QuickTime 3
for Windows Programmers

Apple Technical Publications

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

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manuals distributed with an Apple product, Apple will replace the manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged manuals for as long as the software is included in Apple’s Media Exchange program. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD “AS IS,” AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Tables and Listings

CHAPTER 1

What Is QuickTime 3 for Windows?

This manual is a programmer’s introduction to QuickTime, version 3, for the Windows platform. QuickTime is Apple Computer, Inc.’s industry-standard software architecture for creating, editing, and presenting digital media on personal computers. Originally developed for the Mac OS platform,QuickTime is now available to developers for the 32-bit Windows 95 and Windows NT 4.0 platforms as well, via the QuickTime 3 Software Development Kit (SDK) for Windows.

If you are a Windows developer, the SDK 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.

The core of the QuickTime 3 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. The Macintosh Toolbox routines it supports are listed and described in *Mac OS for QuickTime Programmers.*

This DLL is intended only for QuickTime cross-platform support, not as a general tool for porting Macintosh code to Windows. For a complete list of QuickTime and Mac OS functions supported for Windows code, see the functions index in the QuickTime 3 online documentation.

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.

The purpose of this manual is to help you through that process. If your primary development background is on Windows, the book will introduce you to some of the basic Mac OS concepts that you'll need to understand in order to use QuickTime effectively. There are just a few of these, and they correspond pretty closely to ideas that you're already familiar with from Windows programming. Table 1-1 lists these basic 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

Please note, though, that this manual does *not* attempt to teach you all there is to know about QuickTime itself. That information is presented in the following books, all of which are included in both online and Adobe Acrobat (PDF) form with the QuickTime 3 Software Development Kit:

- *Inside Macintosh: QuickTime*
- *Inside Macintosh: QuickTime Components*
- *QuickTime 3 Reference*
- *Programming With QuickTime Sprites*
- *QuickTime Music Architecture*
- *Mac OS For QuickTime Programmers*
- *Mac OS Sound*
- *Programming With QiuickTime VR 2.1*
- *3D Graphics Programming With QuickDraw 3D 1.5.4*

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 book 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. Once you've seen how to do that much, you can consult the *Inside Macintosh* volumes and the *QuickTime 3 Reference* to learn how to accomplish more advanced operations such as creating and editing movies or developing new QuickTime components.

When you have mastered the basics of QuickTime programming, the other books listed above will help you explore the worlds of sprites, music and sound, virtual reality environments, and 3D graphics modeling, all of which are part of QuickTime for Windows.

CHAPTER 2

QuickTime 3 for Windows: A Quick Start

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. In the next chapter, we'll discuss each of these steps in turn, along with the related QTML concepts that you need to understand in order to use QuickTime effectively.

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
                •
                •

        NativeEventToMacEvent (&winMsg, &qtmlEvent);// Convert message to a QTML event

        MCIsPlayerEvent (theMC, (const EventRecord *) &qtmlEvent);
                                                    // Pass event to movie controller
        switch ( message )
            {
                case WM_CREATE:
                    CreatePortAssociation (hWnd, NULL); // Register window with QTML
                    break;

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

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

                                if ( GetFile (movieFile) )  // Get file name from user
                                    OpenMovie (hWnd, movieFile);    // Open the movie
                                break;

                                    •
                                    •

                            default:
                                return DefWindowProc (hWnd, message,
                                                      wParam, lParam);
                        }  /* end switch ( wmId ) */

                    break;

                case WM_CLOSE:
                    DestroyPortAssociation (hWnd);      // Unregister window with QTML
                    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];

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

        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 */
```

Using QuickTime 3 for Windows

This chapter introduces the basic QTML routines for building QuickTime capabilities into your Windows application, along with the underlying QTML concepts they're based on. See the relevant volumes of *Inside Macintosh: QuickTime* and *QuickTime Components* and *QuickTime 3 Reference* to learn more about QuickTime and its more advanced capabilities.

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:

`kInitQTMLUseDefault`	Use standard behavior.
`kInitQTMLUseGDIFlag`	Use the Windows Graphics Device Interface (GDI) for all drawing.
`kInitQTMLNoSoundFlag`	Don't initialize the Sound Manager; disable sound for all movies.

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.

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 3-1 shows how these initialization and termination calls fit into the structure of a typical `WinMain` routine.

Listing 3-1 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

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

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

///////////////////////////////////////////////////////////////////////////////
```

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

        return (msg.wParam);

    }  /* end WinMain */
```

Graphics Ports

Because of its Mac OS origins, QuickTime uses the QuickDraw graphics routines—the Macintosh counterpart to the Windows Graphics Device Interface, or GDI—to draw to the screen. 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`. See *Mac OS For QuickTime Programmers* for a complete description of this data structure and its contents.

Note
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.

Note
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`. See Chapter 5, “Redefined API Names,” for a complete list of such name conflicts. ◆

As we’ll see, graphics ports are intimately 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. We’ll see how to do this in the next section.

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 contents of this structure are fully described in *Mac OS For QuickTime Programmers.*

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 3-2.

Listing 3-2 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 3-3 shows an example.

Listing 3-3 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

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

                        .
                        .

            }  /* end switch ( msgType ) */

                .
                .

    }  /* end WinProc */
```

Graphics Worlds

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** (described in detail in *Mac OS For QuickTime Programmers.*

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
```

You can read all about this function and its parameters in *Mac OS For QuickTime Programmers.* What's relevant here is that 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.

Note, however, that 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`. All other parameters are as described in *Mac OS For QuickTime Programmers,* with the exception of the `pixelFormat` parameter that replaces `pixelDepth` in the original `NewGWorld` function. Valid settings for this parameter are as follows:

```
    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 3-4 shows an example.

Listing 3-4 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)
```

Note
This is useful for drawing offscreen because you can create GWorlds and then direct the movie to draw them there. ◆

Like `SetGWorld`, `SetMovieGWorld` 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.

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. Ordinarily, 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 3-5).

Listing 3-5 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 does the job:

```
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 3-6 shows how to use this function to mediate between the Windows common dialog box and the QTML `OpenMovieFile` function.

Note
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). ◆

▲ **WARNING**
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 3-6 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
        FSMakeFSSpec (0, 0L, pathName, &fileSpec);  // Make specification record

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

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

    }  /* 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 3-5) 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 3-7 illustrates how to combine these QuickTime calls to read a movie in from its movie file.

Listing 3-7 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
        .
        .
errCode = OpenMovieFile (&fileSpec, &theFile, fsRdPerm);    // Open the movie file
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. Movie controllers and the functions available for working with them are discussed fully in *Inside Macintosh: QuickTime Components* and *QuickTime 3 Reference.*

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` (as shown in the section "Movies and Movie Files"). 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 3-8, 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 3-8 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 3-9 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 3-9 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.

Note
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` (described in *Inside Macintosh: QuickTime* and *QuickTime 3 Reference*) 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 Functions

The utility functions described in this chapter constitute a set of routines, specific to Windows, that will help you with QuickTime programming on the Windows 95 and Windows NT platform.

The interfaces to these routines are through the header files `QuickDraw.h` and `QTML.h`.

Access to Windows Data Structures

The utility functions described in this section provide access to Windows data structures that QuickTime uses as part of its implementation.

CreatePortAssociation

`CreatePortAssociation` associates a graphics port (a data structure of type `GrafPort`) with an onscreen native window.

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

`theWnd` Native window to associate the `GrafPort` with. In Windows, this parameter represents the movie `HWND`.

`wStorage` A pointer to a window record used for window storage. If you specify `NIL`, this function will allocate storage for you.

`flags` Option flags:

`kQTMLNoIdleEvents`
: If you set this flag, QuickTime will not pass periodic idle messages to the `WndProc` associated with this window. In this case it is your responsibility to task any movies playing in this window. When this flag is not set, QuickTime makes sure the `WndProc` associated with the onscreen window gets periodic idle messages so your code can in turn idle any movie controllers contained within that window.

DISCUSSION

The `CreatePortAssociation` function associates a QuickDraw graphics port with an onscreen window. The graphics port provides a drawing context for QuickTime and QuickDraw. In addition, QuickTime hooks the native window, so that any window state changes are reflected in the associated graphics port. Before you dispose of the native window, call `DestroyPortAssociation`.

DestroyPortAssociation

`DestroyPortAssociation` removes the graphics port associated with an onscreen window.

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

`cgp` A pointer to the QuickDraw graphics port associated with a native window.

DISCUSSION

The `DestroyPortAssociation` function removes the graphics port associated with an onscreen native window. This association was established previously via the `CreatePortAssociation` call. The `DestroyPortAssociation` function unhooks the native window `WndProc` and deallocates and window storage. Call this function before you destroy the native window.

