```
                dc c' LQuit \N258*Qq',i1'RETURN'
                dc c'.'

Menu3           dc c'>L Appetizers    \N3',i1'RETURN'
                dc c' LApple Salad \N259',i1'RETURN'
                dc c' LApple Jello \N260',i1'RETURN'
                dc c' LApple Slices \N261',i1'RETURN'
                dc c' LApple Juice \N262',i1'RETURN'
                dc c'.'

Menu4           dc c'>L Entrees    \N4',i1'RETURN'
                dc c' LApple Duckling \N263',i1'RETURN'
                dc c' LApple Dumplings \N264',i1'RETURN'
                dc c'.'

Menu5           dc c'>L Beverages    \N5',i1'RETURN'
                dc c' LApple Shake \N265',i1'RETURN'
                dc c' LApple Cola \N266',i1'RETURN'
                dc c' LApple Wine \N267',i1'RETURN'
                dc c'.'

Menu6           dc c'>L Desserts    \N6',i1'RETURN'
                dc c' LApples \N268',i1'RETURN'
                dc c' LApple Pie \N269',i1'RETURN'
                dc c' LApple Turnover \N270',i1'RETURN'
                dc c'.'

                END

***

MenuTable       DATA

*               Menu 1 (apple)
                dc i'ignore'            ; one for the NDAs
                dc i'ignore'

*               Menu 2 (file)
                dc i'doQuit'            ; quit item selected

*               Menu 3 (appetizers)
                dc i'CheckIt'           ; 'salad'
                dc i'CheckIt'           ; 'jello'
                dc i'CheckIt'           ; 'slices'
                dc i'CheckIt'           ; 'juice'

*               Menu 4 (entrees)
                dc i'CheckIt'           ; 'duckling'
```

```
                dc i'CheckIt'           ; 'dumplings'

*               Menu 5 (beverages)
                dc i'CheckIt'           ; 'shake'
                dc i'CheckIt'           ; 'cola'
                dc i'CheckIt'           ; 'wine'

*               Menu 6 (desserts)
                dc i'CheckIt'           ; 'an apple'
                dc i'CheckIt'           ; 'pie'
                dc i'CheckIt'           ; 'turnover'

                END

***

TaskTable       DATA

                dc i'ignore'            ; 0 null
                dc i'ignore'            ; 1 mouse down
                dc i'ignore'            ; 2 mouse up
                dc i'ignore'            ; 3 key down
                dc i'ignore'            ; 4 undefined
                dc i'ignore'            ; 5 auto-key down
                dc i'ignore'            ; 6 update event
                dc i'ignore'            ; 7 undefined
                dc i'ignore'            ; 8 activate
                dc i'ignore'            ; 9 switch
                dc i'ignore'            ; 10 desk acc
                dc i'ignore'            ; 11 device driver
                dc i'ignore'            ; 12 application
                dc i'ignore'            ; 13 application
                dc i'ignore'            ; 14 application
                dc i'ignore'            ; 15 application
                dc i'ignore'            ; 0 in desk

*
* TaskMaster events
*

                dc i'DoMenu'            ; 1 in menu bar
                dc i'ignore'            ; 2 in system window
                dc i'ignore'            ; 3 in content of window (Move It)
                dc i'ignore'            ; 4 in drag
                dc i'ignore'            ; 5 in grow
                dc i'ignore'            ; 6 in go-away
                dc i'ignore'            ; 7 in zoom
                dc i'ignore'            ; 8 in info bar
```

```
                dc i'ignore'            ; 9 in ver scroll
                dc i'ignore'            ; 10 in hor scroll
                dc i'ignore'            ; 11 in frame
                dc i'ignore'            ; in drop

                END

***

ToolTable       DATA

                dc i'5'                 ; number of tools in table
                dc i'$04,$0100'         ; QuickDraw
                dc i'$06,$0100'         ; Event Manager
                dc i'$0E,$0000'         ; Window Manager
                dc i'$0F,$0100'         ; Menu Manager
                dc i'$10,$0100'         ; Control Manager

                END

***

EventData       DATA

EventRecord     anop                    ; table for Event Manager
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2
TaskData        ds 4
TaskMask        dc i4'$0FFF'

                END

***

QuitData        DATA

QuitFlag        ds 2

QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

***
```

```
GlobalData      DATA

MyID            dc  i'0'                ; program ID word
MyDP            ds  2

                END
```

Listing 9–10
MENU.C program

```
/************************************/
/* Data and routine to create menus */
/************************************/

/*    Set up menu strings. Because C uses \ as an escape character,
we use two when we want a \ as an ordinary character. The \
at the end of each line tells C to ignore the carriage return. This lets
us set up our items in an easy-to-read vertical alignment. */

char *menu1 = "\
>L@\\XN1\r\
 LAn Apple Menu\\N257\r\
.";

char *menu2 = "\
>L File   \\N2\r\
 LQuit \\N258*Qq\r\
.";

char *menu3 = "\
>L Appetizers   \\N3\r\
 LApple Salad \\N259\r\
 LApple Jello \\N260\r\
 LApple Slices \\N261\r\
 LApple Juice \\N262\r\
.";

char *menu4 = "\
>L Entrees   \\N4\r\
 LApple Duckling \\N263\r\
 LApple Dumplings \\N264\r\
.";

char *menu5 = "\
>L Beverages   \\N5\r\
 LApple Shake \\N265\r\
 LApple Cola \\N266\r\
 LApple Wine \\N267\r\
.";
```

```
char *menu6 = "\
>L Desserts    \\N6\r\
 LApples \\N268\r\
 LApple Pie \\N269\r\
 LApple Turnover \\N270\r\
.";

#define QUIT_ITEM 258  /* these will help us check menu item numbers */
#define LAST_ITEM 270

BuildMenu()
{
    InsertMenu(NewMenu(menu6),0);
    InsertMenu(NewMenu(menu5),0);
    InsertMenu(NewMenu(menu4),0);
    InsertMenu(NewMenu(menu3),0);
    InsertMenu(NewMenu(menu2),0);
    InsertMenu(NewMenu(menu1),0);
    FixMenuBar();
    DrawMenuBar();
}

/*******************************/
/* Main routine and event loop  */
/*******************************/

WmTaskRec   myEvent;
Boolean done = false;

main()
{
    StartTools();
    BuildMenu();
    EventLoop();
    ShutDown();
}

/*   When a menu bar event is returned, test the item number for a
checkable item. Use the logical inverse of the value returned by
GetMItemMark as a parameter to CheckMItem. This will toggle the check
mark for each item.  */

EventLoop()
{
Word *data = (Word *)&myEvent.wmTaskData; /* address of item id  */

    myEvent.wmTaskMask = 0x0FFF;
    while (!done)
```

```
        if ( TaskMaster(everyEvent,&myEvent) == wInMenuBar ) {
           if (*data == QUIT_ITEM)
              done = true;
           else if ((*data > QUIT_ITEM) && (*data <= LAST_ITEM))
              CheckMItem (!GetMItemMark(*data), *data);
           HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu
            id  */
        }
}
```

CHAPTER 10

Doing Windows

Using the Window Manager

Yes, the Apple IIGS does windows—and with real class, too! To make sure they're done properly, the IIGS employs a Window Manager. The Window Manager—like the Event Manager, which was introduced in chapter 7—is a very important toolkit in the Apple IIGS Toolbox. It is the Window Manager's job to handle all windows placed on the IIGS desktop. It can create, draw, shrink, expand, scroll, and move windows. When you've finished working with a window, the Window Manager can remove it from your screen. When you're through with a window for good, the Window Manager can dispose it and deallocate its memory.

The Window Manager takes care of all kinds of windows, not just picture windows and document windows, but also dialog windows, alert windows, and windows custom-tailored for specific programs. Want a round window or a triangular window? The Window Manager can make one. How about a window that seems to explode when you click the mouse in its go-away box or a window with custom-designed controls? No problem for the Apple IIGS Window Manager. It's a toolkit that can do just about any kind of window.

Kinds of Windows

The kinds of windows the Window Manager can manage are divided into three categories:

- *Document windows.* Most of the windows used in IIGS programs are in this category. A window doesn't have to contain text to be classified as a document window. Windows that contain pictures drawn with programs like PaintWorks Plus are also document windows.
- *Dialog windows.* There are three kinds of dialog windows: modal dialogs, modeless dialogs, and alert windows. Although low-level operations for all three types of windows can be handled by the Window Manager, they are mostly the responsibility of the Dialog Manager. So we won't go into detail about them until chapter 11, which is all about the Dialog Manager.
- *Custom-designed windows.* You can design custom windows using the Window Manager, but that is beyond the scope of this book. If you'd like to design your own windows, you can find some tips on how to do it in the *Apple IIGS Toolbox Reference.*

Window Frames

There are two kinds of predefined window frames: *alert window frames* and *document window frames.* An alert window frame is a double black line. A document frame is a single black line or includes controls.

A window does not have to be an alert window to have an alert window frame; document windows can have alert window frames, too. A standard document window frame and an alert window frame are illustrated in figure 10–1.

Controls

The screen of the IIGS represents a working desktop. Various graphic objects appear on this desktop and are manipulated with a mouse. A window is a

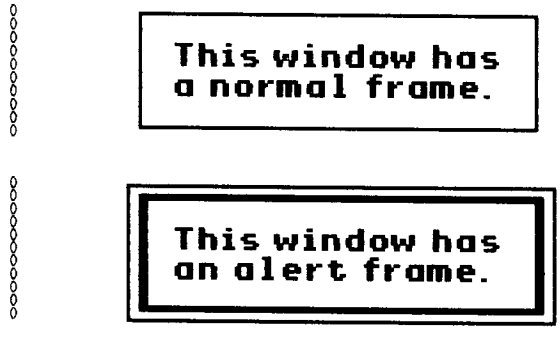


Figure 10–1
Document frame and alert frame

desktop object that presents information; it can contain a document, a picture, a message, or other items. Windows can be almost any size or shape, and one or more of them can be on the desktop at any time.

Windows owe their name to the fact that they can show you more information than the IIGS screen can display at one time. When a window is on the screen, you can look through it into a larger area. The information displayed through a window can be pictures, text, data, or all three. When you look at something through a window—for example, a picture—the window can be moved around over the picture with a control called a *scroll bar.*

Most document windows have two scroll bars: a horizontal scroll bar, which scrolls the window horizontally, and a vertical scroll bar, which scrolls the window vertically. You'll learn how to use both kinds of scroll bars before you finish this chapter.

A document window can also have the following controls:

- A *title bar,* which is a horizontal bar that displays the window's title, if there is one. A title bar can contain a close box, which makes the window disappear from the screen, and a zoom box, which changes the window's size. A title bar can be used as a *drag region* for moving the window.
- A *grow region,* which is a small box in the lower right corner of a window that changes the window's size.
- An *information bar,* another horizontal bar in which an application can display information that won't be affected by the movements of scroll bars.

Information bars may have their uses, but they are not popular in programs written for the IIGS. A standard document window, without an information bar, is illustrated in figure 10–2. The controls in the title bar of a document window are used as follows:

- Clicking the mouse anywhere in an inactive window highlights its title bar and makes it the active window, the window in which drawing and other activities take place. The title bars of all other windows become unhighlighted. Although these windows remain on the screen, they become inactive windows. According to Apple's *Human Interface Guidelines,* there should never be more than one active window on the screen.
- Clicking the mouse in the close box, or go-away region, closes the window. Usually, when you click the mouse in the close box, an application program calls the Window Manager routine `HideWindow`, which makes the window disappear from the screen.
- Pressing the mouse button in the window's drag region (title bar) and then dragging the window pulls an outline of the window across the screen. Holding the mouse button down and releasing it in a new location moves the window there. Unless the Apple key is held

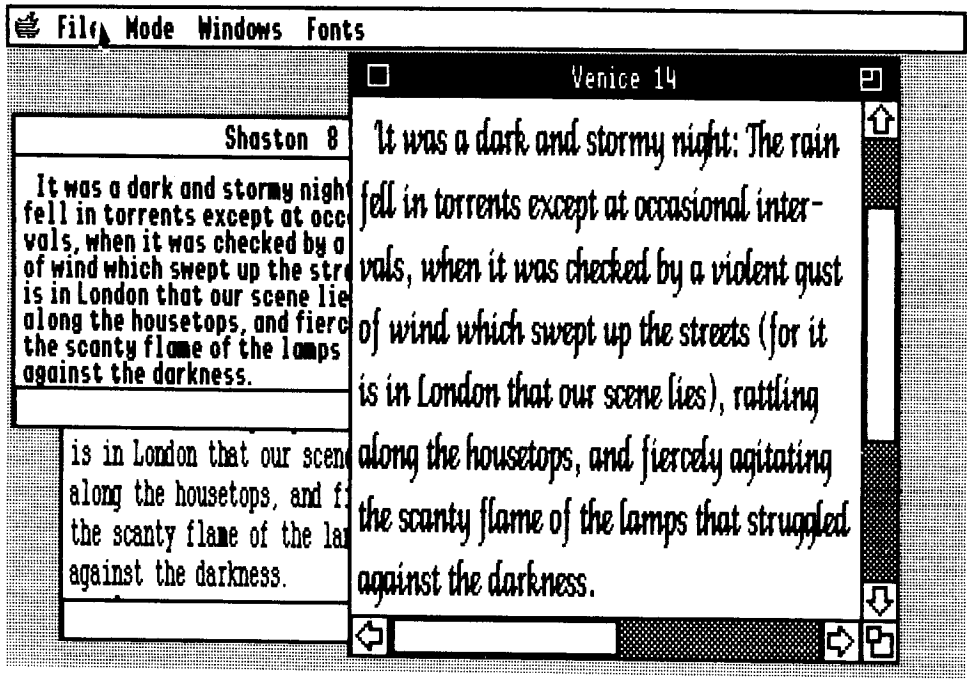


Figure 10–2
Standard document window

down when the mouse button is released, the moved window becomes the active window.

- Clicking the mouse inside the grow box and then dragging the grow box changes the window's size.

To keep windows from getting lost, the Window Manager prevents them from being dragged completely across the screen. The title bar can never be moved to a point where the visible area of the title bar is less than four pixels square.

Some windows are created by application programs and others are created by tools in the Toolbox. (For example, the Dialog Manager can create dialog windows.) Windows created by application programs and by tools in the Toolbox are known collectively as *application windows*. Another class of windows, called *system windows*, display desk accessories.

What the Window Manager Does

The Window Manager draws windows using QuickDraw II and the Control Manager, and it disposes them with the help of the Memory Manager. After a window is drawn on the screen, the Window Manager's main function is

to keep track of overlapping windows. The Window Manager handles tasks so that you can draw in any window without running into windows in front of it. You can move a window to a different place on the screen, change its size, or change its plane (front-to-back order), and you don't have to worry about details, such as how parts of various windows cover parts of other windows. The Window Manager redraws windows as needed and ensures that they overlap properly.

Window Regions

Every window is made up of two regions:

- A *content region,* which is the area that lies inside the window's frame. An application can draw objects and text in this portion of a window.
- A *frame region,* which is the outline of the entire window, including its title bar and standard window controls.

A window's content region and frame region make up what is known as the *structure region* of the window.

Every window also has a *data area*: a block of memory that includes all the data that can be viewed through the window. If the window has scroll bars, they can be used to move the window over its data area.

If a window has a grow box, a zoom box, or both, they can be used to increase or decrease the size of the window, causing more or less of its data area to be displayed. When the window is scrolled, it moves over the data area. But when the window is moved from one part of the screen to another, the data area is moved with it, so the view remains the same.

Initializing the Window Manager

Before the Window Manager can be started up, it must be loaded into memory, and QuickDraw and the Event Manager must be loaded and initialized. The Window Manager call `WindStartup` can then be issued to initialize the Window Manager. Then you can use the Window Manager call `NewWindow` to create any windows needed in a program.

TaskMaster

In programs that use the Window Manager, there are two ways to handle user input. One way is to use the Event Manager call `GetNextEvent`. The other is to use the Window Manager call `TaskMaster`.

