null event** and returns a zero result.)

The QuickTime function `MCIsPlayerEvent`

```
ComponentResult MCIsPlayerEvent
        (MovieController      mc,       // Movie controller
         const EventRecord    *e)       // Event to be processed
```

accepts a movie controller and an event record as parameters, determines whether the event is directed to the controller, and processes it as appropriate. This allows the movie controller to "run itself", handling all mouse and keyboard interactions with the user and displaying its movie on the screen accordingly. Even if the movie controller has no interest in the given event (for instance, if it's a null event), the controller receives some processing time to advance the presentation of the movie itself.

Although the function returns a result of type `ComponentResult` (equivalent to a long integer) to indicate whether the movie controller has processed the event, you should normally ignore this result and simply pass all messages through both `MCIsPlayerEvent` and your window procedure's normal message dispatch.

Listing 2-10 shows how to use the `NativeEventToMacEvent` and `MCIsPlayerEvent` functions to convert each message you receive to an event, then pass it to the window controller for action.

Listing 2-10 Displaying a movie

```
MovieController  theController;       // Movie controller for movie
LRESULT
    CALLBACK WinProc
        (HWND    thisWindow,          // Handle to window
         UINT    msgType,             // Message type
         WPARAM  wParam,              // Message-dependent parameter
         LPARAM  lParam)              // Message-dependent parameter
    {
        MSG          winMsg;          // Windows message structure
        EventRecord  qtmlEvt;         // Macintosh event record
        DWORD        msgPos;          // Mouse coordinates of message


        winMsg.hwnd = thisWindow;     // Window handle

        winMsg.message = msgType;     // Message type
        winMsg.wParam = wParam;       // Word-length parameter
        winMsg.lParam = lParam;       // Long-word parameter

        winMsg.time = GetMessageTime();  // Get time of message

        msgPos = GetMessagePos();        // Get mouse position
        winMsg.pt.x = LOWORD(msgPos);    // Extract x coordinate
        winMsg.pt.y = HIWORD(msgPos);    // Extract y coordinate

        NativeEventToMacEvent (&winMsg, &qtmlEvt); // Convert to event

        MCIsPlayerEvent (theController, &qtmlEvt); // Pass event to QuickTime

        switch ( msgType )               // Dispatch on message type
            {
                .                        // Handle message according to type
                .

            } /* end switch ( msgType ) */
    } /* end WinProc */
```

Resources

Mac OS **resources** are items of structured data that reside in files and can be read in on demand to help determine a program's behavior. Although Windows has the concept of resources as well, they're far less central to the system's software architecture than they are on the Mac OS platform.

Every Mac OS file consists of two separate **forks,** stored independently but logically joined under a single file name. The **data fork** consists of a single stream of data bytes intended to be read sequentially, and corresponds to what's generally considered a file on most other platforms. The **resource fork,** by contrast, contains a collection of individual resources that are accessed via a four-character **resource type** and an integer **resource ID.** For example, an icon to be displayed on the screen might be identified by resource type `'ICON'` and resource ID 1; the contents of a menu by type `'MENU'`, ID 128; the layout of a dialog box by type `'DLOG'`, ID 1000; and so forth.

Four-character codes like the ones that represent resource types are used on the Mac OS platform for a wide variety of other purposes as well. For example, every file is stamped with a four-character **file type** and a four-character **creator signature** identifying the application program to which the file belongs; these play an analogous role on the Mac OS platform to the three-character file-name extension in the DOS/Windows file system.

QuickTime uses four-character codes to identify such things as track types, media types, and component types. Internally, such codes are simply 32-bit long integers; at the source-language level, they are typically represented by a string of four characters enclosed in single quotation marks, such as `'abcd'`.

Because DOS/Windows files don't have a counterpart to the Macintosh resource fork, other mechanisms have to be adopted to accommodate resource information. For example, although QuickTime movie files use both forks on the Mac OS platform, those on Windows have only the equivalent of the data fork. One approach is to store only the contents of the data fork from the Mac OS movie file into the corresponding Windows movie file (extension .mov), while storing the resource fork into a companion file with extension .qtr ("QuickTime resources"). If a needed resource cannot be found in the .mov file, the QTML resource-handling routines will automatically look for a matching .qtr file and will attempt to locate the resource there. The drawback to this approach is that the user, when moving or copying a movie file from one place to another, must remember to move the matching resource file along with it. This is a nuisance to the user and is likely to lead to dissatisfaction with your application.

Fortunately, QuickTime supports another solution to the cross-platform resource problem. The QuickTime function `FlattenMovie` allows you to create a **single-fork movie file** with an empty resource fork and all of the resource data stored in the data fork instead. The resulting file can then be transported to Windows (or other platforms) without losing any of the movie's data. This is generally a better solution for cross-platform compatibility, since it requires the user to move one file instead of two.

In porting existing QuickTime applications from the Mac OS platform to Windows, the problem also arises of how to transport resources belonging to the application program itself. On the Mac OS platform, such resources normally reside in the resource fork of the application (`'APPL'`) file. A utility named RezWack, provided as part of the QuickTime 3 Software Development Kit for Windows, incorporates these resources from the resource fork of the Mac OS version into the executable (.exe) file of the Windows version. The QTML resource-management routines will correctly locate and read in the resources from the application's .exe file.