UpdatePort

The UpdatePort function forces the update of the port.

```
OSErr UpdatePort(GrafPtr port);
```

port	A Mac OS graphics port.

DISCUSSION

This routine updates the various fields of a graphics port from the current HWND settings. The port's visRgn, strucRgn, and bounds are updated.

GetHWNDPort

The GetHWNDPort function gets a Mac OS graphics port pointer for a Windows HWND window handle.

```
GrafPtr GetHWNDPort(void *theHWND);
```

theHWND	A window handle
function result	A pointer to a Mac OS GrafPort data structure.

GetPortHDC

The GetPortHDC function returns a Windows HDC.

```
void *GetPortHDC(GrafPtr port);
```

port	A Mac OS graphics port.
function result	A Windows HDC.

GetPortHBITMAP

The GetPortHBITMAP function returns a HBITMAP.

```
void *GetPortHBITMAP(GrafPtr port);
```

port	A Mac OS graphics port.
function result	A Windows HBITMAP.

DISCUSSION

Use this routine to get the HBITMAP object associated with an offscreen graphics world. This HBITMAP will be a DIBSECTION. Do not dispose of this HBITMAP.

GetPortHPALETTE

The GetPortHPALETTE function returns a HPALETTE.

```
void *GetPortHPALETTE(GrafPtr port);
```

port	A Mac OS graphics port.
function result	A Windows HPALETTE.

GetPortHFONT

The GetPortHFONT function returns a handle to the currently-selected Windows font.

```
void *GetPortHFONT(GrafPtr port);
```

port	A Mac OS graphics port.
function result	A Windows font.

QTGetDDObject

The QTGetDDObject function returns the Direct Draw object currently in use by QuickTime.

```
OSErr QTGetDDObject(void **lpDDObject);
```

lpDDObject Specifies the DirectDraw object.

DISCUSSION

This function is useful for developers who want to call DirectDraw methods directly.

QTSetDDObject

The QTSetDDObject function sets the DirectDraw object currently in use by QuickTime.

```
OSErr QTSetDDObject(void *lpNewDDObject);
```

lpNewDDObject Specifies the DirectDraw object.

This function is useful for developers who want to call DirectDraw methods directly.

QTSetDDPrimarySurface

The QTSetDDPrimarySurface function allows you to set the primary DirectDraw surface used by QuickTime.

```
OSErr QTSetDDPrimarySurface(void *lpNewDDSurface, unsigned long flags);
```

lpNewDDSurface
Contains a pointer to a DirectDraw surface.

`flags` Contains flags that control the set operation. The following flags are valid:

`kDDSurfaceLocked` If set, QuickTime won't attempt to lock the graphics device when blitting to the `PixMap`.

`kDDSurfaceStatic` If set, QuickTime assumes Windows on this device do not move.

DISCUSSION

This function is useful for multimedia developers who want to wrap QuickTime around surfaces they have already created.

InitializeQHdr

`InitializeQHdr` initializes a Windows `QHdr` data structure for use by the Toolbox.

```
void InitializeQHdr(QHdr *qhdr);
```

`qhdr` A pointer to a `QHdr` record.

DISCUSSION

The `InitializeQHdr` function initializes the various fields of the Windows queue header to startup values and associates a mutex with the queue to provide safe access via the Toolbox `Enqueue` and `Dequeue` routines. The mutex identifier is stored in the `MutexID` field of the `QHdr`. Your application or component is not required to manage this mutex; the Toolbox functions `Enqueue` and `Dequeue` will handle this for you. A `QHdr` structure is typically used by QuickTime image decompressor components to manage frame queues. Once you are done with the queue, call `TerminateQHdr` to free the mutex.

TerminateQHdr

`InitializeQHdr` terminates a Windows `QHdr` data structure.

```
void TerminateQHdr (QHdr *qhdr);
```

`qhdr` A pointer to a `QHdr` record.

DISCUSSION

The `TerminateQHdr` function deallocates the data structures created by `InitializeQHdr`.

IsTaskBarVisible

The `IsTaskBarVisible` routine returns the current visibility state of the taskbar.

```
Boolean IsTaskBarVisible(void);
```

function result If the taskbar is visible, the function returns `true`.

ShowHideTaskBar

The `ShowHideTaskBar` routine shows or hides the Windows taskbar.

```
void ShowHideTaskBar(Boolean showIt);
```

`showIt` If `true`, show the taskbar. Otherwise, hide the taskbar.

This call can be used to hide the taskbar during full-screen movie playback.

QTMLAcquireWindowList

The `QTMLAcquireWindowList` function acquires exclusive access to the global list of (Macintosh-style) windows, so that the list will not change until you call `QTMLReleaseWindowList`.

```
void QTMLAcquireWindowList( void );
```

DISCUSSION

If you want to call the `LMGetWindowList` function or the `FrontWindow` function and then proceed to walk down the `next` pointers, you need to protect yourself against another thread modifying the list while you walk. Call the `QTMLAcquireWindowList` function before and the `QTMLReleaseWindowList` routine after.

QTMLReleaseWindowList

The `QTMLReleaseWindowList` function allows other threads to modify the global list of (Macintosh-style) windows again.

```
void QTMLReleaseWindowList( void );
```

Data Conversion

The utility functions described in this section map between data formats used by Windows and those used by QuickTime.

NativeEventToMacEvent

NativeEventToMacEvent converts Win32 messages to Macintosh events.

```
long NativeEventToMacEvent(void *nativeEvent, EventRecord *macEvent);
```

macEvent	A pointer to Macintosh EventRecord structure to be filled in.

DISCUSSION

Use this function from a WndProc to convert a message structure to an equivalent Macintosh event record. NativeEventToMacEvent translates Win32 message types into Macintosh event types and fills in the various other EventRecord fields based on the source Win32 MSG. Typically, when your application hosts a movie controller, it should call NativeEventToMacEvent to translate a Win32 MSG to an EventRecord, and then pass the resulting EventRecord to MCIsPlayerEvent for processing. This function returns noErr if the translation succeeded. You should never call this function and then exit early from your WndProc without calling DefWindowProc or returning an appropriate result code from your WndProc.

GetPictFromDIB

You use the GetPictFromDIB function to create a QuickDraw PicHandle from a handle to a DIB.

```
PicHandle GetPictFromDIB (void *h);
```

h	A handle to a DIB

DESCRIPTION

The GetPictFromDIB function returns a PicHandle when passed a handle to a DIB. The caller is responsible for releasing the memory of the PicHandle. You call the function KillPicture to release the memory of PicHandle.

Note that this function does not work for HBITMAP.

The format of the DIB handle is the same as returned by GetClipboardData with CF_DIB.

GetDIBFromPict

You use the GetDIBFromPict function to create a handle to a DIB from a QuickDraw PicHandle.

```
void *GetDIBFromPict (PicHandle hPict);
```

hPict	Specifies a handle to a Mac OS-style PICT.
function result	A handle to a DIB.

DESCRIPTION

The GetDIBFromPict function returns a global handle to a DIB when passed a PicHandle. The caller is responsible for releasing the memory of the DIB handle. You call the function GlobalFree to release the memory of the DIB handle.

Note that the DIB handle is not the same as HBITMAP.

NativeRegionToMacRegion

The NativeRegionToMacRegion function converts a Windows HRGN to a Macintosh region handle.

```
RgnHandle NativeRegionToMacRegion(void *nativeRegion)
```

nativeRegion	A Windows HRGN.
function result	A Macintosh region handle.

DISCUSSION

The RgnHandle should be disposed of by the caller.

MacRegionToNativeRegion

The `MacRegionToNativeRegion` function converts a Macintosh region handle to a Windows `HRGN`.

```
void *MacRegionToNativeRegion(RgnHandle macRegion);
```

`macRegion` A Macintosh region handle.

function result A Windows `HRGN`.

FSSpecToNativePathName

The `FSSpecToNativePathName` function extracts the native pathname from an `FSSpec`.

```
OSErr FSSpecToNativePathName(FSSpec *inFile, char *outName,
                             unsigned long outLen, long flags);
```

`inFile` Contains a pointer to a `FSSpec`.

`outName` Contains a pointer to a buffer to hold a C string.

`outLen` Specifies the maximum size of the buffer in bytes, including thestring terminator.

`flags` Contains flags that control the conversion. The following flags are valid:

`kFullNativePath`
This indicates that the full pathname should be returned.

`kFileNameOnly`
Only the part of the pathname corresponding to the file should be returned. This might be useful to return a string for a window’s title.

`kDirectoryPathOnly`
The full pathname up to and including the enclosing directory but not the filename should

be returned. This can be useful to get a path for the enclosing directory that might be used to find related files in the same directory.