The easiest way to use the Window Manager is with TaskMaster. As you may recall from chapter 9, TaskMaster can handle events related to menus

as well as events that involve windows. The interaction between TaskMaster and menus is covered in chapter 9. In this chapter, you see how to use TaskMaster in programs that make use of windows.

WINDOW.S1 shows how an assembly language program can handle windows using TaskMaster. WINDOW.C is a C language version of the same program. Both programs are at the end of this chapter.

When TaskMaster is used in a program, it does the following. First, TaskMaster makes the Event Manager call `GetNextEvent`. If an event isn't ready, TaskMaster returns a task code of 0. If an event is ready, TaskMaster looks at it and tries to handle it. If TaskMaster can't handle the event, it returns the event code to the application. The application can then handle the event as if its event code had been returned by `GetNextEvent`.

If TaskMaster can handle the event, it calls standard functions to try to complete the task. For example, if you press the mouse button in an active window's zoom box, TaskMaster makes the Window Manager call `TrackZoom` until the mouse leaves the zoom box or the mouse button is released. If you release the mouse button while the mouse is in the zoom box, TaskMaster calls `ZoomWindow` to zoom the window either in or out, as appropriate. This takes care of the complete zoom operation selected by the user, so TaskMaster returns no event.

If TaskMaster can handle only part of an event, it does what it can and then returns control to the calling program. For example, if you press the mouse in the active window's content region, TaskMaster can detect it, but it can't do anything further. In this case, TaskMaster returns a task code of $0013 (`wInContent`). That lets an application program know that the mouse button has been pressed in the active window's content region, but it is up to the application to determine what to do next.

The operation of TaskMaster is covered in detail in chapter 9, but here's a brief review. A call to TaskMaster takes three paramaters: a word to save a space on the stack, an event mask, and a pointer to a task record.

The event mask passed to TaskMaster is like an event mask used by the Event Manager. The task record used by TaskMaster is like an event record used by the Event Manager, but with two extra fields. Each time TaskMaster makes a `GetNextEvent` call, `GetNextEvent` fills in the first seventeen fields of the task record being used by TaskMaster. Then TaskMaster handles any events it can handle, fills in the last two fields of the task record, and returns.

Listing 10–1 is a task record used in this chapter's example program, WINDOW.S1. The WINDOW.S1 program, listed in its entirety at the end of this chapter, is a sketcher program that allows the user to draw into a window with a mouse. When a sketch is drawn, each dot in it is actually drawn twice: once into the window on the screen and once into a pixel image that paints the window's contents each time the window is updated. Thus, sketches drawn using the WINDOW.S1 program do not disappear from memory when a window is removed from the screen. They remain in memory and can show up in a window again when it is redrawn on the screen. In later chapters, the WINDOW.S1 program becomes even more sophisticated.

Listing 10–1
Task record in WINDOW.S1

```
EventData       DATA

EventRecord     anop
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2
TaskData        ds 4
TaskMask        dc i4'$0FFF'

                END
```

As you may recall from chapter 9, the event mask passed to TaskMaster as a parameter is different from the `TaskMask` passed to TaskMaster as part of a task record. The event mask passed to TaskMaster is the same kind of mask that is passed to the Event Manager in the `GetNextEvent` call.

A task mask is a word used by an application to tell TaskMaster what kinds of events it should look for and what kinds of events it should ignore. The high word of a task mask—bits 16 through 31–should always be clear. In the low word of a task mask, each bit corresponds to a task; a set bit causes TaskMaster to look for an event, and a cleared bit tells TaskMaster to ignore an event. For TaskMaster to look for every type of event it can handle, the task mask should be $0000FFFF. The bit layouts of an event mask and a task mask are listed in chapter 9.

Window Records

For each window used in an application program, the Window Manager maintains a *window record.* A window record contains a number of fields, but only the first seven are directly accessible to application programs. The rest of the fields in a window record can be accessed only through calls to the Window Manager. Table 10–1 shows the seven window record fields accessible to application programs.

When the Window Manager is active, it maintains a window list: a list of all windows currently open. It is important to note that a window can be open but hidden, and thus not visible on the screen.

As table 10–1 shows, the first field in a window record is a pointer to the Window Manager's window list. The second field is the window's GrafPort—the GrafPort itself, not a pointer to it. Thus, the length of the GrafPort field is the length of a GrafPort; the field is 186 bytes.

When a window is created using the Window Manager call `NewWindow`, the call returns a pointer to the new window's GrafPort. Thus, the value returned by `NewWindow` is also a pointer to the second field of a window

Table 10–1
Window Record Fields Accessible to Application Programs

Name	Length	Function
wNext	Long	Pointer to next window in the window list
wport	186 bytes	Window's port; returned window pointers point to here
wStrucRgn	Handle	Handle of window's structural region (frame plus content)
wContRgn	Handle	Handle of window's content region
wUpdateRgn	Handle	Handle of update regions (regions that needs redrawing)
wControl	Handle	Handle of application's first control in content region
wFrame	Word	Bit array that describes window's frame

record. So the value returned by NewWindow, as well as being a pointer to a GrafPort, can also calculate the addresses of the other six fields of a window record.

In addition to a GrafPort and a pointer to the next window in the window list, a window record contains a pointer to the window's title. A window's title is a bit array that provides details about the window's frame and the handles of four regions used to draw the window. The bit array in the wFrame field of a window record is shown in table 10–2.

NewWindow Call

Every window used in an application program must be set up with a call to the Window Manager routine NewWindow. A call to NewWindow takes two parameters: 2 null words (zeros) to save spaces on the stack and a pointer to a parameter block. The call returns with a pointer to a window pushed onto the stack. Listing 10–2 is a NewWindow call used in the WINDOW.S1 program.

Listing 10–2
Call to NewWindow

```
PushLong #0                    ; space for result
PushLong #WinOParamBlock
_NewWindow


pla
sta WinOPtr
pla
sta WinOPtr+2
```

Table 10–2
Bits in the wFrame Field

Bit	Name of Field	Value
0	F_HILITED	1 = Frame highlighted
		0 = Frame not highlighted
1	F_ZOOMED	1 = Currently zoomed
		0 = Frame not zoomed
2	F_ALLOCATED	1 = Record was allocated
		0 = Record was provided by application
3	F_CTRL_TIE	1 = Control's state is independent
		0 = Inactive window has inactive controls
4	F_INFO	1 = Information bar
		0 = No information bar
5	F_VIS	1 = Window is currently visible
		0 = Window is invisible
6	F_QCONTENT	1 = Return `wInContent` even if window is inactive
		0 = Don't return `wInContent` if window is inactive
7	F_MOVE	1 = Title bar is a drag region
		0 = No drag region
8	F_ZOOM	1 = Zoom box on title bar
		0 = No zoom box (zoom box must have title bar)
9	F_FLEX	1 = `GrowWindow` and `ZoomWindow` won't change the origin
		0 = `GrowWindow` and `ZoomWindow` will affect the origin
10	F_GROW	1 = Grow box
		0 = No grow box (grow box must have at least one scroll bar)
11	F_BSCRL	1 = Window frame horizontal scroll bar
		0 = No horizontal scroll bar
12	F_RSCRL	1 = Window frame vertical scroll bar
		0 = No vertical scroll bar
13	F_ALERT	1 = Alert type frame (don't set grow box, close box, info bar, title bar, or scrolls)
		0 = Standard frame
14	F_CLOSE	1 = Close box
		0 = No close box (close box must have title bar)
15	F_TITLE	1 = Title bar
		0 = No title bar

Parameter Blocks

Before an application makes a NewWindow call, it must set up a parameter block that spells out many details about the window. Listing 10–3 is a NewWindow parameter block used in the WINDOW.S1 program. The fields in a window's parameter block are described in table 10–3.

Listing 10–3
Parameter block for a NewWindow call

```
WinOParamBlock anop
          dc     i'WinOEnd-WinOParamBlock'
          dc     i2'%1101110111000000' ; Bits describing frame
          dc     i4'WinOTitle'         ; Pointer to title
          dc     i4'0'                 ; RefCon
          dc     i2'26,0,188,308'      ; Full size (0=default)
          dc     i4'0'                 ; Color table pointer
          dc     i2'0'                 ; Vertical origin
          dc     i2'0'                 ; Horizontal origin
          dc     i2'200'               ; Data area height
          dc     i2'320'               ; Data area width
          dc     i2'200'               ; Max cont height
          dc     i2'320'               ; Max cont width
          dc     i2'2'                 ; No. of pixels to scroll vertically
          dc     i2'2'                 ; No. of pixels to scroll horizontally
          dc     i2'20'                ; No. of pixels to page vertically
          dc     i2'32'                ; No. of pixels to page horizontally
          dc     i4'0'                 ; Information bar text string
          dc     i2'0'                 ; Info bar height
          dc     i4'0'                 ; DefProc field
          dc     i4'0'                 ; Routine to draw info bar
          dc     i4'Paint0'            ; Routine to draw content
          dc     i'26,0,188,308'       ; Size/position of content
          dc     i4'$FFFFFFFF'         ; Plane to put window in
          dc     i4'0'                 ; Address for record (0=to allocate)

WinOEnd   anop
```

Windows and GrafPorts

Before the NewWindow call returns, it creates a GrafPort for the window being set up and pushes a pointer to that GrafPort onto the stack. From that point, the application that created the window can treat it as a GrafPort. The application can draw into the window using QuickDraw II routines.

When the NewWindow call sets up a window, it uses the information passed in the window's parameter block to create the window's attributes. For example, the first field in the parameter block describes the window's frame—using the bit layout illustrated in table 10–2—and the second field

Table 10–3
Fields in a Window Parameter Block

Field	Name	Length	Description
1	paramlength	Word	Number of bytes in parameter table
2	wFrame	Word	Bit array describing window frame
3	wTitle	Pointer	Pointer to window's title
4	wRefCon	Long	Reserved for application's use
5	wZoom	Rect	Size and position of window when zoomed (0 = screen size)
6	wColor	Pointer	Pointer to window's color table
7	wYOrigin	Word	Content's vertical origin
8	wXOrigin	Word	Content's horizontal origin
9	wDataH	Word	Height of entire document or pixel image
10	wDataW	Word	Width of entire document or pixel image
11	wMaxH	Word	Maximum height of content allowed by GrowWindow
12	WMaxW	Word	Maximum width of content allowed by GrowWindow
13	wScrollVer	Word	Number of pixels to scroll document vertically using scroll bar arrows
13	wScrollHor	Word	Number of pixels to scroll document horizontally using scroll bar arrows
14	wPageVer	Word	Number of pixels to scroll vertically using page control
14	wPageHor	Word	Number of pixels to scroll horizontally using page control
15	wInfoRefCon	Long	Value passed to information bar draw routine
16	wInfoHeight	Word	Height of information bar
17	wFrameDefProc	Pointer	Address of standard window definition procedure
18	wInfoDefProc	Pointer	Address of routine that draws information bar interior
19	wContDefProc	Pointer	Address of routine that draws content region interior
20	wPosition	Rect	Window's starting position and size
21	wPlane	Long	Window's starting plane (FFFFFFFF = frontmost)
22	wStorage	Pointer	Address of memory to use for window record (0 = don't care)
23	paramlength	Word	Total number of bytes in parameter table, including this field

contains the window's title. In subsequent fields, the width and height of the window's data areas and content areas are defined. A data area is a rectangle that encloses all the data a window can work with (for example, a pixel map). A content area is a rectangle enclosing the largest portion of the data area that may be displayed on the screen.

Some fields in a window's parameter block duplicate fields in the window's window record. When a window is created using a `NewWindow` call, the call uses information provided in the window's parameter block to fill in the corresponding fields of the window's window record.

One very important field in a window parameter block is the fourth field from the end of the block. This field contains a pointer to a routine that is used to draw the contents of the window each time the window is displayed on the screen. In the WINDOW.S1 program, the field looks like this:

```
dc      i4'Paint0'              ; Routine to draw content
```

The routine that paints a window must be written according to a specific format, and must end with the assembly language mnemonic `rtl`.

In the `Paint0` segment of the WINDOW.S1 program, the QuickDraw call `PPToPort` copies the contents of a specific pixel map into the window used in the program. This pixel map is set up in a program segment called `MakeWin0` and is accessed in the program by the pointer `PicOPtr`.

The program segments `MakeWin0` and `Paint0` are in listing 10–5, the complete listing of the WINDOW.S1 program at the end of this chapter. Here is what happens in the segment of code labeled `Paint0`.

First, the Memory Manager call `NewHandle` reserves a 32K block of RAM—enough memory to hold a pixel map that is the size of one screen. The call returns with a handle to the requested block of data pushed onto the stack. This handle is then pulled off the stack and stored in a variable called `Win0Handle`. Later in the program, the `Paint0` routine uses the block of data pointed to by `Win0Handle` to draw the contents of the program's window on the screen.

When the handle called `Win0Handle` is assigned, a segment of code labeled `Deref` dereferences the handle (converts it into a pointer). The `Deref` routine also locks the handle being dereferenced so the Memory Manager can't move the handle's block of memory in the middle of an important operation, which could crash the program. Later, when the important operation is over, the `Unlock` routine unlocks the handle, enabling the Memory Manager to manage it again.

When `Win0Handle` is dereferenced, the pointer thus obtained is stored in a `LocInfo` data structure at the end of the WINDOW.S1 program in a field labeled `PicOPtr`. Then a `NewWindow` call creates a new window. To set the new window's attributes, the `NewWindow` call uses the parameter block in listing 10–3.

As explained previously, the WINDOW.S1 program allows you to draw into a screen window and, at the same time, to draw into the pixel map that paints the window on the screen each time it is updated or redrawn. This is why sketches drawn with the WINDOW.S1 program do not vanish from

memory when a window is removed from the screen. Instead, they remain in RAM and can be redrawn into a window when it shows up again on the screen.

To make this technique possible, the WINDOW.S1 program creates a GrafPort that can be used to draw into the pixel map from which the program's window is drawn. This GrafPort is set up in the `NewPort` program segment. For its `LocInfo` data, the new GrafPort uses the `PicOLocInfo` data structure in the `PortData` data segment at the end of the program.

When the GrafPort that points to a pixel image is created, the WINDOW.S1 program clears the area of memory used for the pixel image with the `BlkFill` program segment. In this segment, the pen color is set to white and the QuickDraw call `PaintRect` clears the bit image to white. Later in the program, when the user asks for a new blank screen by making the menu choice `New`, the program uses the `BlkFill` routine to clear both the window port and the bit image port to white.

(Incidentally, the `PaintRect` call can be used to fill any block of RAM with any value, even in a nongraphics program. To "stuff" a block of memory, just pass to `PaintRect` the size of the area you want filled and the value you want it filled with. `PaintRect` does the rest—and you save the time and effort it would take to write a 65C816 block fill program.)

Window Manager's GrafPort

The WINDOW.S1 program, like every program that uses windows, has another GrafPort that is created by the Window Manager. When you use the Window Manager in a program, it always creates a special GrafPort that has the entire screen as its port rectangle. In all programs that use the Window Manager, this port is known as the Window Manager port. The Window Manager uses it to draw all windows, along with their scroll bars and other controls, on the IIGS screen.

How a Window Is Drawn

When the Window Manager draws or redraws a window, it always draws the window's frame first. Then it draws the window's contents.

During this process, the Window Manager manipulates regions of the Window Manager port as necessary to ensure that only what should be drawn is drawn. The Window Manager generates an *update event* to draw a window's contents. But before an update event can take place, the Window Manager must accumulate, in the update region, the areas of the window's content region that need updating.

In programs that use either TaskMaster or the Event Manager, the Event Manager periodically calls a routine called `CheckUpdate` to see if there is a window on the screen whose update region is not empty. If it finds one, it reports that an update event has occurred and passes a pointer to the window that needs updating in the event message field of its event record. If TaskMaster is used, it then updates the window as required. Programs that don't use TaskMaster have to do the updating themselves. Obviously, it's easier to use TaskMaster.