Windows Utility Routines

QuickTime includes a set of "glue" routines to help you write code that will run with both Mac OS and Windows, as discussed int his chapter.

File Pathnames

These routines convert Macintosh and Windows file paths into various other forms needed for multiplatform compatibility:

- `QTMLGetCanonicalPathName` takes a Windows file path and converts it into the canonical path to that file.
- `QTMLGetVolumeRootPath` takes a Windows path and returns the portion of it that points to the volume root.
- `FSSpecToNativePathName` converts a Macintosh `FSSpec` to a C string pathname for Windows.
- `NativePathNameToFSSpec` converts a C string file pathname for Windows to a Macintosh `FSSpec`.

Desktop Compatibility

These routines perform housekeeping tasks for windows, ports, graphics devices, and the Windows taskbar:

- `InitializeQHdr` initializes a Windows `QHdr` data structure for use by QuickTime.
- `TerminateQHdr` releases Windows-specific `QHdr` data.
- `ShowHideTaskBar` shows or hides the Windows taskbar.
- `IsTaskBarVisible` returns the current visibility state of the taskbar.
- `QTSetDDPrimarySurface` lets you set the primary surface for the DirectDraw object used by QuickTime.
- `QTGetDDObject` returns the DirectDraw object currently in use by QuickTime.
- `QTSetDDObject` sets the DirectDraw object currently in use by QuickTime.
- `GetPortHDC` returns a handle to the device context for a Windows grafport.
- `GetPortHBITMAP` returns a handle to the Windows `bitmap` associated with a grafport.
- `GetPortHPALETTE` returns a handle to the Windows palette associated with a grafport.
- `GetPortHFONT` returns a handle to the Windows font associated with a grafport.
- `UpdatePort` forces the update of a Windows port.

- CreatePortAssociation associates a graphics port (a data structure of type CGrafPort) with an onscreen native window.
- DestroyPortAssociation removes the graphics port associated with an onscreen window.
- GetGDeviceSurface returns the DirectDraw surface associated with a GWorld.

QuickTime Media Layer

These functions support the QuickTime Media Layer, which forms a connection between QuickTime and Windows processes:

- InitializeQTML initializes the QuickTime Media Layer.
- TerminateQTML terminates the QuickTime Media Layer.
- QTMLCreateMutex creates a synchronization object to facilitate mutually exclusive access to a data structure under Windows.
- QTMLDestroyMutex deallocates a synchronization object that was created by QTMLCreateMutex.
- QTMLGrabMutex confers ownership of a mutex created by QTMLCreateMutex.
- QTMLTryGrabMutex determines if you would be able to get immediate ownership of a mutex created by QTMLCreateMutex.
- QTMLReturnMutex releases ownership of a QTMLMutex object.
- QTMLCreateSyncVar creates a synchronization variable, used to provide guarded access to resources shared across threads and processes.
- QTMLDestroySyncVar releases ownership of a synchronization variable.
- QTMLTestAndSetSyncVar performs a one-shot atomic test and set operation of a QTMLSyncVar object.
- QTMLWaitAndSetSyncVar acquires the lock for a QTMLSyncVar object.
- QTMLResetSyncVar resets the lock for a QTMLSyncVar object.
- QTMLRegisterInterruptSafeThread registers a thread of execution that is allowed to make interrupt-safe calls.
- QTMLUnregisterInterruptSafeThread unregisters a thread of execution.
- QTMLYieldCPU yields time to other threads while your code is in a tight loop.
- QTMLYieldCPUTime yields time to other threads and specifies the sleep time while in a tight loop.
- QTMLGetWindowWndProc returns the WNDPROC specified by QTMLSetWindowWndProc, or NULL if no application-defined WNDPROC is set.
- QTMLSetWindowWndProc lets you specify an application-defined WNDPROC that QuickTime calls after it processes the message for the HWND.
- QTMLAcquireWindowList sets the mutex to the QuickTime windows list, so that it does not change until you call QTMLReleaseWindowList.
- QTMLReleaseWindowList releases the QuickTime windows list.

General Utilities

These general utilities convert handles and strings from one form to another:

- `GetPictFromDIB` creates a Macintosh QuickDraw `PicHandle` from a handle to a Windows DIB.
- `GetDIBFromPict` creates a handle to a DIB from a `PicHandle`.
- `NativeRegionToMacRegion` converts a Windows HRGN to a Mac region handle.
- `MacRegionToNativeRegion` converts a Mac region handle to a Windows HRGN.
- `NativeEventToMacEvent` converts Win32 messages to Macintosh events. (An earlier version, `WinEventToMacEvent`, is now a macro that calls `NativeEventToMacEvent`).
- `c2pstr` converts a C-formatted string to Pascal format in place.
- `p2cstr` converts a Pascal-formatted string to C format in place.

Sample Code

Sample code is available illustrating the use of QuickTime on the Windows platform. The code uses the Windows single document interface (SDI) to present a movie on the screen, allowing the user to control its display by manipulating a standard movie controller with the mouse. The sample also supports basic operations such as file saving and simple cut-and-paste editing. The sample is available at http://developer.apple.com/samplecode/simpleeditsdi.win/simpleeditsdi.win.dmg.

Document Revision History

This table describes the changes to *QuickTime Guide for Windows*.

Date	Notes
2006-01-10	Reformatted content and changed title from "QuickTime for Windows."
2002-09-17	New document that introduces Windows programming techniques for QuickTime.