As an example, consider the following Windows full path:

```
D:\Media\My Movies\Really Cool Movies\Tasty Fish.mov
```

If you have an FSSpec for this path, you can extract either the whole path or portions of the path using one of the above flags. For the above path and each flag, the resulting strings are:

Using kFullNativePath gives

```
D:\Media\My Movies\Really Cool Movies\Tasty Fish.mov
```

Using kFileNameOnly gives

```
Tasty Fish.mov
```

Using kDirectoryPathOnly gives

```
D:\Media\My Movies\Really Cool Movies
```

DISCUSSION

Sometimes, developers may need to convert a FSSpec returned by QuickTime APIs to a native pathname to be passed into the current operating system. The FSSpecToNativePathName function accepts an FSSpec and fills in the buffer pathname whose size is pathnameMaxBufferSize with the equivalent pathname string. This size must also include the size necessary to hold the string terminator.

NativePathNameToFSSpec

The NativePathNameToFSSpec function, given a native pathname, returns an FSSpec for that file.

```
OSErr NativePathNameToFSSpec(char *inName, FSSpec *outFile, long flags);
```

`inName` Contains a pointer to the native pathname.

`outFile` Contains a pointer to `FSSpec`.

`flags` Contains flags that control the conversion.

DISCUSSION

Given a C string pathname from the operating system, this routine updates the `FSSpec` of `outFile` to describe the same file. There are no flags currently defined, so you should pass 0. If the file does not currently exist, the error `fnfErr` is returned, but the resulting `FSSpec` is still valid for creating the file.

QTMLGetCanonicalPathName

The `QTMLGetCanonicalPathName` routine takes a native file path and returns the one canonical path to that file.

```
OSErr QTMLGetCanonicalPathName(char *inName, char *outName,
                     unsigned long outLen);
```

`inName` Specifies the input path.

`outName` Specifies where the routine puts the canonical path.

`outLen` Specifies the length of the `outName` buffer, so that the routine knows not to write past the end of your buffer.

DISCUSSION

This routine takes a native file path and returns the one canonical path to that file.

Some of the tasks performed by this routine include:

- removing all ".."s from the path
- converting all short (8.3) names back to their long name
- restoring the correct capitalization

For example, if you have a file with the following path

```
c:\Program Files\Some Product\test.mov
```

and the 8.3 path happens to be:

```
c:\PROGRA~1\SOMEPR~1\test.mov
```

you can pass any of the following paths to `QTMLGetCanonicalPathName`

```
c:\Some other folder\..\program FILES\another
program\..\somepr~1\TeSt.MoV

C:\proGra~1\Some product\..\SOMEPR~1\..\..\program files\some
product\test.mov

C:\PROGRA~1\SOMEPR~1\TEST.MOV
```

and it will always return

```
c:\Program Files\Some Product\test.mov
```

DISCUSSION

There is a one-to-one mapping between canonical pathnames and files. In other words, you can determine if two paths point to the same file by canonicalizing both paths, and then doing a string compare.

This routine also works for universal naming convention (UNC) paths. These paths are of the form:

```
\\my_server\shared_folder\another folder\test.mov
```

QTMLGetVolumeRootPath

The `QTMLGetVolumeRootPath` routine takes a Windows path and returns that portion of it which points to the volume root.

```
OSErr QTMLGetVolumeRootPath(char *fullPath, char * volumeRootPath,
                     unsigned long volumeRootLen);
```

`fullPath` Specifies the path being passed in.

`volumeRootPath`
: Specifies where this routine writes the volume root path.

`volumeRootLen`
: Specifies the length of the `volumeRootPath` buffer, so the routine knows not to write past the end of your buffer.

DISCUSSION

This routine works in the following way. If you pass in

```
c:\some folder\test.mov
```

it will return `c:\`

and if you pass in

```
\\my_server\shared_folder\mystuff\test.mov
```

it will return

```
\\my_server\shared_folder\
```

This is useful when you need to call Windows routines, such as `GetVolumeInformation`, which take a volume root path as an argument.

QTML Compatibility

The utility functions described in this section implement the QuickTime Media Layer (QTML) and perform miscellaneous tasks to make Windows programs compatible with QuickTime.

InitializeQTML

`InitializeQTML` initializes the QuickTime Media Layer.

```
OSErr InitializeQTML(long flags);
```

`flags` Option flags:

`kInitializeQTMLNoSoundFlag`

If this flag is set, the Sound Manager is not initialized and therefore no sound APIs will be supported during the session. Use this flag only if no sound support is needed.

`kInitializeQTMLUseGDIFlag`

If this flag is set, neither `DirectDraw` nor DCI services will be used for onscreen graphics support. When this flag is not set, QuickTime will try to use `DirectDraw` and then DCI to support direct-to-surface graphics support as well as take advantage of any hardware acceleration provided by these services. You should normally not set this flag.

DISCUSSION

Use `InitializeQTML` to initialize a QTML session, before calling `EnterMovies`. You should not make this call from a QuickTime component such as an image decompressor; it is provided only for host applications.

TerminateQTML

`TerminateQTML` terminates the QuickTime Media Layer.

```
void TerminateQTML(void);
```

DISCUSSION

Use `TerminateQTML` to terminate a QTML session after calling `ExitMovies`. You should not make this call from a QuickTime component, such as an image decompressor; it is provided only for host applications.

QTMLCreateMutex

QTMLCreateMutex creates a synchronization object to facilitate mutually exclusive access to a Windows data structure.

```
QTMLMutex QTMLCreateMutex(void);
```

function result A mutex object.

DISCUSSION

The QTMLCreateMutex function creates a mutex object for guarded access to data structures and routines that require mutually exclusive access. In a multithreaded preemptive environment, such as Windows NT, you can use the various mutex utility functions such as QTMLGrabMutex to protect a shared resource from simultaneous access by multiple threads or processes. Mutex objects are used throughout QTML to provide such protection.

QTMLDestroyMutex

QTMLDestroyMutex deallocates a synchronization object created by the QTMLCreateMutex function.

```
void QTMLDestroyMutex(QTMLMutex theMutex);
```

theMutex A mutex object.

DISCUSSION

Call the QTMLDestroyMutex function to deallocate the mutex object created by QTMLCreateMutex.

QTMLGrabMutex

QTMLGrabMutex confers ownership of a mutex created by the QTMLCreateMutex function.

```
void QTMLGrabMutex(QTMLMutex theMutex);
```

theMutex	A mutex object.

DISCUSSION

Call the QTMLGrabMutex function when you require exclusive ownership of the resource guarded by the mutex. This function will return when you have gained this ownership. In the case where another thread or process holds the mutex, this function waits until that process or thread relinquishes control. If you need to determine if you can grab the mutex, without actually grabbing it, call QTMLTryGrabMutex.

QTMLTryGrabMutex

QTMLTryGrabMutex determines if you would be able to get immediate ownership of a mutex created by QTMLCreateMutex.

```
Boolean QTMLTryGrabMutex (QTMLMutex theMutex);
```

theMutex	A mutex object.

DISCUSSION

Call the QTMLTryGrabMutex function when you need to preflight a QTMLGrabMutex call. It returns true if you are able to immediately grab the mutex, via the QTMLGrabMutex call, without having to wait. Under normal circumstances, you should not need to make this call.

QTMLReturnMutex

QTMLReturnMutex releases ownership of a QTMLMutex object.

```
void QTMLReturnMutex (QTMLMutex theMutex);
```

theMutex A mutex object.

DISCUSSION

Call the QTMLReturnMutex function to balance the call to QTMLGrabMutex when you are ready to relinquish control of the mutex and corresponding shared resource. By making this call you allow other processes or threads waiting for the release of this mutex to gain access.

QTMLCreateSyncVar

QTMLCreateSyncVar creates a synchronization variable, used to provide guarded access to resources shared across threads and processes.

```
QTMLSyncVarPtr QTMLCreateSyncVar(void);
```

function result A pointer to a synchronization variable.

DISCUSSION

Call the QTMLCreateSyncVar function to create a synchronization variable that allows for mutually-exclusive access to resources. The synchronization variable routines use atomic tests to ensure that the portions of the routines that perform the testing cannot be interrupted during the test.

QTMLDestroySyncVar

`QTMLDestroySyncVar` releases ownership of a synchronization variable.

```
void QTMLDestroySyncVar(QTMLSyncVarPtr p);
```

`p`	A pointer to a synchronization variable.

DISCUSSION

Call the `QTMLDestroySyncVar` function to deallocate the `QTMLSyncVar` object created by `QTMLCreateSyncVar`.

QTMLTestAndSetSyncVar

`QTMLTestAndSetSyncVar` performs a one-shot atomic test and set operation of a `QTMLSyncVar` object.

```
long QTMLTestAndSetSyncVar(QTMLSyncVarPtr p);
```

`p`	A pointer to a synchronization variable.
function result	0 if successful.