Some Window Manager routines can change the state of a window from inactive to active or from active to inactive. For each change, the Window Manager generates an *activate event,* passing along the window pointer in

the event message. The activeFlag bit in the modifiers field of the event record is set if the window becomes active and cleared if it becomes inactive.

When the Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event to the application or TaskMaster through its GetNextEvent routine. An activate event has the highest priority of any type of event, so when the Event Manager detects one it gets immediate action.

Usually, activate events are generated in pairs, because when one window becomes active another usually becomes inactive, and vice versa. Occasionally, however, a single activate event is generated, for example, when there is only one window in the window list or when an active window is closed permanently.

When a pair of activate events comes along, the Window Manager first generates the event for the window becoming inactive. It then generates the event for the window becoming active. In most applications, pairs of activate events are handled competently by TaskMaster. Rarely does an application program have to intervene.

Coordinates and the Window Manager

When NewWindow is called to create a window, it takes the window's bounds rectangle from the LocInfo field of the window's GrafPort. Thus, a window's local coordinates begin in the upper left corner of the bounds rectangle specified in the LocInfo field of the window's GrafPort. In a window's global coordinate system, coordinate 0,0 is always assigned to the pixel in the upper left corner of the window's bounds rectangle.

Global Coordinates in WINDOW.S1

In the WINDOW.S1 program, the LocInfo record that defines the window's bounds rectangle is titled PicOLocInfo. This record is in a data segment labeled PortData, which appears at the end of the program. The bounds rectangle defined in the PicOLocInfo record appears in the PicOFrame field. In the WINDOW.S1 program, therefore, the bounds rectangle assigned to the program's window is the rectangle 0,0,200,320.

The global coordinates of a window are always based on a pixel image, specifically, the pixel image pointed to by the second field of the window's LocInfo record. In a window's global coordinate system, coordinate 0,0 is always assigned the pixel in the upper left corner of the window's pixel image.

Local Coordinates in WINDOW.S1

The pointer to the pixel image used in the WINDOW.S1 program is PicOPtr. This pointer is the second field in a LocInfo record called PicOLocInfo. The PicOLocInfo record is in a data segment called PortData, which appears at the end of the program.

Port Rectangle in WINDOW.S1

The port rectangle of a window is a rectangle outlining the maximum portion of the window that can be displayed on the screen at any given time. If a window is partially hidden (for example, partly covered by another window or partly off the screen), the window's visible region (VisRgn) is also used

to determine how much of the window is visible on the screen. In the WINDOW.S1 program, the Window Manager takes care of VisRgns automatically. But, as you shall see shortly, the program has to perform a few manipulations using port rectangles.

Coordinate Conversions in WINDOW.S1

In programs like WINDOW.S1, coordinates often have to be converted from one system to another. Some QuickDraw and Window Manager routines use global coordinates, but others use local coordinates. For example, in the segment of the WINDOW.S1 program labeled MoveIt, TaskMaster returns mouse coordinates in global coordinates, and the Event Manager call GetMouse and the QuickDraw II call LineTo require local coordinates. For this reason, the QuickDraw call GlobalToLocal is used to convert the global coordinates returned by TaskMaster to the local coordinates required by other calls.

The MoveIt segment of the WINDOW.S1 program is the heart of the program. In this section, mouse movements are tracked and lines are drawn on the screen. TaskMaster detects the location of the IIGS mouse and returns it, in global coordinates, in the EventWhere field of its task record. The mouse location is then converted into local coordinates in these two lines:

```
PushLong #EventWhere
_GlobalToLocal
```

The GlobalToLocal call converts the global coordinates in the EventWhere record to local coordinates. After this conversion, the EventWhere field contains local coordinates, which can then be used by calls that require them. In the MoveIt segment, other conversions are taken care of by the StartDrawing and SetOrigin calls.

When a window is created, the upper left coordinate of its bounds rectangle are usually set to 0,0. Thus, in the local coordinate system used by a new window, the first pixel in its bounds rectangle is generally assigned the coordinate 0,0.

As you have seen, every window has both a port rectangle and a bounds rectangle. The intersection of a window's bounds rectangle and port rectangle make up the largest possible area of the window that can be displayed on the screen.

Suppose a window has a bounds rectangle that starts at local coordinate 0,0 and is the same size as the screen. Let's also suppose the window has a port rectangle that covers a smaller area in the middle of the screen. The coordinates of this port rectangle are 65,50 (the vertical coordinate is listed first). A bounds rectangle and a port rectangle that fit this description are illustrated in figure 10–3.

Now let's assume you want to use the WINDOW.S1 program to draw a sketch in the window (that is, in the port rectangle) shown in figure 10–3. You first have to convert the mouse location returned by TaskMaster from global coordinates to local coordinates. But, because of the way the IIGS Window Manager works, you also have to reset the origin of the window's

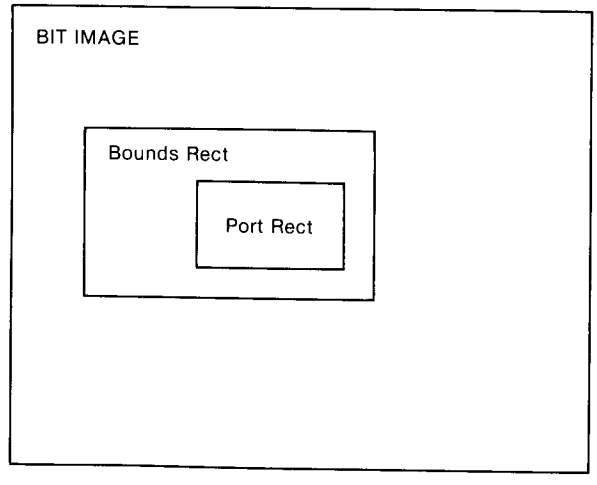


Figure 10–3
Relationship between a bit image, BoundsRect, and PortRect

port rectangle; you have to change the value of the upper left corner of the port rectangle, as expressed in local coordinates.

This is why the port rectangle's origin must be reset. When the Window Manager draws all the windows on a screen—complete with scroll bars, title bars, and all other necessary features—it uses a GrafPort that has the whole screen as its bounds rectangle. But before the Window Manager can draw the content region of a single window (for example, when the window has to be updated or redrawn), it has to switch to that window's GrafPort and change the origin of the window's port rectangle from its usual value of 0,0 to the value it had when it was a port rectangle in the Window Manager's GrafPort, which uses the whole screen as its bounds rectangle.

The logic of this procedure is a little difficult to follow. After the origin of a window's port rectangle is changed, the Window Manager can draw into the window, and the drawing ends up in the proper location on the screen.

When the Window Manager has finished drawing in a window, it must set the window's origin back to 0,0 before it can leave the window's port and return to its own GrafPort, so that it can regain the capability of drawing anywhere on the screen.

When the Window Manager has to draw in a window, it automatically carries out all the procedures just outlined. But when an application wants to draw in a window, it has to perform the same kinds of operations the Window Manager performs when it draws in a window.

To start drawing in a window, an application can use one of two approaches. It can either

- Make the QuickDraw call `SetPort` to make the window's port the current port and then make the QuickDraw call `Set Origin` with the proper parameters
- Make the Window Manager call `StartDrawing`, which carries out both of the previous steps automatically

The simpler approach is to use the StartDrawing call—and that is what is done in the WINDOW.S1 program.

After an application has finished drawing into a window, it must return the origin of the window's port rectangle to its original state by making the QuickDraw call SetOrigin using parameters 0,0.

Running the WINDOWS.S1 Program

After the procedure for drawing into a window is understood, the operation of the WINDOW.S1 program becomes straightforward. The main part of the WINDOWS.S1 program is MainProgram. In this section, the tools used by the program are initialized, a menu is constructed, and the MakeWin0 subroutine is called to create a window.

Next, the NewPort subroutine is called to set up a GrafPort used by the window's pixel map. Then the BlkFill subroutine is called to clear the pixel map to white. (You could clear the screen to another color by simply replacing the color code $FF in the BlkFill routine with a different color code.)

When the window's pixel image is cleared, the WINDOW.S1 program jumps to the EventLoop subroutine. This is the main event loop of the program. While the event loop is running, TaskMaster continuously looks for button down events. If TaskMaster detects a button down event, the program uses a jump table labeled TaskTable to determine what should be done.

If TaskMaster reports a menu event, the table called TaskTable sends the program to the doMenu subroutine. It is up to doMenu to carry out an appropriate response to the user's menu selection. Depending upon the menu choice, the doMenu routine can either call the Repaint subroutine to draw a new window, call the doWin0 subroutine to redraw a window, or jump to the doQuit subroutine to end the program.

If a window event is detected, TaskMaster takes care of all routine window-related operations, such as scrolling the window or changing its size. If TaskMaster detects a button down event in the window's go-away box, the program jumps to a short subroutine titled doGoAway, which hides the window. If TaskMaster reports a button down event in the window's content region, the program jumps to the MoveIt subroutine, which enables the user to draw in the window.

The MoveIt routine, as noted, is the heart of the WINDOW.S1 program. In this segment of code, as long as the mouse is inside a window and the mouse button is down, the QuickDraw call LineTo draws a line on the screen tracing the mouse's movements. When the mouse button is released, the mouse's movements are still followed, but the tracing is done using the MoveTo call rather than the LineTo call, so no line is drawn on the screen.

You can clear the window at any time by making the menu selection New. You can temporarily hide the window being drawn by clicking the mouse in the window's go-away region. If a window is hidden, but is not erased with a click in the menu item New, you can bring the window back

into view by making the menu selection Untitled (for now, the title of the window). After New is selected, however, the window is permanently erased and cannot be retrieved from memory.

Other Features of WINDOW.S1

The WINDOW.S1 program has some new features that should be mentioned before you conclude this chapter. One is the `InsertSysDisk` subroutine, which is called from the `ToolInit` program segment. The other new and noteworthy feature is a macro called `ErrorCheck`, which is also called from the `ToolInit` segment of the program.

The `InsertSysDisk` subroutine is called when the WINDOW.S1 program tries to load the tools it needs and finds that the IIGS system disk—on which some tools are stored—is not currently in the computer's disk drive. When this condition is detected, `InsertSysDisk` is called and prints a message on the screen asking the user to insert the system disk in the disk drive.

The `ErrorCheck` macro is called following several critical routines, such as the loading of essential tools. If the calling of a vital routine is aborted by an error, the `ErrorCheck` macro ends the program. A system failure message—a rolling-Apple symbol accompanied by an error message and an error number—is displayed on the screen.

InsertSysDisk Routine

To see how the `InsertSysDisk` routine works, look through the `ToolInit` segment for the label `LoadEmUp`. Study the code that follows the labels `LoadEmUp` and `DoInsertDisk`, and you'll see that this section of code forms a loop. When the program comes to the `LoadEmUp` label, it makes the Tool Locator call `LoadTools` to load all the tools used in the program. The `LoadTools` call, like most Toolbox calls, uses a specific convention for detecting errors. If the call is completed successfully, without an error, it returns with the P register's carry flag clear and a value of 0 in the accumulator. If an error is encountered in making the call, however, the call returns with the carry bit set and an error number in the accumulator.

In the WINDOW.S1 program, if the `LoadTools` call returns without an error, the program jumps a few lines to a section of code labeled `ToolsLoaded` and the tools that have been loaded start up normally. If the call returns with the carry set and the number 45 in the accumulator, however, the program jumps to the `DoInsertDisk` subroutine, which prints a message on the screen asking the user to insert the IIGS system disk (which contains some of the tools used by the computer). If the user complies and the necessary tools are found, the program proceeds normally. If this doesn't solve the problem, the program ends and a system failure message is displayed.

ErrorCheck Macro

To end programs and display system-death messages after fatal errors occur, the WINDOW.S1 program uses the `ErrorCheck` macro. Several calls to the `ErrorCheck` macro appear in the `ToolInit` segment of the WINDOW.S1 program.

The `ErrorCheck` macro appears in listing 10–4. To use it in your programs, type it into a macro file and add it to your library of macros using APW's `MACGEN` shell command.

Listing 10–4
ErrorCheck

```
 MACRO
&lab ErrorCheck &msg
&lab bcc end&syscnt
 pea x&syscnt|-16
 pea x&syscnt
 ldx #$1503
 jsl $E10000
x&syscnt str '&msg'
end&syscnt anop
 MEND
```

The WINDOW.S1 and INITQUIT.S1 Programs

The WINDOW.S1 program, like the C language programs in the last few chapters, is divided into two parts: WINDOW.S1 and INITQUIT.S1. The WINDOW.S1 program, listing 10–5, and the INITQUIT.S1 program, listing 10–6, are at the end of this chapter.

Splitting a program into two or more parts can save a considerable amount of typing. For example, INITQUIT.S1—the portion of the program that loads, starts up, and shuts down tools—is also used in sample programs in chapters 11 and 12.

In programs written using the APW assembler-editor package, it's easy to divide a program into sections and then put all the sections together again at assembly time. All you have to do is type each section, save it as a separate source code file, and then combine the files you have saved using the APW assembler directive `COPY`. Look at the end of the WINDOW.S1 program in listing 10–5, and you'll see that the last line of the listing is

```
COPY INITQUIT.S1
```

When the APW assembler reaches that line, it starts assembling INITQUIT.S1 and adds it to WINDOW.S1, just as if the two listings were a single listing. Furthermore, any number of `COPY` directives can appear at the end of a source code listing. So you can add many modules to an APW program by using the `COPY` directive.

The WINDOW.C and INITQUIT.C Programs

The WINDOW.C program, listing 10–7, is a C language version of WINDOW.S1. It is designed to be used with the INITQUIT.C program, listing 10–8, which performs the same functions as INITQUIT.S1 and was introduced in chapter 9. THe WINDOW.C and INITQUIT.C programs appear at the end of this chapter.

WINDOW.C and INITQUIT.C are combined into one program with the statement

```
#include"initquit.c"
```

This statement is in the first line of the WINDOW.C program.

There are significant differences between WINDOW.C and its assembly language equivalent, WINDOW.S1. In WINDOW.C, for example, the `Sketch()` function, which draws on the screen, is simplified. It uses the function `StartDrawing()` just once, then it uses `SetPort()` thereafter. This is a more streamlined way to write the `Sketch()` routine in C, but the method used in WINDOW.S1 works better in assembly language. Experiment and you'll see why.

In WINDOW.C, the `ErasePic0()` function, which is called `repaint` in WINDOW.S1, is also simplified. Instead of completely dismantling a window environment and then rebuilding it (the technique used in WINDOW.S1) the `ErasePic0()` function keeps the window's environment, but simply erases what is in it. Because of differences in the way in which WINDOW.S1 and WINDOW.C work, this is another approach that works well in C, but the technique used in WINDOW.S1 works better in assembly language.