DISCUSSION

Call the `QTMLTestAndSetSyncVar` function to perform a single test and set operation on the `QTMLSyncVar` object. The function returns 0 if you have acquired the lock.

QTMLWaitAndSetSyncVar

`QTMLWaitAndSetSyncVar` acquires the lock for a `QTMLSyncVar` object,

```
void QTMLWaitAndSetSyncVar(QTMLSyncVarPtr p);
```

`p`	A pointer to a synchronization variable.

DISCUSSION

Call the `QTMLWaitAndSetSyncVar` function to acquire the lock corresponding to a `QTMLSyncVar` object. This function will wait, yielding CPU time to other threads, until the lock is acquired.

QTMLResetSyncVar

`QTMLResetSyncVar` reset the lock for a `QTMLSyncVar` object.

```
void QTMLResetSyncVar(QTMLSyncVarPtr p);
```

`p`	A pointer to a synchronization variable.

DISCUSSION

Call the `QTMLResetSyncVar` function to relinquish the lock obtained from a previous call to `QTMLWaitAndSetSyncVar`.

QTMLRegisterInterruptSafeThread

`QTMLRegisterInterruptSafeThread` registers a thread of execution that is allowed to make interrupt-safe calls.

```
void QTMLRegisterInterruptSafeThread(unsigned long threadID, void *info);
```

`threadID`	Thread ID of the calling thread. This value is obtained by calling the Win32 `GetCurrentThreadId` function.
`info`	Thread information. This value is obtained by calling the Win32 `GetCurrentThread` function.

DISCUSSION

The QTML function dispatcher includes a mechanism that prevents not only the Toolbox from getting reentered but also allows certain APIs to be callable at interrupt time. On the Macintosh, these calls are listed in *Inside Macintosh,* and require that the calling code not allocate, move, or purge memory. On Windows, threads that emulate interrupt handlers need to register with QTML, by calling the `QTMLRegisterInterruptSafeThread` function, so that API calls made from this thread are not blocked. You should make this call at the top of your thread main routine.

QTMLUnregisterInterruptSafeThread

`QTMLUnregisterInterruptSafeThread` unregisters a thread of execution.

```
void QTMLUnregisterInterruptSafeThread(unsigned long threadID);
```

`threadID` Thread ID of the calling thread. This value is obtained by calling the Win32 `GetCurrentThreadId` function.

DISCUSSION

Use the `QTMLRegisterInterruptSafeThread` function to unregister an interrupt safe thread previously registered by the `QTMLRegisterInterruptSafeThread` function. You should make this call at the bottom of your thread main routine, just before the exit.

QTMLYieldCPU

`QTMLYieldCPU` yields time to other threads while your code is in a tight loop.

```
void QTMLYieldCPU(void);
```

DISCUSSION

Use the QTMLYieldCPU function from within tight loops to yield time to other threads. Using this function is similar to calling SystemTask from within a Macintosh event loop.

QTMLYieldCPUTime

QTMLYieldCPUTime yields time to other threads and specifies the sleep time while in a tight loop.

```
void QTMLYieldCPUTime(long milliSecsToSleep, unsigned long flags);
```

milliSecsToSleep
Number of milliseconds to sleep before returning to the caller.

flags
Option flags:

kQTMLHandlePortEvents
If this flag is set, QTML will call the Win32 functions PeekMessage, TranslateMessage, and DispatchMessage to process Win32 messages while in tight spin loops.

DISCUSSION

Use the QTMLYieldCPUTime function from within tight loops to yield time to other threads. This function differs from QTMLYieldCPU in that you can specify the time to sleep as well as optionally have QTML process Win32 messages while waiting for the yield time to expire.

QTMLSetWindowWndProc

The QTMLSetWindowWndProc routine allows you to specify an application-defined window procedure (WNDPROC) which is called by QTML after QTML processes the message for the HWND.

```
void QTMLSetWindowWndProc(WindowPtr wPtr, void *windowProc);
```

wPtr — Specifies the Macintosh window to hook. This must have been created via NewCWindow, NewWindow, or as a result of calling CreatePortAssociation on a native HWND.

windowProc — A Windows WNDPROC procedure. For a detailed description of the WNDPROC procedure, check your Win32 documentation.

DISCUSSION

The QTMLSetWindowWndProc routine is useful if you want to perform some special Windows processing of the native messages that Windows sends to your WindowPtr.

QTMLGetWindowWndProc

The QTMLGetWindowWndProc routine returns the WNDPROC previously specified in QTMLSetWindowWndProc. It returns NULL if no application-defined WNDPROC is set.

```
void *QTMLGetWindowWndProc(WindowPtr);
```

wPtr — Specifies the Macintosh window to hook. This must have been created via NewCWindow, NewWindow, or as a result of calling CreatePortAssociation on a native HWND.

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 5-1, 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 5-1 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

Original Macintosh API name	Mapped name
FrameRgn	MacFrameRgn
GetClassInfo	MacGetClassInfo
GetCurrentThread	MacGetCurrentThread
GetCursor	MacGetCursor
GetDoubleClickTime	MacGetDoubleClickTime
GetFileSize	MacGetFileSize
GetItem	MacGetItem
GetMenu	MacGetMenu
GetNextWindow	MacGetNextWindow
GetParent	MacGetParent
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

Original Macintosh API name	Mapped name
SetCursor	MacSetCursor
SetItem	MacSetItem
SetPort	MacSetPort
SetRect	MacSetRect
SetRectRgn	MacSetRectRgn
ShowCursor	MacShowCursor
ShowWindow	MacShowWindow
StartSound	MacStartSound
StopSound	MacStopSound
TokenType	MacTokenType
UnionRect	MacUnionRect
UnionRgn	MacUnionRgn
XorRgn	MacXorRgn

Conversion From Earlier Versions

Converting an existing Windows program from earlier versions of QuickTime to QuickTime 3 is relatively simple, but there are a few changes that you should be aware of. These include:

- The calls for initializing and terminating the QuickTime Media Layer are now `InitializeQTML` and `TerminateQTML` instead of `QTInitialize` and `QTTerminate`, and the meaning of the initialization routine's parameter has changed; see "Initializing and Terminating QTML and QuickTime" (page 3-17) for more information. Note, however, that the initialization and termination calls for QuickTime itself, `EnterMovies` and `ExitMovies`, remain the same as before.
- QuickTime calls now use the Mac OS data types `Point` and `Rect` to represent points and rectangles, rather than the corresponding Windows types `POINT` and `RECT`. This is because the QuickTime routines expect the coordinates to be specified as 16-bit integers instead of 32 bits, as they are in Windows 95 and Windows NT. For example, the QuickTime routine `GetMovieBox` is now defined as

  ```
  void
      GetMovieBox
          (Movie   theMovie,
           Rect    *boxRect)
  ```

  instead of

  ```
  void
      GetMovieBox
          (Movie    mMovie,
           LPRECT   lprcMovieRect)
  ```

  as in earlier versions.
- QuickTime routines that formerly accepted a Windows window handle (`HWND`) as a parameter now implicitly use the current QTML graphics port

instead, as discussed under “Graphics Ports” (page 3-20). For example, the function NewMovieController now takes only three parameters

```
ComponentInstance
    NewMovieController
        (Movie        theMovie,
         const Rect  *movieRect,
         long         someFlags)
```

instead of four. To obtain the port corresponding to a window, you must first register the window with QTML by calling CreatePortAssociation (page 3-22), then use GetHWNDPort (page 3-23) to get the port pointer. Remember to deregister the window with DestroyPortAssociation (page 3-23) before destroying the window.

- The QuickTime call for driving a movie controller is now MCIsPlayerEvent instead of MCIsPlayerMessage; see “Movie Controllers” (page 3-35) for details.
- As discussed under “File Selection Dialogs” (page 3-30), the QuickTime function OpenMovieFile now accepts a Mac OS file-system specification record (FSSpec) identifying the file to be opened, instead of a string containing the file name.
- QuickTime routines that operate on movie files, such as NewMovieFromFile, now use a Mac OS-style file reference number to identify the file instead of a Windows file reference.
- The QuickTime call DereferenceHandle is no longer necessary with QuickTime 3 .

CHAPTER 7

Example Program

Listing 7-1 shows a simple but complete application program illustrating the use of QuickTime on the Windows platform. The program 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 program also supports basic operations such as file saving and simple cut-and-paste editing. The code shown here is adapted from one of the sample programs provided as part of the QuickTime 3 Software Development Kit for Windows.

Listing 7-1 Simple movie player

```
////////////////////////////////////////////////////////////////////////////////////////
//
//  SimpleEditSDI
//  Written by Keith Gurganus
//
//  A single-document-interface (SDI) application that plays a movie with QuickTime.
//  This program is part of the QuickTime sample source code and is provided as is.
//
//  Copyright:© 1997 by Apple Computer, Inc., all rights reserved.
//
////////////////////////////////////////////////////////////////////////////////////////


#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <windows.h>
#include "QTML.h"
#include "Movies.h"
```

```
/ Resource identifiers

#define   IDM_NEW              100
#define   IDM_OPEN             101
#define   IDM_SAVE             102
#define   IDM_SAVEAS           103
#define   IDM_PRINT            104
#define   IDM_PRINTSETUP       105
#define   IDM_EXIT             106

#define   IDM_UNDO             200
#define   IDM_CUT              201
#define   IDM_COPY             202
#define   IDM_PASTE            203
#define   IDM_LINK             204
#define   IDM_LINKS            205

#define   IDM_HELPCONTENTS     300
#define   IDM_HELPSEARCH       301
#define   IDM_HELPHELP         302
#define   IDM_ABOUT            303
#define   IDM_HELPTOPICS       304

#define   IDC_STATIC            -1

#define   DLG_VERFIRST         400
#define   IDC_COMPANY          DLG_VERFIRST
#define   IDC_FILEDESC         DLG_VERFIRST+1
#define   IDC_PRODVER          DLG_VERFIRST+2
#define   IDC_COPYRIGHT        DLG_VERFIRST+3
#define   IDC_OSVERSION        DLG_VERFIRST+4
#define   IDC_TRADEMARK        DLG_VERFIRST+5
#define   DLG_VERLAST          DLG_VERFIRST+5
#define   IDC_LABEL            DLG_VERLAST+1

#define   IDR_ACCELSIMPLESDI   128
#define   IDR_SIMPLESDI        128
#define   IDR_SMALL            129
#define   IDR_WIN95            131
#define   IDD_ABOUTBOX         132
#define   IDI_BIG              139
```

```
#define   APPNAME   'SimpleEditSDI'

// Macros to determine appropriate code paths

#if defined (WIN32)
    #define IS_WIN32 TRUE
#else
    #define IS_WIN32 FALSE
#endif

#define   IS_NT       IS_WIN32 && (BOOL)(GetVersion() < 0x80000000)
#define   IS_WIN32S   IS_WIN32 && (BOOL)(!(IS_NT) && (LOBYTE(LOWORD(GetVersion()))<4))
#define   IS_WIN95    (BOOL)(!(IS_NT) && !(IS_WIN32S)) && IS_WIN32

// Data type

    typedef struct
        {
            char                filename[255];
            Movie               theMovie;
            MovieController     theMC;
            Boolean             movieOpened;
            HWND                theHwnd;

        }  MovieStuff;

// Global variables

    HINSTANCE    hInst;                              // Current instance
    char         szAppName[] = APPNAME;              // Name of this application
    char         szTitle[]   = APPNAME;              // Title bar text
    MovieStuff   gMovieStuff;                        // Movie structure




// Function prototypes

    BOOL
        InitApplication
            (HINSTANCE   hInstance);
```

```
ATOM
    MyRegisterClass
        (CONST WNDCLASS  *lpwc);

BOOL
    InitInstance
        (HINSTANCE   hInstance,
         int         nCmdShow);

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

LRESULT CALLBACK
    About
        (HWND     hDlg,
         UINT     message,
         WPARAM   wParam,
         LPARAM   lParam);

BOOL
    GetFile
        (char  *fileName);

static UINT APIENTRY
    GenericHook
        (HWND     hWnd,
         UINT     uMsg,
         WPARAM   wParam,
         LPARAM   lParam);


BOOL
    OpenMovie
        (HWND          hwnd,
         MovieStuff   *movieStuff);
```

```
void
    CreateNewMovieController
        (HWND             hwnd,
         Movie            theMovie,
         MovieController  *theMC);

Boolean
    MCFilter
        (MovieController   mc,
         short             action,
         void             *params,
         long              refCon);

void
    GetMaxBounds
        (Rect  *maxRect);

void
    SetWindowTitle
        (HWND            hWnd,
         unsigned char  *theFullPath);

void
    GetFileNameFromFullPath
        (unsigned char  *theFullPath,
         unsigned char  *fileName);

void
    CloseMovie
        (MovieStuff   *movieStuff);

OSErr
    SaveMovie
        (MovieStuff   *movieStuff);

OSErr
    SaveAsMovie
        (MovieStuff   *movieStuff);
```

```
ComponentResult
    EditUndo
        (MovieController   mc);

ComponentResult
    EditCut
        (MovieController   mc);

ComponentResult
    EditCopy
        (MovieController   mc);

ComponentResult
    EditPaste
        (MovieController   mc);

ComponentResult
    EditClear
        (MovieController   mc);

ComponentResult
    EditSelectAll
        (Movie             movie,
         MovieController   mc);

void
    UpdateMenus
        (MovieStuff   movieStuff);
```

```
////////////////////////////////////////////////////////////////////////////////////////
//
//  WinMain
//
////////////////////////////////////////////////////////////////////////////////////////

int CALLBACK
    WinMain
        (HINSTANCE   hInstance,
         HINSTANCE   hPrevInstance,
         LPSTR       lpCmdLine,
         int         nCmdShow)

    {
        MSG      msg;
        HANDLE   hAccelTable;

        if ( !hPrevInstance )

            // Perform instance initialization.
                if ( !InitApplication(hInstance) )
                    return FALSE;

        // Initialize QuickTime Media Layer.
            InitializeQTML(0);

        // Initialize QuickTime.
            EnterMovies();

        // Perform application initialization.
            if ( !InitInstance(hInstance, nCmdShow) )
                return FALSE;

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

        // Main message loop:

            while (GetMessage(&msg, NULL, 0, 0))
                if ( !TranslateAccelerator(msg.hwnd, hAccelTable, &msg) )
                {
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);

                }  /* end if */

        // Terminate QuickTime.
            ExitMovies();

        // Terminate QuickTime Media Layer.
            TerminateQTML();

        return msg.wParam;

        // The following line is included to prevent
        // 'unused formal parameter' warnings.
            lpCmdLine;

    }  /* end WinMain */


////////////////////////////////////////////////////////////////////////////////////////
//
//  InitApplication
//
////////////////////////////////////////////////////////////////////////////////////////

BOOL
    InitApplication
        (HINSTANCE   hInstance)

    {
        WNDCLASS   wc;
        HWND       hwnd;

        // Win32 will always set hPrevInstance to NULL. We only want a single version
        // of this app to run at a time, so let’s check things a little closer.
            hwnd = FindWindow (szAppName, NULL);
            if ( hwnd )
                // We found another instance of ourself. Let’s defer to it:
                    {
                        if ( IsIconic(hwnd) )
                            ShowWindow(hwnd, SW_RESTORE);

                        SetForegroundWindow (hwnd);
                        return FALSE;

                    }  /* end if ( hwnd ) */

        // Fill in window class structure with parameters that describe
        // the main window.
            wc.style         = CS_HREDRAW | CS_VREDRAW;
            wc.lpfnWndProc   = (WNDPROC)WndProc;
            wc.cbClsExtra    = 0;
            wc.cbWndExtra    = 0;
            wc.hInstance     = hInstance;
            wc.hIcon         = LoadIcon (hInstance, MAKEINTRESOURCE(IDI_BIG));
            wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
            wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
            wc.lpszMenuName  = MAKEINTRESOURCE(IDR_SIMPLESDI);
            wc.lpszClassName = szAppName;

        // Register the window class and return success/failure code.
            if (IS_WIN95)
                return MyRegisterClass(&wc);
            else
                return RegisterClass(&wc);

    }  /* end InitApplication */


///////////////////////////////////////////////////////////////////////////////////////
//
//  MyRegisterClass
//
///////////////////////////////////////////////////////////////////////////////////////

ATOM
    MyRegisterClass
        (CONST WNDCLASS  *lpwc)
    {
        HANDLE       hMod;
        FARPROC      proc;
        WNDCLASSEX   wcex;

        hMod = GetModuleHandle ("USER32");
        if ( hMod != NULL )
            {
#if defined (UNICODE)
                proc = GetProcAddress (hMod, "RegisterClassExW");
#else
                proc = GetProcAddress (hMod, "RegisterClassExA");
#endif
                if ( proc != NULL )
                    {
                        // Copy elements from WNDCLASS structure.
                            wcex.style         = lpwc->style;
                            wcex.lpfnWndProc   = lpwc->lpfnWndProc;
                            wcex.cbClsExtra    = lpwc->cbClsExtra;
                            wcex.cbWndExtra    = lpwc->cbWndExtra;
                            wcex.hInstance     = lpwc->hInstance;
                            wcex.hIcon         = lpwc->hIcon;
                            wcex.hCursor       = lpwc->hCursor;
                            wcex.hbrBackground = lpwc->hbrBackground;
                            wcex.lpszMenuName  = lpwc->lpszMenuName;
                            wcex.lpszClassName = lpwc->lpszClassName;

                        // Add extra elements for Windows 95.
                            wcex.cbSize        = sizeof(WNDCLASSEX);