WINDOW.S1 and INITQUIT.S1 Listings

Listing 10–5
WINDOW.S1 program

```
*
*  WINDOW.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'Window'
                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy window.macros

                KEEP window
```

```
*
*  EXECUTABLE CODE STARTS HERE
*

Begin           START
                Using QuitData

                jmp MainProgram          ; skip over data

                END

*
*  SOME DIRECT PAGE ADDRESSES AND A FEW EQUATES
*

DPData          START

DPTemp          gequ    $00
DPPointer       gequ    DPTemp+4
DPHandle        gequ    DPPointer+4

ScreenMode      gequ $00                 ; 320 mode
MaxX            gequ 320                 ; X clamp high

True            gequ $8000
False           gequ $00

                END

*
*  MAIN PROGRAM LOOP
*

MainProgram     START
                Using GlobalData
                Using PortData

                phk
                plb
                tdc                      ; get current direct page
                sta MyDP                 ; and save it for the moment

                jsr ToolInit             ; start up all tools we'll need
                jsr BuildMenu            ; create and draw menu bar
                jsr MakeWin0             ; create empty window
```

```
*** OPEN A PORT SO WE CAN DRAW IN WINDOW'S PIXEL MAP ***

                jsr NewPort

                lda #PicOPort
                sta BlkToFill
                lda #^PicOPort
                sta BlkToFill+2

                jsr BlkFill

*** LINE THAT JUMPS TO THE EVENT LOOP ***

                jsr EventLoop           ; check for key & mouse events

*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

                END

*
*  EVENT LOOP
*

EventLoop       START
                Using QuitData
                Using TaskTable
                Using EventData

Again           anop
                PushWord #0             ; space for result
                PushWord #$FFFF         ; recognize all events
                PushLong #EventRecord
                _TaskMaster
                pla
                asl a                   ; code * 2 = table location
                tax                     ; X is index register
                jsr (TaskTable,x)       ; look up event's routine
                lda QuitFlag
                beq again

                rts

                END
```

```
*
*  ROUTINE TO DRAW SKETCHES ON THE SCREEN
*
MoveIt          START
                Using EventData
                Using GlobalData
                Using PortData

                PushLong TaskData
                _StartDrawing
                PushLong #RectPtr
                _GetPortRect
                PushLong #EventWhere      ; convert them to
                _GlobalToLocal            ; local coordinates
                PushLong EventWhere       ; move cursor to mouse location
                _MoveTo

                pea 0
                pea 0
                _SetOrigin

                PushLong #PicOPort
                _SetPort

                PushLong #RectPtr
                _ClipRect

                PushLong EventWhere
                _MoveTo

loop            pea 0                     ; space for return
                pea 0                     ; check button zero
                _StillDown
                pla
                beq out

                lda TaskData+2
                pha
                lda TaskData
                pha
                _StartDrawing

                lda #^EventWhere
                pha
                lda #EventWhere
                pha
                _GetMouse
```

```
                lda EventWhere+2
                pha
                lda EventWhere
                pha
                _LineTo

                pea 0
                pea 0
                _SetOrigin

                lda #^PicOPort
                pha
                lda #PicOPort
                pha
                _SetPort

                lda EventWhere+2
                pha
                lda EventWhere
                pha
                _LineTo

                brl loop

out             anop
                rts

RectPtr         ds 8

                END

*
*  REPAINT: MAKE NEW EMPTY WINDOW
*

Repaint         START
                Using PortData
                Using WindowData
                Using GlobalData

                PushLong #0
                _GetPort
                PullLong ThisPortPtr

                PushLong #PicOPort
                _SetPort

                PushLong #ScreenRect
```

```
                _ClipRect

                lda #PicOPort
                sta BlkToFill
                lda #^PicOPort
                sta BlkToFill+2

                jsr BlkFill

                PushLong ThisPortPtr
                _SetPort

                PushLong WinOPtr
                _HideWindow

                PushLong WinOPtr
                _CloseWindow

                PushLong PicOHandle
                _DisposeHandle

                jsr MakeWinO
                jsr doWinO

                rts

ThisPortPtr     ds 4
ScreenRect      dc i'0,0,200,320'

                END

NewPort         START
                Using GlobalData
                Using PortData

                PushLong #0             ; space for result
                _GetPort
                PullLong OrigPortPtr    ; save pointer to current port

                PushLong #PicOPort      ; pointer to new port
                _OpenPort               ; open a port for pixel map

                PushLong #PicOPort      ; make new port the current
                _SetPort                ; port (temporarily)

                PushLong #ScreenRect
                _ClipRect
```

```
                PushLong #PicOLocInfo   ; set up loc info for new port
                _SetPortLoc

                PushLong OrigPortPtr    ; make original port
                _SetPort                ; the current port again

                rts

ScreenRect      dc i'0,0,200,320'

                END

*
*  CREATE AND DRAW A WINDOW
*

MakeWinO        START
                using GlobalData
                using WindowData
                using PortData

*** SET HANDLE FOR PIC 0 (new) ***

                PushLong #$00
                PushLong #$8000         ; 32K (one screen)
                PushWord MyID
                PushWord #$C000         ; locked and fixed
                PushLong #0
                _NewHandle

                ErrorCheck 'Could not get handle.'

                pla
                sta PicOHandle
                pla
                sta PicOHandle+2

*** DEREF HANDLE, CLEAR MEMORY, AND CREATE POINTER ***

                lda PicOHandle          ; lock and deref PicOHandle
                ldx PicOHandle+2        ; while we do our thing with it
                jsr Deref

                sta PicOPtr             ; deref gives us a pointer
                stx PicOPtr+2           ; to PicOHandle's pixel map
*                                       ; so we'll save it
```

```
*** SET UP WINDOW 0 ***

                PushLong #0             ; space for result
                PushLong #Win0ParamBlock
                _NewWindow

                pla
                sta Win0Ptr
                pla
                sta Win0Ptr+2

                rts

                END

* DoWin0
* Selects and shows window 0 (blank) in response to menu selection.

DoWin0          START
                using GlobalData
                using WindowData

                PushLong Win0Ptr
                _SelectWindow

                PushLong Win0Ptr
                _ShowWindow

                rts

                END

*
* Paint0
* Draws empty window when TaskMaster calls.
*

Paint0          START
                using GlobalData
                Using PortData
                using WindowData

                phb
                phk
                plb

                phd
                lda MyDP
```

```
                tcd

                PushLong #PicOLocInfo
                PushLong #PicOFrame
                PushWord #0
                PushWord #0
                PushWord #0
                _PPToPort

                pld
                plb
                rtl

                END

***  BLOCK FILL ROUTINE ***

BlkFill         START
                Using GlobalData
                Using WindowData
                Using PortData

                PushLong #0
                _GetPort
                PullLong OrigPortPtr

                PushLong BlkToFill
                _SetPort

                PushWord #$FF
                _SetSolidPenPat

                PushLong #ARect
                _PaintRect

                _PenNormal

                PushLong OrigPortPtr
                _SetPort

                rts

OrigPortPtr     ds 4
ARect           dc i'0,0,200,320'

                END
```

```
*
*  CREATE AND DRAW MENU
*

BuildMenu       START
                using MenuData          ; proceeding from back to front

                PushLong #0             ; space for return
                PushLong #Menu3
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu2         ; 'wait' screen menu bar
                _NewMenu
                PushWord #0
                _InsertMenu

                PushLong #0             ; space for return
                PushLong #Menu1
                _NewMenu
                PushWord #0
                _InsertMenu

                PushWord #0             ; init & draw the menu bar
                _FixMenuBar
                pla                     ; discard menu bar height

                _DrawMenuBar

                rts

                END

*
* DoMenu
* Called when TaskMaster tells us a new menu item is selected.
*

DoMenu          START
                Using TaskTable
                Using EventData
                Using MenuTable

                lda TaskData
                cmp #256
                bcc GiveUp              ; this should never happen
```

```
                and #$00FF
                asl a
                tax

                jsr (MenuTable,x)

GiveUp          anop
                PushWord #False          ; draw normal
                PushWord TaskData+2      ; which menu
                _HiliteMenu

                rts

                END

*
* InsertSysDisk
* This routine is called when tools need to be loaded and the
* system disk is offline.  Routine asks user to insert system disk.
*

InsertSysDisk   START

                _SetPrefix SetPrefixParams
                _GetPrefix GetPrefixParams

                PushWord #0              ; space for result
                PushWord #195            ; x pos
                PushWord #30             ; y pos
                PushLong #PromptStr      ; prompt string
                PushLong #VolStr         ; vol string
                PushLong #OKStr
                PushLong #CancelStr
                _TLMountVolume

                pla

                rts

PromptStr       str 'Please insert the disk'

VolStr          ds 16

OKStr           str 'OK'

CancelStr       str 'Shutdown'

GetPrefixParams dc i'7'
```

```
                dc i4'VolStr'

SetPrefixParams dc i'7'
                dc i4'BootStr'

BootStr         str '*/'

                END

*
*  WINDOW GO-AWAY ROUTINE
*

doGoaway        START
                Using EventData

                PushLong TaskData
                _HideWindow
                rts

                END

*
*  A USEFUL AND CONVENIENT WAY NOT TO DO ANYTHING
*

Ignore          START

                rts

                END

*
* Deref
* Derefs the handle passed in a and x registers.
* Result passed back in a and x registers.
*

Deref           START
                sta DPTemp
                stx DPTemp+2
                ldy #2
                lda [DPTemp],y
                tax
                lda [DPTemp]
                rts

                END
```

```
*
*  DATA SEGMENTS
*

*
* Menu Data
*

MenuData        DATA

Return          equ 13

Menu1           dc c'>L@\XN1',i1'RETURN'
                dc c' LA Window Program \N257',i1'RETURN'
                dc c'.'

Menu2           dc c'>L File \N2',i1'RETURN'
                dc c' LNew \N258V',i1'RETURN'
                dc c' LQuit \N259',i1'RETURN'
                dc c'.'

Menu3           dc c'>L Windows \N3',i1'RETURN'
                dc c' LUntitled \N260',i1'RETURN'
                dc c'.'

                END
***

MenuTable       DATA

*               Menu 1 (apple)
                dc i'ignore'            ; one for the NDAs
                dc i'ignore'            ; 'a window program'

*               Menu 2 (file)
                dc i'Repaint'           ; 'doWin0' (new window)
                dc i'doQuit'            ; quit item selected

*               Menu 3 (windows)
                dc i'doWin0'            ; 'untitled'

                END
```

```
***

TaskTable       DATA

                dc i'ignore'            ; 0 null
                dc i'ignore'            ; 1 mouse down
                dc i'ignore'            ; 2 mouse up
                dc i'ignore'            ; 3 key down
                dc i'ignore'            ; 4 undefined
                dc i'ignore'            ; 5 auto-key down
                dc i'ignore'            ; 6 update event
                dc i'ignore'            ; 7 undefined
                dc i'ignore'            ; 8 activate
                dc i'ignore'            ; 9 switch
                dc i'ignore'            ; 10 desk acc
                dc i'ignore'            ; 11 device driver
                dc i'ignore'            ; 12 application
                dc i'ignore'            ; 13 application
                dc i'ignore'            ; 14 application
                dc i'ignore'            ; 15 application
                dc i'ignore'            ; 0 in desk

*
* TaskMaster events
*

                dc i'DoMenu'            ; 1 in menu bar
                dc i'ignore'            ; 2 in system window
                dc i'MoveIt'            ; 3 in content of window (MoveIt)
                dc i'ignore'            ; 4 in drag
                dc i'ignore'            ; 5 in grow
                dc i'doGoAway'          ; 6 in go-away
                dc i'ignore'            ; 7 in zoom
                dc i'ignore'            ; 8 in info bar

                dc i'ignore'            ; 9 in ver scroll
                dc i'ignore'            ; 10 in hor scroll
                dc i'ignore'            ; 11 in frame
                dc i'ignore'            ; in drop

                END

***

ToolTable       DATA

                dc i'8'                 ; number of tools in table
                dc i'$04,$0100'         ; quickdraw
```

```
                dc i'$06,$0100'         ; event manager
                dc i'$0E,$0000'         ; window manager
                dc i'$0F,$0100'         ; menu manager
                dc i'$10,$0100'         ; control manager
                dc i'$14,$0000'
                dc i'$15,$0000'
                dc i'$17,$0000'         ; std file manager

                END

***

EventData       DATA

EventRecord     anop                    ; table for Event Manager
EventWhat       ds 2
EventMessage    ds 4
EventWhen       ds 4
EventWhere      ds 4
EventModifiers  ds 2
TaskData        ds 4
TaskMask        dc i4'$0FFF'

                END

***

QuitData        DATA

QuitFlag        ds 2

QuitParams      dc  i4'0'
                dc  i4'0'
                dc  i4'0'

                END

***

WindowData      DATA

PicOHandle      ds  4
WinOPtr         ds  4
WinOTitle       str 'Untitled'
```

```
WinOParamBlock anop

          dc    i'WinOEnd-WinOParamBlock'
          dc    i2'%1101110111000000' ; Bits describing frame
          dc    i4'WinOTitle'         ; Pointer to title
          dc    i4'0'                 ; RefCon
          dc    i2'26,0,188,308'      ; Full Size (0= default)
          dc    i4'0'                 ; Color Table Pointer
          dc    i2'0'                 ; Vertical origin
          dc    i2'0'                 ; Horizontal origin
          dc    i2'200'               ; Data Area Height
          dc    i2'320'               ; Data Area Width
          dc    i2'200'               ; Max Cont Height
          dc    i2'320'               ; Max Cont Width
          dc    i2'2'                 ; No. of pixels to scroll vertically
          dc    i2'2'                 ; No. of pixels to scroll horizontally
          dc    i2'20'                ; No. of pixels to page vertically
          dc    i2'32'                ; No. of pixels to page horizontally
          dc    i4'0'                 ; Infomation bar text string
          dc    i2'0'                 ; Info bar height
          dc    i4'0'                 ; DefProc
          dc    i4'0'                 ; Routine to draw info bar
          dc    i4'Paint0'            ; Routine to draw content
          dc    i'26,0,188,308'       ; Size/position of content
          dc    i4'$FFFFFFFF'         ; Plane to put window in
          dc    i4'0'                 ; Address for window record (0 to
                                      ; allocate)
*
WinOEnd   anop

             END

***

GlobalData     DATA

BlkToFill      ds 4

MyID           dc  i'0'              ; program ID word
MyDP           ds  2

BlockSize      ds  4
FilVal         ds  2

               END
```

```
***
PortData        DATA

OrigPortPtr     ds 4                    ; pointer to original port

PicOPort        ds $AA

PicOLocInfo     dc i'$00'                 ; 320 mode
PicOPtr         ds 4                    ; MakeWinO fills this in
                dc i'160'               ; width
PicOFrame       dc i'0,0,200,320'         ; pic image frame rect

                END

*
* IOData
*
IOData          DATA

ReplyRecord     anop
GoodFlag        ds 2
FType           dc i'193'               ; $c1
AuxFType        dc i'0'                 ; #0
FName           ds 15
FullPathName    ds 128

CreateParams    anop
NameC      dc   i4'0'
           dc   i2'$00C3'               ; DRNWR
CType      dc   i2'$00C1'               ; super high-res graphics
CAux       dc   i4'$00000000'           ; Aux
           dc   i2'$0001'               ; type
           dc   i2'$0000'               ; create date
           dc   i2'$0000'               ; create time

DestParams      anop
NameD           dc    i4'0'

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
                ds 4

ReadParams      anop
ReadID          ds 2
PicDestIN       ds 4
                dc i4'$8000'          ; this many bytes
                ds 4                  ; how many xfered
```

```
WriteParams     anop
WriteID         ds 2
PicDestOUT      ds 4
                dc i4'$8000'        ; this many bytes
                ds 4                ; how many xfered

CloseParams     anop
CloseID         ds 2

                END

***

                COPY INITQUIT.S1
```