                            wcex.hIconSm       = LoadIcon(wcex.hInstance,
                                                          MAKEINTRESOURCE(IDR_SMALL));

                        // Return RegisterClassEx(&wcex).
                            return (*proc)(&wcex);

                    }  /* end if ( proc != NULL ) */

            }  /* end if ( hMod != NULL ) */

        return RegisterClass(lpwc);
    }  /* end MyRegisterClass */
```

```
////////////////////////////////////////////////////////////////////////////////////////
//
//  InitInstance
//
////////////////////////////////////////////////////////////////////////////////////////

BOOL
    InitInstance
        (HINSTANCE    hInstance,
         int          nCmdShow)

    {
        HWND   hWnd;

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

        // Create our window.
            hWnd = CreateWindow(szAppName,
                                szTitle,
                                WS_OVERLAPPEDWINDOW,
                                CW_USEDEFAULT, 0,
                                CW_USEDEFAULT, 0,
                                NULL,
                                NULL,
                                hInstance,
                                NULL);
            if ( !hWnd )
                return FALSE;

            ShowWindow   (hWnd, nCmdShow);
            UpdateWindow (hWnd);

            return TRUE;

    }  /* end InitInstance */


////////////////////////////////////////////////////////////////////////////////////////
//
//  WndProc
```

```
//
///////////////////////////////////////////////////////////////////////////////////////

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

    {
        int           wmId, wmEvent;
        PAINTSTRUCT   ps;
        HDC           hdc;

        if ( Hwnd2Wptr(hWnd) )
            {
                MSG           msg;
                EventRecord   macEvent;
                LONG          thePoints = GetMessagePos();

                msg.hwnd    = hWnd;
                msg.message = message;
                msg.wParam  = wParam;
                msg.lParam  = lParam;

                msg.time = GetMessageTime();

                msg.pt.x = LOWORD(thePoints);
                msg.pt.y = HIWORD(thePoints);

                // Convert the message to a QTML event.
                    NativeEventToMacEvent (&msg, &macEvent);

                // If we have a movie controller, pass the QTML event.
                    if ( gMovieStuff.theMC )
                        MCIsPlayerEvent (gMovieStuff.theMC,
                                         (const EventRecord *) &macEvent);

            }  /* end if ( Hwnd2Wptr(hWnd) ) */
```

```
switch ( message )
    {
        case WM_CREATE:
            memset (&gMovieStuff, 0, sizeof(MovieStuff));

            // Register this HWND with QTML.
            CreatePortAssociationEx (hWnd, NULL, kQTMLHandlePortEvents);
                gMovieStuff.theHwnd = hWnd;
                break;

        case WM_INITMENU:
            UpdateMenus (gMovieStuff);
            break;

        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);

            //Parse the menu selections.
                switch ( wmId )
                    {
                        case IDM_ABOUT:
                            DialogBox (hInst,
                                       MAKEINTRESOURCE(IDD_ABOUTBOX),
                                       hWnd,
                                       (DLGPROC)About);
                            break;

                        case IDM_EXIT:
                            CloseMovie (&gMovieStuff);
                            DestroyPortAssociationEx
                                    ( (CGrafPtr)Hwnd2Wptr(hWnd),
                                      kQTMLHandlePortEvents );
                            DestroyWindow (hWnd);
                            break;

                        case IDM_OPEN:
                            // Close any open movie.
                                CloseMovie (&gMovieStuff);
                            // Open a movie file.
                                if ( GetFile (gMovieStuff.filename) )
                                    {
                                        // Open the movie and size the window.
                                            OpenMovie (hWnd, &gMovieStuff);

                                        // Update the menus.
                                            UpdateMenus (gMovieStuff);

                                    }  /* end if */
                            break;

                        case IDM_SAVE:
                            SaveMovie (&gMovieStuff);
                            UpdateMenus (gMovieStuff);
                            break;

                        case IDM_SAVEAS:
                            SaveAsMovie (&gMovieStuff);
                            UpdateMenus (gMovieStuff);
                            break;

                        case IDM_UNDO:
                            EditUndo (gMovieStuff.theMC);
                            break;

                        case IDM_CUT:
                            EditCut (gMovieStuff.theMC);
                            break;

                        case IDM_COPY:
                            EditCopy (gMovieStuff.theMC);
                            break;

                        case IDM_PASTE:
                            EditPaste (gMovieStuff.theMC);
                            break;

                        case IDM_CLEAR:
                            EditClear (gMovieStuff.theMC);
                            break;
```

```
                            case IDM_SELECTALL:
                                EditSelectAll (gMovieStuff.theMovie,
                                               gMovieStuff.theMC);
                                break;

                            default:
                                return DefWindowProc (hWnd, message,
                                                      wParam, lParam);
                        }  /* end switch ( wmId ) */

                    break;

            case WM_PAINT:
                hdc = BeginPaint (hWnd, &ps);
                    // Add any additional drawing code here...
                EndPaint (hWnd, &ps);
                break;

            case WM_CLOSE:
                // Unregister the HWND with QTML.
                    DestroyPortAssociation( (CGrafPtr)Hwnd2Wptr(hWnd) );
                    break;

            case WM_DESTROY:
                PostQuitMessage (0);
                break;

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

        }  /* end switch ( message ) */

    return 0;

}  /* end WndProc */
```

```
/////////////////////////////////////////////////////////////////////////////////////
//
//  About
//
/////////////////////////////////////////////////////////////////////////////////////

LRESULT CALLBACK
    About
        (HWND     hDlg,
         UINT     message,
         WPARAM   wParam,
         LPARAM   lParam)

    {
        switch (message)
            {
                case WM_COMMAND:
                    if ( LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL )
                        {
                            EndDialog (hDlg, TRUE);
                            return TRUE;

                        }  /* end if */

                    break;

            }  /* end switch (message) */

        return FALSE;

    }  /* end About */

/////////////////////////////////////////////////////////////////////////////////////
//
//  GetFile
//
/////////////////////////////////////////////////////////////////////////////////////

BOOL
    GetFile
        (char  *fileName)
    {
        OPENFILENAME   ofn;

        memset (&ofn, 0, sizeof(OPENFILENAME));
        fileName[0] = '\0';

        ofn.lStructSize     = sizeof(OPENFILENAME);
        ofn.hwndOwner       = GetActiveWindow();
        ofn.lpstrFile       = (LPSTR)fileName;
        ofn.nMaxFile        = 255;
        ofn.lpstrFilter     = "QuickTime Movies (*.mov;*.avi)\0*.mov;*.avi\0All Files 
(*.*)\0*.*\0";
        ofn.nFilterIndex    = 1;
        ofn.lpstrInitialDir = NULL;

        if (USEEXPLORERSTYLE)
            ofn.Flags |= OFN_ENABLEHOOK | OFN_EXPLORER;
        else
            ofn.Flags |= OFN_ENABLEHOOK;

        ofn.lpfnHook = GenericHook;

        if ( GetOpenFileName(&ofn) )
            return TRUE;
        else
            return FALSE;

    }  /* end GetFile */

////////////////////////////////////////////////////////////////////////////////////////
//
//  GenericHook
//
////////////////////////////////////////////////////////////////////////////////////////

static UINT APIENTRY
    GenericHook
        (HWND     hWnd,
         UINT     uMsg,
         WPARAM   wParam,
         LPARAM   lParam)
```

```
{
    switch ( uMsg )
        {
            case WM_INITDIALOG:
                // Center window
                    {
                        Point   ptTopLeft;
                        RECT    rcWindow;
                        BOOL    retValue;
                        HWND    theWnd = hWnd;
                        RECT    rcDesktopWindow;
                        long    width;
                        long    height;

                        // If we are using Windows 95 or NT 4.0,
                        // use the new Explorer style.
                            if ( USEEXPLORERSTYLE )
                                theWnd = GetParent(hWnd);

                        GetWindowRect (theWnd, &rcWindow);
                        width  = rcWindow.right  - rcWindow.left;
                        height = rcWindow.bottom - rcWindow.top;

                        GetWindowRect (GetDesktopWindow(), &rcDesktopWindow);
                        ptTopLeft.h = (short)((rcDesktopWindow.right
                                                    + rcDesktopWindow.left) / 2
                                                - width / 2);
                        ptTopLeft.v = (short)((rcDesktopWindow.top
                                                    + rcDesktopWindow.bottom) / 3
                                                - height / 3);

                        retValue = SetWindowPos (theWnd,
                                                 0,
                                                 ptTopLeft.h,
                                                 ptTopLeft.v,
                                                 0,
                                                 0,
                                                 SWP_NOZORDER | SWP_NOSIZE);

                        return TRUE;
                        }  /* end case WM_INITDIALOG */

            }  /* end switch ( uMsg ) */

        return 0;

    }  /* end GenericHook */


///////////////////////////////////////////////////////////////////////////////////////
//
//  OpenMovie
//
///////////////////////////////////////////////////////////////////////////////////////

BOOL
    OpenMovie
        (HWND         hwnd,
         MovieStuff  *movieStuff)

    {
        BOOL  isMovieGood = FALSE;

        if ( strlen( (char*)movieStuff->filename ) != 0 )
            {
                OSErr    err;
                short    theFile = 0;
                long     controllerFlags = 0L;
                FSSpec   sfFile;
                short    movieResFile;
                char     theFullPath[255];

                // Make a copy of our full pathname.
                    strcpy (theFullPath, movieStuff->filename);

                // Convert to a Pascal string.
                    c2pstr( (char*)theFullPath );

                // Make an FSSpec.
                    FSMakeFSSpec (0, 0L, theFullPath, &sfFile);
```

```
// Set the port.
    SetGWorld ( (CGrafPtr)Hwnd2Wptr( (void *)hwnd ), nil);

// Open the movie file.
    err = OpenMovieFile (&sfFile, &movieResFile, fsRdPerm);
    if (err == noErr)
        {
            // Get the movie from the file.
                err = NewMovieFromFile (&movieStuff->theMovie,
                                        movieResFile,
                                        nil,
                                        nil,
                                        newMovieActive,
                                        nil);

            // Close the movie file.
                CloseMovieFile (movieResFile);
                if (err == noErr)
                    {
                        // Create a movie controller.
                            CreateNewMovieController (hwnd,
                                                    movieStuff->theMovie,
                                                    &movieStuff->theMC);

                        // Set flags.
                            movieStuff->movieOpened = TRUE;
                            isMovieGood = TRUE;

                        // Convert pathname back to a C string.
                            p2cstr ( (char*)theFullPath );

                        // Set window title.
                            SetWindowTitle (movieStuff->theHwnd,
                                                theFullPath);

                    }  /* end if (err == noErr) */

                else
                    theFullPath[0] = '\0';

        }  /* end if (err == noErr) */
```

```
                else
                    theFullPath[0] = '\0';

            }  /* end if ( strlen( (char*)movieStuff->filename ) != 0 ) */

        return isMovieGood;

    }  /* end OpenMovie */


/////////////////////////////////////////////////////////////////////////////////////////
//
//  CreateNewMovieController
//
/////////////////////////////////////////////////////////////////////////////////////////

void
    CreateNewMovieController
        (HWND              hwnd,
         Movie             theMovie,
         MovieController  *theMC)

    {
        Rect   bounds;
        Rect   maxBounds;
        long   controllerFlags;
        Rect   theMovieRect;

        // 0,0 movie coordinates.
            GetMovieBox (theMovie, &theMovieRect);
            MacOffsetRect (&theMovieRect, -theMovieRect.left, -theMovieRect.top);

        // Attach a movie controller.
            *theMC = NewMovieController (theMovie, &theMovieRect, mcTopLeftMovie);

        // Get the controller rect.
            MCGetControllerBoundsRect (*theMC, &bounds);

        // Enable editing.
            MCEnableEditing (*theMC, TRUE);
        // Tell the controller to attach a movie's CLUT to the window as appropriate.
            MCDoAction (*theMC, mcActionGetFlags, &controllerFlags);
            MCDoAction (*theMC, mcActionSetFlags,
                        (void *)(controllerFlags | mcFlagsUseWindowPalette) );

        // Allow the controller to accept keyboard events.
            MCDoAction (*theMC, mcActionSetKeysEnabled, (void *)TRUE);

        // Set the controller action filter.
            MCSetActionFilterWithRefCon (*theMC, MCFilter, (long)hwnd);

        // Set the grow box amount.
            GetMaxBounds (&maxBounds);
            MCDoAction (*theMC, mcActionSetGrowBoxBounds, &maxBounds);

        // Size our window.
            SizeWindow ((WindowPtr)Hwnd2Wptr(hwnd),
                        bounds.right,
                        bounds.bottom,
                        FALSE);

    }  /* end CreateNewMovieController */


////////////////////////////////////////////////////////////////////////////////////////
//
//  MCFilter
//
////////////////////////////////////////////////////////////////////////////////////////

Boolean
    MCFilter
        (MovieController   mc,
         short             action,
         void             *params,
         long              refCon)
    {
        if ( action == mcActionControllerSizeChanged )
            {
                Rect        bounds;
                WindowPtr   w;

                MCGetControllerBoundsRect (mc, &bounds);
                w = Hwnd2Wptr( (HWND)refCon );
                SizeWindow ( (WindowPtr)w, bounds.right, bounds.bottom, TRUE );

            }  /* end if ( action == mcActionControllerSizeChanged ) */

        return FALSE;

    }  /* end MCFilter */


/////////////////////////////////////////////////////////////////////////////////////////
//
//  GetMaxBounds
//
/////////////////////////////////////////////////////////////////////////////////////////

void
    GetMaxBounds
        (Rect  *maxRect)

    {
        RECT   deskRect;

        GetWindowRect (GetDesktopWindow(), &deskRect);
        OffsetRect (&deskRect, -deskRect.left, -deskRect.top);

        maxRect->top    = (short)deskRect.top;
        maxRect->bottom = (short)deskRect.bottom;
        maxRect->left   = (short)deskRect.left;
        maxRect->right  = (short)deskRect.right;

    }  /* end GetMaxBounds */
```

```
///////////////////////////////////////////////////////////////////////////////////////
//
//  SetWindowTitle
//
///////////////////////////////////////////////////////////////////////////////////////

void
    SetWindowTitle
        (HWND            hWnd,
         unsigned char  *theFullPath)

    {
        unsigned char   titleName[256];
        titleName[0] = '\0';

        GetFileNameFromFullPath ( theFullPath, (unsigned char *)&titleName );
        SetWindowText ( hWnd, (const char *)titleName );

    }  /* end SetWindowTitle */


///////////////////////////////////////////////////////////////////////////////////////
//
//  GetFileNameFromFullPath
//
///////////////////////////////////////////////////////////////////////////////////////

void
    GetFileNameFromFullPath
        (unsigned char  *theFullPath,
         unsigned char  *fileName)

    {
        int   i         =  0;
        int   j         = -1;
        int   stringLen =  0;

        stringLen = strlen( (char *)theFullPath );
        if ( stringLen > 0 )
            {
                while ( i < stringLen )
```

```
                    {
                        if ( theFullPath[i] == 0x5c || theFullPath[i] == '/' )
                            j = i;
                        i++;

                    }  /* end while ( i < stringLen ) */

                if ( j > -1 )
                    strcpy ( (char *)fileName, (char *)&theFullPath[j+1] );
                else
                    strcpy ( (char *)fileName, (char *)theFullPath );

            }  /* end if ( stringLen > 0 ) */

    }  /* end GetFileNameFromFullPath */


////////////////////////////////////////////////////////////////////////////////////////
//
//  CloseMovie
//
////////////////////////////////////////////////////////////////////////////////////////

void
    CloseMovie
        (MovieStuff  *movieStuff)

    {
        if ( movieStuff->movieOpened )
            {
                movieStuff->movieOpened = FALSE;

                if ( movieStuff->theMC )
                    DisposeMovieController (movieStuff->theMC);

                if ( movieStuff->theMovie )
                    DisposeMovie (movieStuff->theMovie);

                movieStuff->theMovie = NULL;
                movieStuff->theMC    = NULL;
            }  /* end if ( movieStuff->movieOpened ) */

    }  /* end CloseMovie */


///////////////////////////////////////////////////////////////////////////////////////
//
//  SaveMovie
//
///////////////////////////////////////////////////////////////////////////////////////

void
    SaveMovie