Listing 10–6
INITQUIT.S1 program

```
*
*  INITQUIT.S1: WHERE WE INITIALIZE OUR TOOLS
*

ToolInit        START
                Using GlobalData
                Using ToolTable

*** START UP TOOL LOCATOR ***

                _TLStartup              ; Tool Locator

*** INITIALIZE MEMORY MANAGER ***

                PushWord #0
                _MMStartup
                ErrorCheck 'Could not init Memory Manager.'

                pla
                sta MyID

*** INITIALIZE MISC. TOOL SET ***

                _MTStartup
                ErrorCheck 'Could not init Misc Tools.'

*** GET SOME DIRECT PAGE MEMORY FOR TOOLS THAT NEED IT ***

                PushLong #0             ; space for handle
                PushLong #$800          ; eight pages
```

```
                PushWord MyID
                PushWord #$C001          ; locked, fixed, fixed bank
                PushLong #0
                _NewHandle

                ErrorCheck 'Could not get direct page.'

                pla
                sta DPHandle
                pla
                sta DPHandle+2

                lda [DPHandle]
                sta DPPointer

*** INITIALIZE QUICKDRAW II ***

                lda DPPointer            ; pointer to direct page
                pha
                PushWord #ScreenMode     ; either 320 or 640 mode
                PushWord #160            ; max size of scan line
                PushWord MyID
                _QDStartup
                ErrorCheck 'Could not start QuickDraw.'

*** INITIALIZE EVENT MANAGER ***

                lda DPPointer            ; pointer to direct page
                clc
                adc #$300                ; QD direct page + #$300
                pha                      ; (QD needs 3 pages)
                PushWord #20             ; queue size
                PushWord #0              ; X clamp low
                PushWord #MaxX           ; X clamp high
                PushWord #0              ; Y clamp low
                PushWord #200            ; Y clamp high
                PushWord MyID
                _EMStartup
                ErrorCheck 'Could not start Event Manager.'

*** LOAD SOME TOOLS FROM RAM ***

LoadEmUp        PushLong #ToolTable
                _LoadTools
                bcc ToolsLoaded

                cmp #$45                 ; prodos error: vol not found
                beq doInsertDisk
```

```
                sec
                ErrorCheck 'Could not load tools.'

DoInsertDisk    anop
                jsr InsertSysDisk
                cmp #1
                beq LoadEmUp
                sec
                ErrorCheck 'Tool loading aborted.'

*** WINDOW MANAGER ***

ToolsLoaded     PushWord MyID
                _WindStartup
                ErrorCheck 'Could not Start Window Manager.'

                PushLong #$0000
                _Refresh

*** CONTROL MANAGER ***

                PushWord MyID
                lda DPPointer           ; DP to use = qd dp + $400
                clc
                adc #$400
                pha
                _CtlStartup
                ErrorCheck 'Could not start Control Manager.'

*** MENU MANAGER ***

                PushWord MyID
                lda DPPointer           ; DP to use = qd dp +$500
                clc
                adc #$500
                pha
                _MenuStartup
                ErrorCheck 'Could not start Menu Manager.'

                _ShowCursor

*** LINE EDIT ***

                PushWord MyID
                lda DPPointer
                clc
                adc #$600               ; qd dp + $600
                pha
```

```
                _LEStartup
                errorcheck 'Could not start up Line Edit.'

*** DIALOG MANAGER ***

                PushWord MyID
                _DialogStartup
                errorcheck 'Could not start Dialog Manager.'

*** STANDARD FILE MANAGER ***

                PushWord MyID
                lda DPPointer
                clc
                adc #$700                ; qd dp + $700
                pha
                _SFStartup
                errorcheck 'Could not start up SF Manager.'

                rts

                END

*
*  THE ROUTINE THAT ENDS THE PROGRAM
*

EndIt           START

                Using QuitData

                _Quit QuitParams

*** A QUIT CALL SHOULDN'T RETURN; IF IT DOES, WE'RE FINI ***

                ErrorCheck 'We returned from a quit call!'

                END

*
*  SHUT DOWN ALL THE TOOLS WE STARTED UP
*

ShutDown        START
                Using GlobalData
                Using WindowData

                _SFShutdown
```

```
                _DialogShutdown
                _LEShutdown
                _MenuShutDown
                _CtlShutDown
                _WindShutDown
                _EMShutDown
                _QDShutDown
                _MTShutDown

                PushLong DPHandle
                _DisposeHandle

                PushLong PicOHandle
                _DisposeHandle

                PushWord MyID
                _MMShutDown
                _TLShutDown

                rts

                END

*
*  ROUTINE THAT SETS THE QUIT FLAG
*

doQuit          START
                Using QuitData

                lda #$8000
                sta QuitFlag
                rts

                END
```

WINDOW.C and INITQUIT.C Listings

Listing 10–7
WINDOW.C program

```
#include "initquit.c"

************************************/
/* Data and routine to create menus */
/************************************/
```

```
/*  Set up menu strings. Because C uses \ as an escape character, we use
two when we want a \ as an ordinary character. The \ at the end of each
line tells C to ignore the carriage return. This lets us set up our items
in an easy-to-read vertical alignment.  */

char *menu1 = "\
>L@\XN1\r\
 LA Window Program \\N257\r\
.";

char *menu2 = "\
>L File \\N2\r\
 LNew  \\N258V\r\
 LQuit \\N259\r\
.";

char *menu3 = "\
>L Windows \\N3\r\
 LUntitled \\N260\r\
.";

#define QUIT_ITEM 259    /* these will help us check menu item numbers */
#define QUIT_ITEM 259    /* these will help us check menu item numbers */
#define NEW_ITEM 258
#define UNTIT_ITEM 260

BuildMenu()
{
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
   InsertMenu(NewMenu(menu1),0);
   FixMenuBar();
   DrawMenuBar();
}


/*************************************************************************/
* Data structures and routine to set up offscreen drawing environment */
/*************************************************************************/

LocInfo pic0LocInfo = { mode320,
                        NULL, /* space for pointer to pixel image */
                      160,  /* width of image in bytes = 320 pixels */
                      0,0,200,320  /* frame rect */
                    };

Rect screenRect = {0,0,200,320};
GrafPort pic0Port;
```

```
#define IMAGE_ATTR attrLocked+attrFixed+attrNoCross+attrNoSpec+attrPage

PicOSetup()  /* called once by MakeWindow at start of program */
{
GrafPortPtr thePortPtr;

   picOLocInfo.ptrToPixImage = *(NewHandle(0x8000L,myID,IMAGE_ATTR,NULL));
   thePortPtr = GetPort();
   OpenPort(&picOPort);
   SetPort(&picOPort);
   SetPortLoc(&picOLocInfo);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(thePortPtr);
}

/*****************************************************/
/* Data structures and routine to create window      */
/*****************************************************/

/* Initialize template for NewWindow  */

#define FRAME fQContent+fMove+fZoom+fGrow+fBScroll+fRScroll
+fClose+fTitle

ParamList template = { sizeof (ParamList),
                       FRAME,
                       "\pUntitled",   /* pointer to title */
                       0L,             /* RefCon           */
                       26,0,188,308, /* full size (0=default)  */
                       NULL,         /* use default ColorTable */
                       0,0,          /* origin           */
                       200,320,      /* data area height & width */
                       200,320,      /* max cont height & width  */
                       2,2,    /* ver & hor scroll increment */
                       20,32,  /* ver & hor page increment  */
                       NULL,   /* no info bar text string  */
                       0,      /* info bar height = none   */
                       NULL,   /* default def proc         */
                       NULL,   /* no info bar draw routine */
                       NULL,   /* draw content must be filled in
                        at run time */
                       26,0,188,308, /* starting content rect */
                       -1L,    /* topmost plane */
                       NULL    /* let window manager allocate record */
                     };
```

```
/* Window's draw content routine */
pascal void DrawContent()
{
    PPToPort(&picOLocInfo,&(picOLocInfo.boundsRect),0,0,modeCopy);
}

GrafPortPtr winOPtr;

MakeWindow() /* complete template, make (the window,
 and setup offscreen port */
{
    template.wContDefProc = DrawContent;
    winOPtr = NewWindow(&template);
    PicOSetup(); /* create offscreen image for use by DrawContent */
}

/*************************************************************************/
/* Main routine. Set up environment, call eventloop, and shut down */
/*************************************************************************/
main()
{
    StartTools();
    BuildMenu();
    MakeWindow();
    EventLoop();
    DisposeHandle(FindHandle(picOLocInfo.ptrToPixImage));
    ShutDown();
}

/*************************************/
/* Event loop and supporting routines */
/*************************************/
WmTaskRec   myEvent;
Boolean done = false;

EventLoop()
{
   myEvent.wmTaskMask = 0xOFFF;
   while(!done)
      switch ( TaskMaster(everyEvent,&myEvent)) {
         case wInMenuBar:
            DoMenus();
            break;
         case wInGoAway:
            HideWindow(winOPtr);
            break;
         case wInContent:
            Sketch();
```

```
    }
}

DoMenus()
{
Word *data = (Word *)&myEvent.wmTaskData; /* address of item id  */

    switch(*data) {
      case QUIT_ITEM:
         done = true;
         break;
      case NEW_ITEM:
         ErasePic0();
         HideWindow(win0Ptr);
         CloseWindow(win0Ptr);
         win0Ptr = NewWindow(&template);
      case UNTIT_ITEM:
         SelectWindow(win0Ptr);
         ShowWindow(win0Ptr);
         break;
    }
    HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu id */
}

ErasePic0()
{
GrafPortPtr oldPortPtr;

   oldPortPtr = GetPort();
   SetPort(&pic0Port);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(oldPortPtr);
}

Sketch() /* sketch into current port and into offscreen port */
{
Point mouseLoc;
GrafPortPtr thePortPtr = (GrafPortPtr)myEvent.wmTaskData;
Rect theRect;

   mouseLoc = myEvent.wmWhere;

   StartDrawing(thePortPtr);  /* set up correct drawing coordinate
    system */
   GetPortRect(&theRect);     /* copy current Port Rect          */
   GlobalToLocal(&mouseLoc);  /* get cursor pos in local coordinates */
```

```
    MoveTo(mouseLoc);          /* set pen position to mouse loc  */
    SetPort(&picOPort);        /* switch to offscreen port       */
    ClipRect(&theRect);        /* clip offscreen drawing to window's
     port rect */
    MoveTo(mouseLoc);          /* set offscreen pen to same location  */
    SetPort(thePortPtr);       /* switch back to window's port   */

    while (StillDown(0)) {
       GetMouse(&mouseLoc);     /* get new mouse coordinates */

       LineTo(mouseLoc);        /* draw line in both ports */
       SetPort(&picOPort);
       LineTo(mouseLoc);
       SetPort(thePortPtr);
    }
    SetOrigin(0,0);             /* restore normal coordinates */
}
```

Listing 10–8
INITQUIT.C program

```
#include <TYPES.H>
#include <LOCATOR.H>
#include <MEMORY.H>
#include <MISCTOOL.H>
#include <QUICKDRAW.H>
#include <EVENT.H>
#include <CONTROL.H>
#include <WINDOW.H>
#include <MENU.H>
#include <LINEEDIT.H>
#include <DIALOG.H>

#define MODE mode320 /* 640 graphics mode def. from quickdraw.h */
#define MaxX 320     /* max X for cursor (for Event Mgr) */
#define dpAttr attrLocked+attrFixed+attrBank /* for allocating direct page
 space */

int  myID;           /* for Memory Manager */
Handle zp;           /* handle for page 0 space for tools */

int ToolTable[] = {7,
                   4, 0x0100,  /* QD        */
                   6, 0x0100,  /* Event     */
                  14, 0x0100,  /* Window    */
                  16, 0x0100,  /* Control   */
                  15, 0x0100,  /* Menu      */
```

```
                20, 0x0100,  /* Line Edit */
                21, 0x0100,  /* Dialog    */
                };


StartTools()        /* start up these tools: */
{
   TLStartUp();             /* Tool Locator */
   myID = MMStartUp();      /* Mem Manager  */
   MTStartUp();             /* Misc Tools */
   LoadTools(ToolTable);    /* load tools from disk */
   ToolInit();              /* start up the rest    */
}


ToolInit()      /* init the rest of needed tools */
{
   zp = NewHandle(0x700L,myID,dpAttr,0L);   /* reserve 6 pages */


   QDStartUp((int) *zp, MODE, 160, myID);    /* uses 3 pages */
   EMStartUp((int) (*zp + 0x300), 20, 0, MaxX, 0, 200, myID);
   WindStartUp(myID);
   RefreshDesktop(NULL);
   CtlStartUp(myID, (int) (*zp + 0x400));
   MenuStartUp(myID, (int) (*zp + 0x500));
   LEStartUp(myID, (int) (*zp + 0x600));
   DialogStartUp(myID);
   ShowCursor();
}


ShutDown()          /* shut down all of the tools we started */
{
   GrafOff();
   DialogShutDown();
   LEShutDown();
   MenuShutDown();
   CtlShutDown();
   WindShutDown();
   EMShutDown();
   QDShutDown();
   MTShutDown();
```

```
    DisposeHandle(zp); /* release our page 0 space */
    MMShutDown(myID);
    TLShutDown();
}
```

CHAPTER 11

Dialog with a IIGS

Using the Dialog Manager

The main channel of communication between the Apple IIGS and its user is handled by a tool set known as the Dialog Manager. When a program needs to inform the user of something important or give the user guidance—or when a program needs to obtain information from the user—the Dialog Manager provides the interface between computer and user.

The Dialog Manager communicates with the IIGS user through *dialog windows*—boxes that are usually programmed to appear on the screen when they are needed. Dialog windows can display messages, obtain user input, or both. They can contain icons, pictures, text, and user-operated controls. Some icons can stay on the screen for a long time and can be moved around. Others remain in one spot until they are deactivated and then go away as quickly as they appeared.

In this chapter, you'll take a look at various kinds of dialog windows, and you'll see how dialogs can be used in IIGS programs.

What Dialog Windows Look Like

Dialog windows resemble ordinary document windows, but they have controls that ordinary windows usually do not have. A dialog window usually appears near the top of the screen, in the center of the screen and slightly below the

menu bar, and is somewhat narrower than the screen. Figure 11–1 shows a typical dialog window.

As figure 11–1 shows, a dialog window looks something like a printed form. Like a paper form, a dialog can contain messages, illustrations, and blanks to be filled in by the user. These features can be presented in many formats, such as

- Messages designed to provide the user with information, instructions, or alerts.
- Controls such as buttons, scroll bars, and squares that can be checked off by the user. Text messages may or may not be supplied along with these controls.
- Rectangles in which the user may type in text. These rectangles, called edit lines, may be blank when they appear on the screen or they may contain default text that can be edited by the user.
- Graphic symbols: either icons or pictures drawn using QuickDraw. Icons are easier to manage than QuickDraw pictures and are thus more commonly used. But there is no reason why a QuickDraw picture can't appear in a dialog window.
- Any other types of items an application can define.

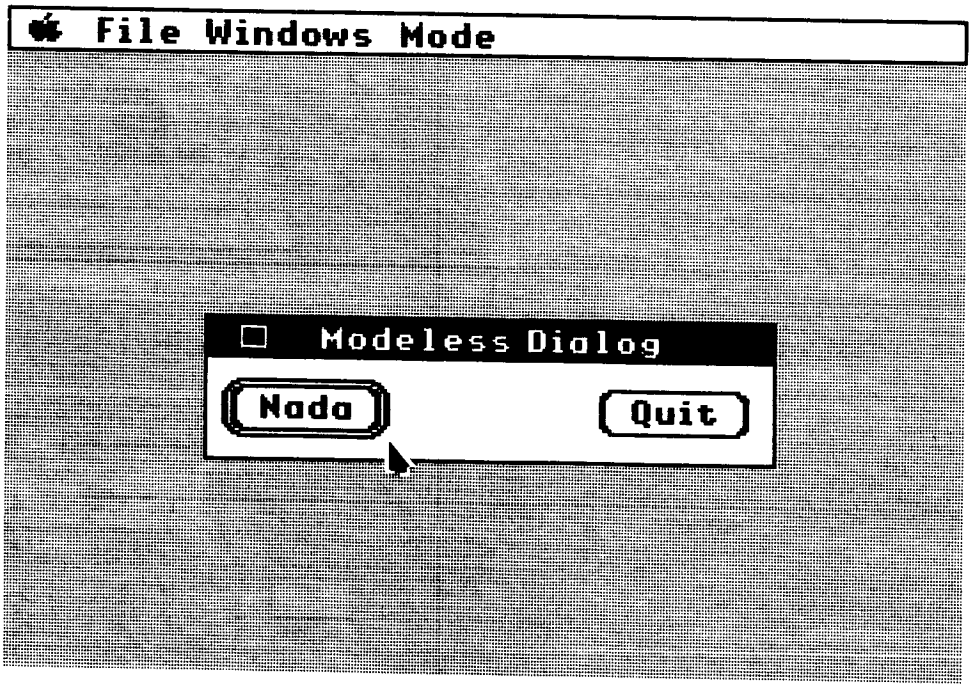


Figure 11–1
Typical dialog window

Dialog I/O

The simplest kind of dialog window is one that requires no response at all. Such a noninteractive dialog might be created to print a message on the screen while an application is performing a time-consuming process. When the operation is finished, the dialog could be removed from the screen.