        (MovieStuff  *movieStuff)

    {
        OSErr   theErr = noErr;

        if ( strlen(movieStuff->fileName) != 0 )
            {
                long     movieFlattenFlags = flattenAddMovieToDataFork;
                FSSpec   sfFile;
                OSType   creator = OSTypeConst('TVOD');
                long     createMovieFlags = createMovieFileDeleteCurFile;
                char     theFullPath[255];

                // Make a copy of our full pathname.
                    strcpy (theFullPath, movieStuff->fileName);

                // Convert to a Pascal string.
                    c2pstr( (char*)theFullPath );

                // Make an FSSpec.
                    FSMakeFSSpec (0, 0L, theFullPath, &sfFile);

                // Try to delete the original movie file.
                    DeleteMovieFile (&sfFile);
                // Flatten into a single fork.
                    FlattenMovie (movieStuff -> theMovie,
                                  movieFlattenFlags,
                                  &sfFile,
                                  creator,
                                  -1,
                                  createMovieFlags,
                                  nil,
                                  NULL);

                // Check for error.
                    theErr = GetMoviesError ();

            }  /* end if ( strlen(movieStuff->fileName) != 0 ) */

        return theErr;

    }  /* end SaveMovie */


////////////////////////////////////////////////////////////////////////////////////////
//
//  SaveAsMovie
//
////////////////////////////////////////////////////////////////////////////////////////

void
    SaveAsMovie
        (MovieStuff  *movieStuff)

    {
        unsigned char   lpszPathName[256];
        OPENFILENAME    ofn;
        OSErr           theErr = noErr;

        memset (&ofn, 0, sizeof(OPENFILENAME));
        lpszPathName[0] = '\0';

        ofn.lStructSize     = sizeof(OPENFILENAME);
        ofn.hwndOwner       = GetActiveWindow();
        ofn.lpstrFile       = (LPSTR)lpszPathName;
        ofn.nMaxFile        = sizeof(lpszPathName);
        ofn.lpstrFilter     = "QuickTime Movies (*.mov) \0 *.mov\0";
        ofn.lpstrFileTitle  = NULL;
        ofn.nMaxFileTitle   = (unsigned long)NULL;
        ofn.lpstrInitialDir = NULL;
        ofn.Flags           = OFN_OVERWRITEPROMPT;

        if ( GetSaveFileName (&ofn) )
            {
                long     movieFlattenFlags = flattenAddMovieToDataFork;
                FSSpec   sfFile;
                OSType   creator = OSTypeConst('TVOD');
                long     createMovieFlags = createMovieFileDeleteCurFile;

                // Convert pathname to a Pascal string.
                    c2pstr( (char*)lpszPathName );

                // Make an FSSpec.
                    FSMakeFSSpec (0, 0L, lpszPathName, &sfFile);

                // Try to delete the original movie file.
                    DeleteMovieFile (&sfFile);

                // Flatten into a single fork.
                    FlattenMovie (movieStuff -> theMovie,
                                  movieFlattenFlags,
                                  &sfFile,
                                  creator,
                                  -1,
                                  createMovieFlags,
                                  nil,
                                  NULL);

                // Check for error.
                    theErr = GetMoviesError ();

                // Convert pathname back to a C string.
                    p2cstr( (char*)lpszPathName );

                // Set window title.
                    SetWindowTitle (movieStuff->theHwnd, lpszPathName);
            }  /* end if ( GetSaveFileName (&ofn) ) */

        return theErr;

    }  /* end SaveAsMovie */


///////////////////////////////////////////////////////////////////////////////////////
//
//  EditUndo
//
///////////////////////////////////////////////////////////////////////////////////////

ComponentResult
    EditUndo
        (MovieController   mc)

    {
        ComponentResult   theErr = invalidMovie;

        if ( mc )
            theErr = MCUndo (mc);

        return theErr;

    }  /* end EditUndo */


///////////////////////////////////////////////////////////////////////////////////////
//
//  EditCut
//
///////////////////////////////////////////////////////////////////////////////////////

ComponentResult
    EditCut
        (MovieController   mc)
    {
        Movie             scrapMovie;
        ComponentResult   theErr = invalidMovie;

        if ( mc )
            {
                scrapMovie = MCCut (mc);
                if ( scrapMovie )
                    {
                        theErr = PutMovieOnScrap (scrapMovie, 0L);
                        DisposeMovie (scrapMovie);

                    }  /* end if ( scrapMovie ) */

            }  /* end if ( mc ) */

        return theErr;

    }  /* end EditCut */


///////////////////////////////////////////////////////////////////////////////////////
//
//  EditCopy
//
///////////////////////////////////////////////////////////////////////////////////////

ComponentResult
    EditCopy
        (MovieController   mc)

    {
        Movie             scrapMovie;
        ComponentResult   theErr = invalidMovie;

        if ( mc )
            {
                scrapMovie = MCCopy (mc);
                if ( scrapMovie )
                    {
                        theErr = PutMovieOnScrap (scrapMovie, 0L);
                        DisposeMovie (scrapMovie);

                    }  /* end if ( scrapMovie ) */

            }  /* end if ( mc ) */

        return theErr;

    }  /* end EditCopy */


////////////////////////////////////////////////////////////////////////////////////////
//
//  EditPaste
//
////////////////////////////////////////////////////////////////////////////////////////

ComponentResult
    EditPaste
        (MovieController   mc)

    {
        ComponentResult   theErr = invalidMovie;

        if ( mc )
            theErr = MCPaste (mc, nil);

        return theErr;

    }  /* end EditPaste */


////////////////////////////////////////////////////////////////////////////////////////
//
//  EditClear
//
////////////////////////////////////////////////////////////////////////////////////////

ComponentResult
    EditClear
        (MovieController   mc)
    {
        ComponentResult   theErr = invalidMovie;

        if ( mc )
            theErr = MCClear (mc);

        return theErr;

    }  /* end EditClear */


///////////////////////////////////////////////////////////////////////////////////////
//
//  EditSelectAll
//
///////////////////////////////////////////////////////////////////////////////////////

ComponentResult
    EditSelectAll
        (Movie             movie,
         MovieController   mc)

    {
        TimeRecord        tr;
        ComponentResult   theErr = noErr;

        if ( movie && mc )
            {
                tr.value.hi = 0;
                tr.value.lo = 0;
                tr.base     = 0;
                tr.scale    = GetMovieTimeScale(movie);
                MCDoAction (mc, mcActionSetSelectionBegin, &tr);

                tr.value.lo = GetMovieDuration(movie);
                MCDoAction (mc, mcActionSetSelectionDuration, &tr);

            }  /* end if ( movie && mc ) */
        else
```

```
            {
                if ( movie == NULL )
                    theErr = invalidMovie;
                else
                    theErr = -1;

            }  /* end else */

        return theErr;

    }  /* end EditSelectAll */


////////////////////////////////////////////////////////////////////////////////////////
//
//  UpdateMenus
//
////////////////////////////////////////////////////////////////////////////////////////

void
    UpdateMenus
        (MovieStuff   movieStuff)

    {
        HMENU   hMenu = GetMenu(movieStuff.theHwnd);

        if ( !hMenu )
            return;

        if ( movieStuff.movieOpened )
            {
                EnableMenuItem (hMenu, IDM_SAVE,      MF_ENABLED);
                EnableMenuItem (hMenu, IDM_SAVEAS,    MF_ENABLED);
                EnableMenuItem (hMenu, IDM_UNDO,      MF_ENABLED);
                EnableMenuItem (hMenu, IDM_CUT,       MF_ENABLED);
                EnableMenuItem (hMenu, IDM_COPY,      MF_ENABLED);
                EnableMenuItem (hMenu, IDM_PASTE,     MF_ENABLED);
                EnableMenuItem (hMenu, IDM_CLEAR,     MF_ENABLED);
                EnableMenuItem (hMenu, IDM_SELECTALL, MF_ENABLED);

            }  /* end if ( movieStuff.movieOpened ) */
        else
            {
                EnableMenuItem (hMenu, IDM_SAVE,      MF_GRAYED);
                EnableMenuItem (hMenu, IDM_SAVEAS,    MF_GRAYED);
                EnableMenuItem (hMenu, IDM_UNDO,      MF_GRAYED);
                EnableMenuItem (hMenu, IDM_CUT,       MF_GRAYED);
                EnableMenuItem (hMenu, IDM_COPY,      MF_GRAYED);
                EnableMenuItem (hMenu, IDM_PASTE,     MF_GRAYED);
                EnableMenuItem (hMenu, IDM_CLEAR,     MF_GRAYED);
                EnableMenuItem (hMenu, IDM_SELECTALL, MF_GRAYED);

            }  /* end else */

    }  /* end UpdateMenus */
```

Index

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Mac OS computers and Adobe™ FrameMaker software. Line art was created using Adobe™Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Stephen Chernicoff and Tom Maremaa

ACKNOWLEDGMENTS TO
Guillermo Ortiz
Keith Gurganus
Jim Batson
Brian Friedkin
Chris Flick
Greg Chapman
Jeff Mitchell
Tim Monroe
Kat Stevens

SPECIAL THANKS TO
Mitchell Gass
George Towner