Another simple type of dialog is one that contains just two items: a printed message and one button, often labeled OK, that the user can press after reading the message. In most cases, the dialog in which the message appears then disappears from the screen.

The button that makes the dialog disappear does not have to be labeled OK. It could be labeled Start or Proceed, or it could have another name. But, for simplicity, we call this button the OK button throughout this chapter.

Many kinds of dialog windows can be used in IIGS programs. Some dialog windows display more than one message on the screen, some display different messages at different times, and some accept input from the user. For example, if a dialog window appears on the screen as the result of some action by the user, it might contain a button labeled Cancel that is clicked to cancel the action that caused the dialog to appear. Or there could be a button labeled Help that is used to request additional information.

Dialog Items

In Dialog Manager jargon, buttons with labels like OK, Cancel, and Help are known as *dialog items*. There are many kinds of dialog items, and each is designed to be used in a slightly different way. Some dialog items provide information to the user, some obtain information from the user, and some do both. The items that can be used in dialog windows can be divided into the following categories:

- Button items. A button item is a simulated pushbutton that contains a label such as OK, Help, or Cancel. A button item usually has round corners and usually contains a label displayed in the standard IIGS type font, or system font. When the user clicks the IIGS mouse inside a button item, an application program can carry out whatever response is appropriate.
- Check items. A check item is a small square box that is empty or contains an *X*. When the user clicks the mouse in an empty check item, an *X* appears. When the user clicks the mouse in a check item that contains an *X*, the *X* disappears.

  A dialog box can contain any number of check items. When a dialog with a user ends, the application using the dialog can check to see which boxes have been checked and which have been left unchecked, and take the appropriate actions.

- Radio items. A radio item is a small circle that is empty or contains a still smaller circle. The inner circle in a radio item is usually black. When the user clicks the mouse in an empty radio item, an inner circle appears. When the user clicks the mouse in a radio item that contains an inner circle, the inner circle disappears.

- Scroll bar items. A scroll bar item is a special scroll bar used only in dialogs. A scroll bar item can be used to display the progress of an operation. For example, the white square, or "thumb" of a scroll bar, can move down the bar as files are printed to show the user how the operation is progressing.

- Static text items. A static text item, usually abbreviated StatText item, consists only of a Pascal-type string (a length byte followed by a string of ASCII characters). StatText items only display information; they cannot accept input from the user. Text in a StatText item does not have to be enclosed in a visible rectangle, and it cannot be edited.

- Long static text items. A long static text item, abbreviated LongStatText item, consists only of a block of text. The text in a LongStatText item is not preceded by a length byte, so its length must be passed to the Dialog Manager as a parameter when the item is created with a `NewDItem` call. More about this call is provided later in this chapter. LongStatText items only display messages; they cannot accept input from the user. Text in a LongStatText item does not have to be enclosed in a visible rectangle, and it cannot be edited.

- Edit line items. An edit line item contains space for one line of text that is entered or edited by the user. The text usually appears inside a visible rectangle. When an edit line item appears on the screen, it is empty or contains default text. If it is empty, you can fill it in by typing information, and you can edit the information after it has been typed. If the item contains default text when it appears on the screen, that text can be edited by the user.

- Icon items. An icon item contains an icon. Icons used in dialog windows are stored in memory in a specific format and appear in the dialog window when it is displayed on the screen. When the user clicks the mouse in an icon item, the application using the dialog can take whatever action is appropriate.

- Picture items. A picture item contains a picture drawn with QuickDraw II. When the user clicks the mouse in a picture item, the application using the dialog can take whatever action is appropriate.

- User items. Any item that is not in any of the previous categories is called a user item. User items are defined by application programs.

Types of Dialog Windows

There are three kinds of dialog windows: modal dialogs, modeless dialogs, and alert dialogs. Let's take a closer look at each of these types of dialog windows.

Modal Dialogs

Modal dialogs require the user to respond to a dialog message before taking any other action. Modal dialogs derive their name from the fact that they put a program in a state, or mode, of being unable to take any action outside a dialog window. A modal dialog usually has at least one button item that is clicked to perform some action and a Cancel button that is clicked to make the dialog box go away. Normally, clicking the mouse anywhere outside the dialog window only makes the IIGS speaker beep.

In programs written according to Apple's *Human Interface Guidelines,* one button item in a dialog window may be outlined in bold; that is, it may have a double outline. If such a button appears in a dialog box, it is usually the OK button, the button that ends the dialog by initiating some action and making the dialog window go away. When a button has a double outline, the Return key on the keyboard can always be pressed as an alternative to clicking the outlined button. In short, a button with a double outline is the dialog's default button—the safest button to use in the current situation. If there is no boldly outlined button, pressing the Return key will have no effect on the dialog. A typical modal dialog window is illustrated in figure 11–2.

Modeless Dialogs

A dialog cannot be modal and modeless at the same time; different routines create these two types of dialogs. When a program is running, however, it can be difficult to distinguish between a modal dialog and a modeless dialog because they often look alike.

A *modeless dialog,* like a modal dialog, usually has an OK button (often doubly outlined) and a Cancel button. And, just like a modal dialog, a modeless dialog can contain other controls that do not erase the dialog window and do not result in any change in a program until an OK button is pressed to make the dialog go away.

But modeless dialogs do not put a program into any special state, or mode, and thus do not require the user to respond to a dialog before taking any other action. When a modeless dialog is on the screen, it can stay there while the user performs actions unrelated to the dialog. For example, the user might be permitted to work in various windows on the desktop before clicking a button in the dialog window.

Because a modeless dialog can remain on the screen while document windows (or even other dialog windows) are in use, you can create a modeless dialog window that has a title bar and thus can be moved on the screen. Because of this feature—and because they can stay on the screen while various operations take place—modeless dialogs are used as desk accessories. Clocks, calculators, notepads, and other desk accessory items are often incorporated into programs in the form of modeless dialogs.

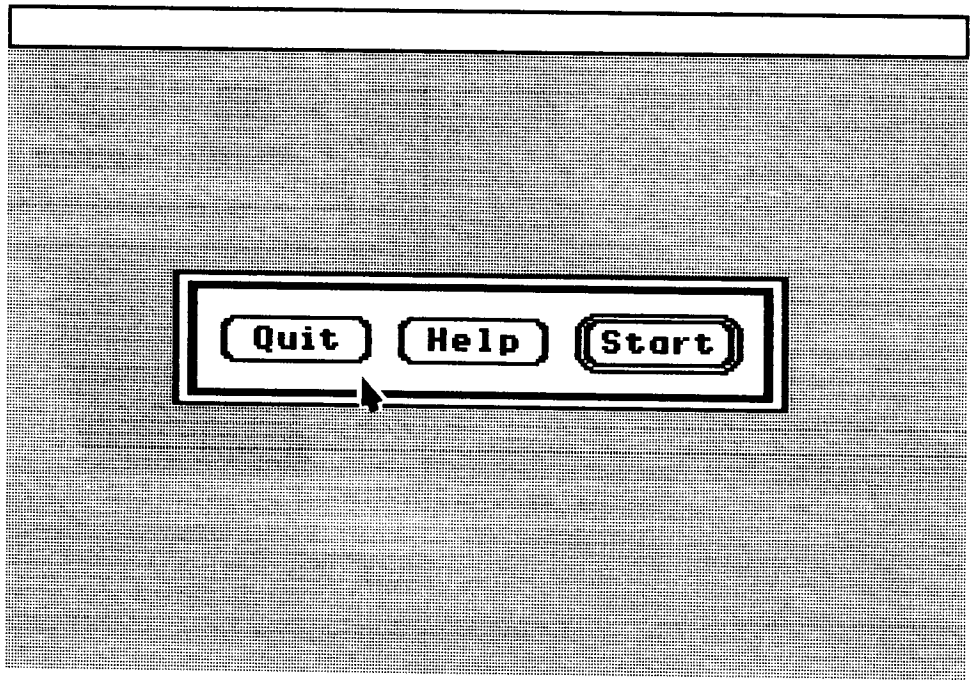


Figure 11–2
Modal dialog window

Figure 11–3 shows a modeless dialog box that is similar to a document window. Like a standard document window, it has both a title bar and a close box. So it can be moved, hidden, closed, and opened again, like any other similarly equipped window.

Alert Dialogs

An *alert dialog* looks much like a modal dialog (or a modeless dialog without a title bar). But an alert dialog has a special function. It appears only when something has gone wrong or when something important must be brought to the user's attention. Alert dialogs can provide a program with a convenient method for reporting errors or issuing warnings.

An alert window is usually placed slightly farther below the menu bar than a modal or modeless dialog. And an alert dialog often contains an icon that gives the user a visual clue about the nature of the alert. There are three standard types of alert icons: Stop, Note, and Caution. You can also design other kinds of icons. An alert dialog can also be programmed to beep or make other sounds when it is activated.

To help the user who isn't sure how to proceed when an alert box appears, the button used most often in the current situation is displayed with a double outline. This button is also the alert's default button. If the user presses the Return key, the effect is the same as clicking the alert's default button.

One special feature of an alert dialog is that it can behave in a different way each time it is activated. This feature can give the user increasingly

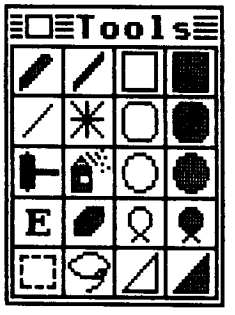


Figure 11–3
Modeless dialog window

severe warnings each time an error is made or a dangerous situation becomes more dangerous. For example, the first time an error is made, the error might beep the speaker but generate no alert box. Thereafter, each successive error might cause an alert dialog to be displayed, and each alert might carry an increasingly severe warning.

Furthermore, the sound produced by an alert dialog does not have to be a beep. It can be any sequence of tones, which may occur either by themselves or with an alert dialog. Figure 11–4 is an illustration of a typical alert dialog window.

Manipulating Dialog Windows

After a modal or modeless dialog is created, it can be manipulated like any other window. With the help of routines provided by the Window Manager and QuickDraw, an application can do just about anything to a dialog window: show, hide, or move it, change its size or plane, or close and discard it when it is no longer needed. The Dialog Manager even recognizes the `ClipRgn` field of the dialog window's GrafPort, so the QuickDraw II `SetClipRgn` and `ClipRect` routines can keep portions of a window from being displayed on the screen.

When an alert window is designed, however, the Dialog Manager takes care of most details, so that all alert windows have a standard appearance and behavior. The size and location of the box are supplied as part of the definition of the alert and are not changed easily. You do not have to specify an alert window's plane because an alert always appears in front of all other windows. After an alert window is on the screen, the application that uses it never has to manipulate it. That's because an alert window requires the user to respond before doing anything else, and the user's response makes the box disappear.

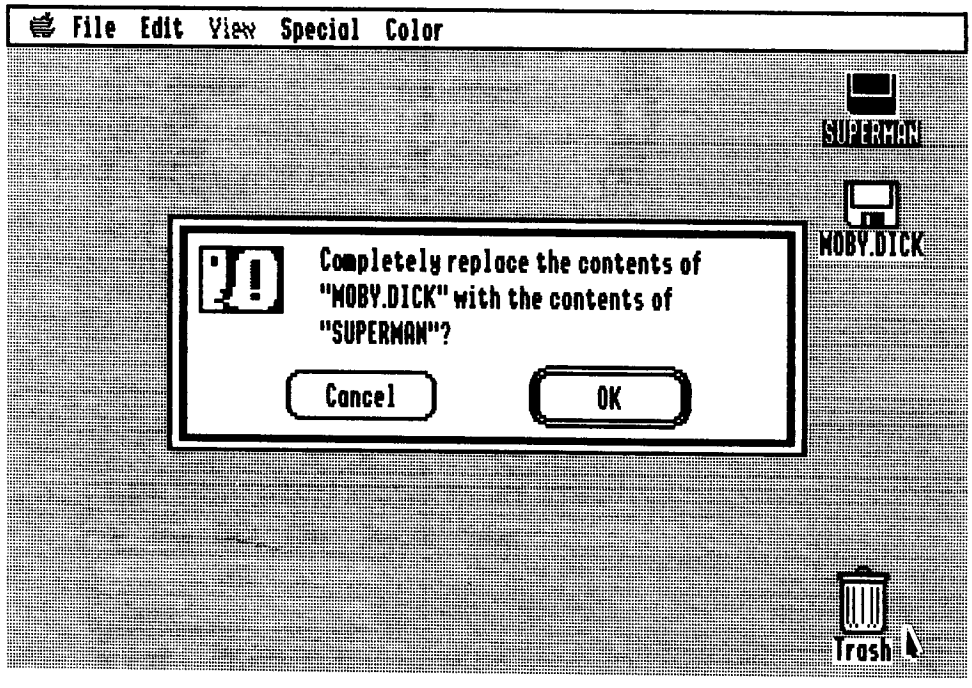


Figure 11–4
Alert dialog window

Initializing the Dialog Manager

Before the dialog is started, the following tool sets must be loaded and started:

- Tool Locator (always loaded and active)
- Memory Manager
- Miscellaneous Tool Set
- QuickDraw II
- Event Manager
- Window Manager
- Control Manager
- LineEdit Tool Set

After these tools are loaded and initialized, the `DialogStartUp` call can be made to start up the Dialog Manager. If you want the type font used in your dialog and alert windows to be something other than the system font, you can make the Dialog Manager call `SetDAFont`.

When the Dialog Manager is loaded and started up, the `NewModalDialog`, `NewModelessDialog`, and `GetNewModalDialog` calls can be used to create dialog windows. `NewModelessDialog` creates a dialog using a special kind of dialog record, and `GetNewModelessDialog` creates a dialog using a template that can be accessed by more than one dialog window.

After a dialog is set up, the NewDItem and GetNewDItem calls can be used to create the items that appear in each dialog. The CloseDialog call can be used to close and dispose of any dialogs.

Creating a Dialog Window

The Dialog Manager requires the same kind of information to create a dialog that the Window Manager requires to create a document window. These are the steps that are usually used to set up a dialog window:

1. The application calls NewModalDialog, GetNewModalDialog, or NewModelessDialog. In addition to creating a dialog window, these calls determine how the window looks and behaves.
2. The Dialog Manager must be supplied with a rectangle that becomes the port rectangle of the window's GrafPort.
3. The Dialog Manager must be told whether the window will be visible or invisible when it is created. If it is created as a visible window, it appears on the screen immediately. If it is created as an invisible window, the Window Manager calls SelectWindow and ShowWindow must be made each time the window appears on the screen.

If a modeless dialog is created, the plane in which it appears in relation to other windows must also be specified. By convention, a newly created window always appears in the frontmost plane.

The example program in this chapter, DIALOG.S1, uses the call NewModalDialog to create a modal dialog window. Listing 11–1 shows how NewModalDialog is used in the program. Instructions for typing and compiling the DIALOG.S1 program in both assembly language and C are at the end of this chapter.

Listing 11–1
Calling the NewModalDialog routine

```
PushLong #0                ; output
PushLong #DRect
PushWord #True             ; visible
PushLong #0                ; refcon
_NewModalDialog

pla
sta MDialogPtr
pla
sta MDialogPtr+2
```

As listing 11–1 shows, the **NewModalDialog** call takes four parameters:

- 2 null words (zeros), which provide a space on the stack for a 2-word result.
- A pointer to a rectangle that defines the location of the dialog window on the screen.
- A 1-word space for a Boolean value. If the value is nonzero, or true, the dialog is displayed on the screen as soon as it is created. If the value is zero, the window is not displayed until a specific command, such as **ShowWindow**, is called to display it on the screen.

When a **NewModalDialog** call returns, a pointer to the dialog window which it created is on the stack. In the DIALOG.S1 program, this pointer is stored in the **MDialogPtr** variable.

Creating an Item List

Before a dialog window can be displayed on the screen, the **NewDItem** call must be used to create each item that will appear in the window. The dialog window in the DIALOG.S1 program contains three buttons: Start, Quit, and Help. Listing 11–2 shows how the **NewDItem** call creates the Start button.

Listing 11–2
NewDItem call

```
PushLong MDialogPtr       ; item belongs to this window
PushWord #1               ; item ID number
PushLong #ButtonRect1     ; pointer to button's rect
PushWord #ButtonItem      ; item type
PushLong #ButtonText1     ; item descriptor
PushWord #0               ; item's initial value
PushWord #0               ; visible/invis flag
PushLong #0               ; color table pointer
_NewDItem
```

As listing 11–2 shows, the **NewDItem** call takes eight parameters:

- A pointer to the window to which the item belongs.
- A 1-word identification number that will be used in all dialog-related items to identify the item being created.
- A pointer to a rectangle that defines where the item will appear inside its dialog window. Note that this rectangle is expressed not in screen coordinates, but in local coordinates that treat the dialog window as a bounds rectangle.

- A 1-word parameter identifying the type of item being created. This parameter is a constant that can be found in APW's LIBRARIES/AINCLUDE file, under the filename E16.DIALOG. In the DIALOG.S1 program, the constants for item types are listed in the `DialogData` data segment.

  By convention, the OK button in an alert's item list is always assigned an ID of 1, and the Cancel button should always have an ID of 2. The Dialog Manager provides predefined constants equal to the item ID for OK and Cancel as follows:

```
OK          equ         1
Cancel      equ         2
```

  In a modal dialog's item list, the item whose ID is 1 is generally assumed to be the dialog's default button. If the user presses the Return key, the Dialog Manager normally returns the ID of the default button, just as when that item is actually clicked.

  To conform with Apple's *Human Interface Guidelines,* the Dialog Manager automatically prints a double outline in bold around the default button, unless there is no default button—that is, no button item with an ID number of 1. So, if you don't want a dialog to have a default button, you should not assign any button an ID number of 1. The item types listed in the DIALOG.S1 program are shown in listing 11–3.

- A two-word parameter called a dialog item descriptor. The function of this parameter can vary, depending upon the type of item being created. Table 11–1 shows the functions the item descriptor parameter can have when used with different kinds of items.
- A one-word parameter setting the initial value of the item descriptor, if applicable.
- A flag determining whether the item being created should be visible or invisible when the window is first displayed. This parameter can also include item-specific information, for example, the family number of a radio button or whether a scroll bar is horizontal or vertical. Further information on item-specific data in this parameter is in the *Apple IIGS Toolbox Reference.*
- A pointer to a color table, which can be used to change the standard colors used to draw items in a dialog. Custom color tables can be used for standard or custom-designed controls. But make sure your use of color conforms to Apple's *Human Interface Guidelines.*

Listing 11–3
Item types in DIALOG.S1

```
DialogData      DATA

ButtonItem      equ   10
CheckItem       equ   11
RadioItem       equ   12
```

```
ScrollBarItem   equ   13
UserCtlItem     equ   14
StatText        equ   15
EditText        equ   16
EditLine        equ   17
IconItem        equ   18
PicItem         equ   19
UserItem        equ   20

                END
```

Table 11–1
Item Descriptor Parameter in a NewDItem Call

Type	Function of Descriptor	Value
`ButtonItem`	Pointer to a string containing item's label	N/A
`CheckItem`	N/A	0 = not checked 1 = checked
`RadioItem`	N/A	0 = not checked 1 = checked
`ScrollBarItem`	Pointer to dialog scroll bar action procedure	0 or default value if `ItemDescr` = 0
`UserCtlItem`	Pointer to control definition procedure	Initial value of control
`UserCtlItem2`	Pointer to parameter block	Initial value of control
`StatText`	Pointer to static string	Application use
`LongStatText`	Pointer to the beginning of text	Length of text (0 to 32,767)
`EditLine`	Pointer to default string	Maximum length allowed (0 to 255)
`IconItem`	Handle to the icon	Application use
`PicItem`	Handle to the picture	Application use
`UserItem`	Pointer to item definition procedure	Application use

Using a Dialog Window in a Program

When a modal dialog is created, the `ModalDialog` call can be used to accept user input. Listing 11–4 shows how the `ModalDialog` call is used in the DIALOG.S1 program. Let's take a look now at how the routine in listing 11–4 works. Then we'll see how the routine is used in the DIALOG.S1 program.

Listing 11–4
ModalDialog call

```
Again       PushWord #0              ; space for result
            PushLong #0              ; filter procedure pointer
            _ModalDialog
            pla

next        cmp #3
            beq Again

            cmp #1
            beq noquit

button2     lda #$FFFF               ; button 2 was pressed

            sta QuitFlag

noquit      PushLong MDialogPtr      ; use this exit for #1 or #3
            _CloseDialog

            rts
```

The `ModalDialog` call takes two parameters: a 1-word null (zero) value that saves a space on the stack and a pointer to a user-written filter procedure, if there is one. A filter procedure, usually abbreviated `FilterProc`, is a routine that an application can call to filter out unwanted responses by the user (for example, to ignore non-numeric characters typed in an `EditLine` item that calls for numeric characters only). If a 0 is passed to `ModalDialog` in the `FilterProc` parameter, it means no filter process is set up by the application using the dialog. In that case, `ModalDialog` will not look for one.

In the DIALOG.S1 program, `ModalDialog` is called with two 0 parameters: a null word to save a space on the stack and a null pointer because there is no filter procedure in the program.

When a `ModalDialog` call returns, a 1-word value—the ID number of the item selected by the user—is pushed on the stack. In the DIALOG.S1 program, this value is pulled off the stack and compared with the literal values 3 and 1. If the value is 3—the item ID number for the Help button—the program loops back to the line labeled `Again`. That's because no help function is written for the DIALOG.S1 program. If you expand the program, you may want to write a help function.

If the `ModalDialog` call returns a value of 1—the item ID number of the Start button—the dialog is erased from the screen with a `CloseDialog` call and the DIALOG.S1 program continues, as though there had never been a dialog window on the screen.

If the routine in listing 11–4 discovers that the user has clicked a button that is neither item 1 nor item 3, it is smart enough to determine that the user

If the routine in listing 11–4 discovers that the user has clicked a button that is neither item 1 nor item 3, it is smart enough to determine that the user has made the only other choice, item 2. This is the Quit button, which ends the program by storing a nonzero value in the program's `quit` flag before returning.

The DIALOG.S1 Program

DIALOG.S1 is an expanded version of the WINDOW.S1 program in chapter 10, so you can save yourself a lot of work by modifying WINDOW.S1 instead of typing the entire DIALOG.S1 program. To convert WINDOW.S1 into DIALOG.S1, the following modifications are necessary:

1. Replace the heading of the WINDOW.S1 program with the heading shown in listing 11–5.
2. Add three lines to the main program segment of the WINDOW.S1 program so that the segment looks like the one in listing 11–6.
3. Following the program segment labeled `EventLoop`, insert the segment that appears in listing 11–7. This segment displays a dialog window on the screen.
4. In the data segment labeled `MenuData`, change the line

   ```
   dc c' LA Window Program \N257',i1'RETURN'
   ```

   to

   ```
   dc c' LA Dialog Program \N257',i1'RETURN'
   ```

5. At the end of the program, add the data segment that appears in listing 11–8. This segment provides the item codes used in the DIALOG.S1 program.
6. Before you assemble DIALOG.S1, make sure you have the latest version of INITQUIT.S1 saved on the same disk that holds your DIALOG.S1 source code. Then the `COPY` directive at the end of DIALOG.S1 will combine the DIALOG.S1 and INITQUIT.S1 programs.

When you've typed, assembled, and executed DIALOG.S1, you'll be ready to examine the portion that creates a dialog on the screen. Starting from the beginning of the DIALOG.S1 program, move down the listing until you see the label `Main Program`. Below that label look for this line:

```
jsr doDialog1
```

If you have typed and run the program, you should have no trouble figuring out what this line does. After all tools are initialized and an empty menu bar appears on the screen, the line `jsr doDialog1` simply places a

modal dialog on the screen and waits for the user's input. The user can do one of three things: click Start, which erases the dialog box and resumes execution of the DIALOG.S1 program, click Help, which won't do anything because there is no help routine, or click Quit, which ends the program.

Listing 11–5
Heading segment

```
*
*  DIALOG.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'Dialog'
                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy dialog.macros

                KEEP dialog
```

Listing 11–6
Main loop segment

```
*
*
*   MAIN PROGRAM LOOP

MainProgram     START
                Using GlobalData
                Using PortData

                phk
                plb
                tdc                     ; get current direct page
                sta MyDP                ; and save it for the moment

                jsr ToolInit            ; start up all tools we'll need

*** PUT DIALOG NO. 1 ON THE SCREEN ***

                jsr doDialog1

                jsr BuildMenu           ; create and draw menu bar
                jsr MakeWin0            ; create empty window
```

```
*** OPEN A PORT SO WE CAN DRAW IN WINDOW'S PIXEL MAP ***

                jsr NewPort

                lda #PicOPort
                sta BlkToFill
                lda #^PicOPort
                sta BlkToFill+2

                jsr BlkFill

*** LINE THAT JUMPS TO THE EVENT LOOP ***

                jsr EventLoop           ; check for key & mouse events

*** WHEN EVENT LOOP ENDS, WE'LL SHUT DOWN ***

                jsr Shutdown
                jmp Endit

                END
```

Listing 11–7
Dialog window segment

```
*
*   DODIALOG1: PRINT DIALOG NO. 1 ON THE SCREEN
*

doDialog1       START
                using GlobalData
                using WindowData
                using DialogData
                using QuitData

                PushLong #0             ; output
                PushLong #DRect
                PushWord #True          ; visible
                PushLong #0             ; refcon
                _NewModalDialog

                pla
                sta MDialogPtr
                pla
                sta MDialogPtr+2

                PushLong MDialogPtr     ; item belongs to this window
                PushWord #1             ; item ID number
```

```
                PushLong #ButtonRect1     ; pointer to button's rect
                PushWord #ButtonItem      ; item's id number
                PushLong #ButtonText1     ; item descriptor
                PushWord #0               ; item's initial value
                PushWord #0               ; visible/invis flag
                PushLong #0               ; color table pointer
                _NewDItem

                PushLong MDialogPtr
                PushWord #2
                PushLong #ButtonRect2
                PushWord #ButtonItem
                PushLong #ButtonText2
                PushWord #0
                PushWord #0
                PushLong #0
                _NewDItem

                PushLong MDialogPtr
                PushWord #3
                PushLong #ButtonRect3
                PushWord #ButtonItem
                PushLong #ButtonText3
                PushWord #0
                PushWord #0
                PushLong #0
                _NewDItem

Again           PushWord #0               ; space for result
                PushLong #0               ; filter procedure pointer
                _ModalDialog
                pla

next            cmp #3
                beq Again

                cmp #1
                beq noquit

button2         lda #$FFFF                ; button 2 was pressed
                sta QuitFlag

noquit          PushLong MDialogPtr       ; use this exit for #1 or #3
                _CloseDialog

                rts

DRect           dc i'84,63,114,252'       ; screen coordinates
```

```
ButtonRect1     dc i'8,129,22,179'          ; local coordinates using
ButtonRect2     dc i'8,8,22,58'             ; dialog window's frame
ButtonRect3     dc i'8,67,22,117'           ; as a bounds rectangle

ButtonText1     str 'Start'
ButtonText2     str 'Quit'
ButtonText3     str 'Help'

MDialogPtr      ds 4

                END
```

Listing 11–8
DialogData segment

```
DialogData      DATA

ButtonItem      equ   10
CheckItem       equ   11
RadioItem       equ   12
ScrollBarItem   equ   13
UserCtlItem     equ   14
StatText        equ   15
EditText        equ   16
EditLine        equ   17
IconItem        equ   18
PicItem         equ   19
UserItem        equ   20

                End
```

The DIALOG.C Program

Listing 11–9, DIALOG.C, is a C language version of the DIALOG.S1 program. It is designed to be used with the `include` file INITQUIT.C, and it works just like DIALOG.S1.

Listing 11–9
DIALOG.C program

```
#include "initquit.c"

Boolean done = false;
WmTaskRec   my Event;
```

```
/************************************/
/* Data and routine to create menus    */
/************************************/
/* Set up menu strings. Because C uses \ as an escape character, we use
two when we want a \ as an ordinary character. The \ at the end of each
line tells C to ignore the carriage return. This lets us set up our items
in an easy-to-read vertical alignment. */

char *menu1 = "\
>L@\\XN1\r\
 LA Window Program \\N257\r\
.";

char *menu2 = "\
>L File \\N2\r\
 LNew \\N258V\r\
 LQuit \\N259\r\
.";

char *menu3 = "\
>L Windows \\N3\r\
 LUntitled \\N260\r\
.";

#define QUIT_ITEM 259    /* these will help us check menu item numbers */
#define NEW_ITEM 258
#define UNTIT_ITEM 260

BuildMenu()
{
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
   InsertMenu(NewMenu(menu1),0);
   FixMenuBar();
   DrawMenuBar();
}

/*************************************************************************/
/* Data structures and routine to set up offscreen drawing environment */
/*************************************************************************/
LocInfo picOLocInfo = { mode320,
                        NULL, /* space for pointer to pixel image */
                        160,  /* width of image in bytes = 320 pixels */
                        0,0,200,320  /* frame rect */
                      };

Rect screenRect = {0,0,200,320};
GrafPort picOPort;
```

```
#define IMAGE_ATTR attrLocked+attrFixed+attrNoCross+attrNoSpec+attrPage


PicOSetup()  /* called once by MakeWindow at start of program */
{
GrafPortPtr thePortPtr;

   picOLocInfo.ptrToPixImage = *(NewHandle(0x8000L,myID,IMAGE_ATTR,NULL));
   thePortPtr = GetPort();
   OpenPort(&picOPort);
   SetPort(&picOPort);
   SetPortLoc(&picOLocInfo);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(thePortPtr);
}

/*****************************************************/
/* Data structures and routine to create window      */
/*****************************************************/

/* Initialize template for NewWindow  */

#define FRAME fQContent+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose+fTitle

ParamList template = { sizeof (ParamList),
                       FRAME,
                       "\"Untitled",    /* pointer to title */
                       0L,             /* RefCon           */
                       26,0,188,308, /* full size (0=default)  */
                       NULL,         /* use default ColorTable */
                       0,0,          /* origin           */
                       200,320,      /* data area height & width */
                       200,320,      /* max cont height & width  */
                       2,2,    /* vertical & horizontal scroll increment */
                       20,32,  /* vertical & horizontal page increment */
                       NULL,   /* no info bar text string  */
                       0,      /* info bar height = none   */
                       NULL,   /* default def proc         */
                       NULL,   /* no info bar draw routine */
                       NULL,   /* draw content must be filled in at run time */
                       26,0,188,308, /* starting content rect */
                       -1L,    /* topmost plane */
                       NULL    /* let Window Manager allocate record */
                     };
```

```
/* Window's draw content routine */

pascal void DrawContent()
{
  PPToPort(&picOLocInfo,&(picOLocInfo.boundsRect),0,0,modeCopy);
}

GrafPortPtr winOPtr;

MakeWindow() /* complete template, make window, and set up offscreen port */
{
   template.wContDefProc = DrawContent;
   winOPtr = NewWindow(&template);
   PicOSetup(); /* create offscreen image for use by DrawContent */
}

/************************************************/
/* Data and routine to set up and display dialog */
/************************************************/

ItemTemplate item1 = {1,{8,129,22,179},buttonItem,"\pStart\r",0,0,NULL };
ItemTemplate item2 = {2,{8,8,22,58},buttonItem,"\pQuit\r",0,0,NULL };
ItemTemplate item3 = {3,{8,67,22,117},buttonItem,"\pHelp\r",0,0,NULL};

DialogTemplate dtemp = {{84,63,114,252},true,0L,&item1,&item2,&item3,NULL} ;

DoDialog()   /* Create and display an opening dialog box */
{
GrafPortPtr dlgPtr;
Word hit;

  dlgPtr = GetNewModalDialog(&dtemp);

  while ((hit = ModalDialog(NULL)) == 3);
  done = (hit == 2);
  CloseDialog(dlgPtr);
}

/*********************************************************************/
/* Main routine. Set up environment, call event loop, and shut down */
/*********************************************************************/

main()
{
   StartTools();
   DoDialog();
   BuildMenu();
```

```
    MakeWindow();
    EventLoop();
    DisposeHandle(FindHandle(picOLocInfo.ptrToPixImage));
    ShutDown();
}


/*************************************/
/* Event loop and supporting routines */
/*************************************/

EventLoop()
{
    myEvent.wmTaskMask = 0xOFFF;
    while(!done)
       switch ( TaskMaster(everyEvent,&myEvent)) {
          case wInMenuBar:
             DoMenus();
             break;
          case wInGoAway:
             HideWindow(winOPtr);
             break;
          case wInContent:
             Sketch();
      }
}

DoMenus()
{
Word *data = (Word *)&myEvent.wmTaskData; /*address of item id  */

     switch(*data) {
      case QUIT_ITEM:
          done = true;
          break;
       case NEW_ITEM:
          ErasePicO();
          HideWindow(winOPtr);
          CloseWindow(winOPtr);
          winOPtr = NewWindow(&template);
       case UNTIT_ITEM:
          SelectWindow(winOPtr);
          ShowWindow(winOPtr);
          break;
    }
    HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu id  */
}
```

```
ErasePic0()
{
GrafPortPtr oldPortPtr;

   oldPortPtr = GetPort();
   SetPort(&pic0Port);
   ClipRect(&screenRect);
   EraseRect(&screenRect);
   SetPort(oldPortPtr);
}

Sketch() /* sketch into current port and into offscreen port */
{
Point mouseLoc;
GrafPortPtr thePortPtr = (GrafPortPtr)myEvent.wmTaskData;
Rect theRect;

   mouseLoc = myEvent.wmWhere;

   StartDrawing(thePortPtr); /* set up correct drawing coordinate system */
   GetPortRect(&theRect); fr /* copy current Port Rect          */
   GlobalToLocal(&mouseLoc); /* get cursor pos in local coordinates */

   MoveTo(mouseLoc);         /* set pen position to mouse loc  */
   SetPort(&pic0Port);       /* switch to offscreen port      */
   ClipRect(&theRect);       /* clip offscreen drawing to window's Port Rect */
   MoveTo(mouseLoc);         /* set offscreen pen to same location  */
   SetPort(thePortPtr);      /* switch back to window's port   */

   while (StillDown(0)) {
      GetMouse(&mouseLoc);   /* get new mouse coordinates */

      LineTo(mouseLoc);      /* draw line in both ports */
      SetPort(&pic0Port);
      LineTo(mouseLoc);
      SetPort(thePortPtr);
   }
   SetOrigin(0,0);           /* restore normal coordinates */
}
```

CHAPTER 12

The Standard File Operations Tool Set

And ProDOS 16

Until the advent of the Apple IIGS, it could be difficult to incorporate disk drive operations into assembly language programs. Today, in programs written for the IIGS, the job is much easier. Here are four major reasons.

The Apple IIGS has new features that earlier Apple II computers do not have. For example, the Memory Manager tool set relieves the programmer of the responsibility of dealing with absolute addresses. It also has a new kind of I/O port, a SmartPort, which keeps track of the locations of disk drives and supports named devices and multiple, user-defined file prefixes.

The disk operating system in the IIGS is ProDOS 16—a 16-bit descendent of ProDOS 8, which was designed for the Apple IIe and the Apple IIc. ProDOS 16 is faster, more powerful, and easier to use than its 8-bit predecessor. And, unlike ProDOS 8, ProDOS 16 makes use of several new features of the IIGS.

The APW assembler-editor has a library of ProDOS macros that simplify the job of making ProDOS calls. In this chapter, you'll see how those macros are used.

The Standard File Operations Tool Set, which is included in the IIGS Toolbox, makes the task of working with ProDOS 16 even easier. When the Standard File Operations Tool Set is used in a program, a special dialog box is created every time a file is loaded or saved. You can load or save the file by either clicking the mouse inside a button item or typing the name of the file in a line edit control. You can also search through directories using the

Standard File Tool Set's dialog boxes, and you can even switch disks and change directories. The tool set gives the programmer the option of using predesigned dialog boxes or creating custom-designed boxes. Application programs can select the types of files that will or will not be listed on the screen.

In this chapter, you'll see how easy it is to create, load, save, and edit files using ProDOS 16, the ProDOS macros in the APW assembler-editor package, and the IIGS Standard File Operations Tool Set. These techniques are demonstrated using a sample program called SF.S1, which is listed at the end of this chapter. A C language version, SF.C, is also listed at the end of this chapter. Figure 12–1 shows the Standard File Tool Set screen display.

Introducing ProDOS 16

If you have written Apple II programs using ProDOS 8, you probably won't have any trouble understanding ProDOS 16. ProDOS 16 calls are made in the same way as ProDOS 8 calls: by filling in a block of parameters, pushing the address of the parameter block onto the stack, and jumping to a fixed entry point.

There are two important differences in the way calls are made in ProDOS 8 and ProDOS 16. In ProDOS 16, a program must jump to the ProDOS entry point with a `jsl` instruction rather than a `jsr` instruction, and the entry point is in bank $E1 rather than bank $00. In programs written using the APW

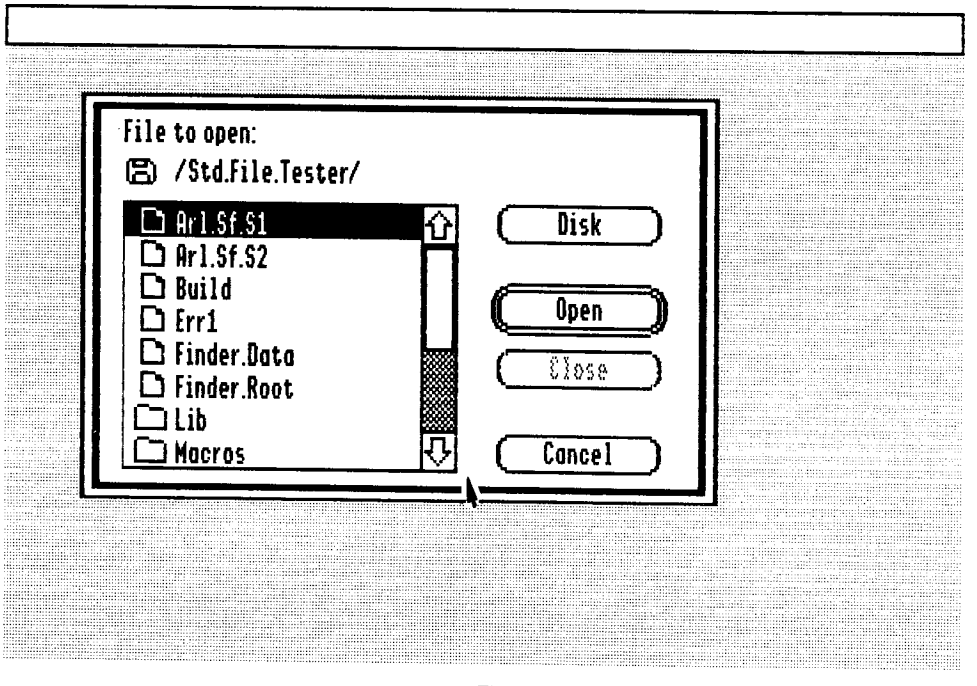


Figure 12–1
Standard File Operations Tool Set screen display

library of ProDOS macros, neither of these details makes any difference; the macros take care of them.

The kernel (or central part) of the Apple IIGS operating system is ProDOS 16, which is covered in detail in the *Apple IIGS ProDOS 16 Reference*. ProDOS accesses the disk drive or disk devices on which files are stored and retrieved and manages the creation and modification of files. ProDOS 16 also controls certain features of the IIGS operating environment, such as pathname prefixes and procedures for quitting programs and starting new ones.

ProDOS 16 can communicate with various disk drives, including hard disk drives, 5.25-inch floppy disk drives, and 3.5-inch disk drives. Because the IIGS has an intelligent disk port called a SmartPort, programs that use ProDOS 16 do not have to specify a disk's slot number or drive number to access the disk. Under ProDOS 16, a disk can also be accessed by its volume name or device name.

In ProDOS 16, just as in ProDOS 8, disks are also known as volumes, and information on a volume is divided into files. A file is an ordered sequence of bytes that has several attributes, including a name and a file type.

There are two primary types of files in ProDOS 16: standard files and directory files. Directory files contain the names and disk locations of other files. When a volume is formatted, a volume directory file is placed on it. The volume directory has the same name as the volume and usually contains the names and disk locations of other directory files.

ProDOS 16 supports a hierarchical file system. In a hierarchical file structure, volume directories can contain the names of other directories, called subdirectories, and subdirectories can, in turn, contain the names of other files or subdirectories.

In ProDOS 16, a file is identified by its pathname: a sequence of filenames starting with the name of the volume directory and ending with the name of the file. A pathname that begins with the name of a volume is a full pathname and is always preceded by a slash (/). If the name of the volume in which a file is stored is known, the file can be referenced by a partial pathname: a pathname that is not preceded by a slash and does not include a volume name.

Whether a pathname is preceded by a slash or not, the names of the directories, subdirectories, and files in the pathname are all separated by slashes. More details about pathnames are in the *Apple IIGS ProDOS 16 Reference*.

Loading a File with ProDOS 16

The SF.S1 program contains three code segments that make calls to ProDOS 16: `EndIt`, `LoadOne`, and `SaveOne`. `EndIt` makes the ProDOS call `Quit` to end the program. `LoadOne` appears in listing 12–1. `SaveOne` is explained shortly.

Listing 12–1
Loading a file using ProDOS 16

```
LoadOne         START
                using IOData

                _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open picture file.'

cont1           anop
                lda OpenID
                sta ReadID
                sta CloseID

                _Read ReadParams
                bcc cont2
                ErrorCheck 'Could not read picture file.'

cont2           anop
                _Close CloseParams

                clc
                rts

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
IOBuffer        ds 4

ReadParams      anop
ReadID          ds 2
PicDestIN       ds 4
                dc i4'$8000'            ; this many bytes
                ds 4                    ; how many xfered

CloseParams     anop
CloseID         ds 2

                END
```

In listing 12–1, the APW macro `Open` opens a file, the `Read` macro copies it into memory, and the `Close` macro closes it. In each of these calls, a label that identifies a parameter block is used as an operand. The parameter blocks used in the program appear at the end of the listing.

In the source code listing of the SF.S1 program, only one parameter—the number of bytes to be read into RAM—is filled in. When you run the

program, a segment of code called `ReadIt` fills in the other parameters. You'll examine the `ReadIt` segment later in this chapter.

As listing 12–1 shows, the ProDOS call `Open` takes three parameters:

- A 1-word file identification number that ProDOS assigns to the file being called when the `Open` call is made.
- A pointer to a string that contains the name of the file to be loaded. The string must be provided by the program using the `Open` call.
- A pointer to a 1,024-byte I/O buffer that ProDOS allocates when the call is made.

The ProDOS `Read` call takes four parameters:

- A 1-word file identification number. This is the ID number ProDOS assigns to the file when it is opened using an `Open` call.
- A pointer to a block of memory in which the file is stored. This block of memory must be provided by the application program making the `Read` call. In the SF.S1 program, the block is allocated using the Memory Manager call `NewHandle` in the segment of code labeled `MakeWin0`.
- A long word containing the number of bytes read into memory. In the SF.S1 program, $8000 bytes (or 32K) of memory are loaded into memory. This number was chosen because it is the length of the IIGS screen buffer and is thus the number of bytes required by one screenful of data.
- A long word that ProDOS fills in with the number of bytes actually transferred after the `Read` call is made.

When the file is read, a `Close` call should be issued to close the file. A `Close` call takes one parameter: the 1-word ID number assigned to the file when it is opened.

Saving a File with ProDOS 16

In the SF.S1 program, the code segment labeled `SaveOne` also makes a call to ProDOS 16. Listing 12–2 shows how ProDOS 16 can be used to save a program.

Listing 12–2
Saving a file using ProDOS 16

```
SaveOne         START
                using IOData
                _Destroy DestParams
                _Create CreateParams
                bcc cont0
                ErrorCheck 'Could not create pic file.'
```

```
cont0           _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open pic file.'

cont1           anop
                lda OpenID
                sta WriteID
                sta CloseID

                _Write WriteParams
                bcc cont2
                ErrorCheck 'Could not write to pic file.'

cont2           anop
                _Close CloseParams

                clc
                rts
DestParams      anop
NameD           dc   i4'0'

CreateParams    anop
NameC      dc   i4'0'
           dc   i2'$00C3'              ; DRNWR
CType      dc   i2'$00C1'              ; super high-res graphics
CAux       dc   i4'$00000000'          ; Aux
           dc   i2'$0001'              ; type
           dc   i2'$0000'              ; create date
           dc   i2'$0000'              ; create time

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
                ds 4

WriteParams     anop
WriteID         ds 2
PicDestOUT      ds 4
                dc i4'$8000'            ; this many bytes
                ds 4                    ; how many xfered

CloseParams     anop
CloseID         ds 2

                END
```

Five ProDOS 16 calls appear in listing 12–2. `Destroy`, `Create`, `Open`, `Write`, and `Close`. Let's take a closer look at each of these calls.

The `Destroy` call deletes a file. It is used in the SF.S1 program to delete one file so that another file can be created and placed in the RAM space left by the first one. The `Destroy` call takes just one paramater: the name of the file being deleted.

The `Create` call takes seven parameters:

- A pointer to a string that contains the name of the file being created. The string must be provided by the program using the `Create` call.
- A word whose bits contain information about how the file can be accessed. Only the low-order byte of this word is significant, and bits 2 through 4 are not used. The meanings of the other five bits are listed in table 12–1.
- A word identifying the file's file type. ProDOS 16 file types are listed in table 12–2.
- A long word identifying the file's auxiliary file type. Many applications use this field. For example, APW source files (file type $B0) use the auxiliary file type parameter to identify the language of a file—that is, whether it is a 65C816 assembly language file, a C file, an exec file, and so on. ProDOS 16 applies no restrictions to this parameter, however, and user-written applications may use it to distinguish between subtypes of files.
- A word identifying the file's storage type. This parameter identifies the level in the ProDOS hierarchy in which a file falls. Values that can be stored in this parameter, and their meanings, are listed in table 12–3. The values most commonly used in this parameter are $01 and $0D. More information on file storage types can be found in the *Apple IIGS ProDOS 16 Reference*.
- Create date: a word specifying the date on which a file was created. Bits 0 through 4 hold the day of the month, bits 5 through 8 hold the number of the month, and bits 9 through 15 hold the year. If no date is specified when a file is created, ProDOS 16 supplies the date from the system clock.
- Create time: a word specifying the time a file was created. Bits 0 through 5 hold the minute and bits 8 through 12 hold the hour. Bits 6, 7, and 13 through 15 are not used. If no date is specified when a file is created, ProDOS 16 supplies the date from the system clock.

An `Open` call must be issued before a file can be saved on a disk. You saw the parameters of an `Open` call previously, when you examined listing 12–1.

The ProDOS 16 call `Write` takes four parameters:

- A 1-word file ID number assigned when the file is opened.
- A pointer to the memory address of the information to be saved as a file.

Table 12–1
Access Byte in the Create Call

Bit	Name	Function	Value
7	D	Destroy enable bit	0 = File can't be destroyed 1 = File can be destroyed
6	RN	Rename enable bit	0 = File can't be renamed 1 = File can be renamed
5	B	Backup needed bit	0 = File backup is required 1 = Backup not required
4		Reserved	
3		Reserved	
2		Reserved	
1	W	Write enable bit	0 = File can't be written to 1 = File can be written to
0	R	Read enable bit	0 = File can't be read 1 = File can be read

- A long word holding the number of bytes to be saved.
- A long word in which ProDOS stores the number of bytes that have actually been transferred after the call is completed.

When you have finished saving a file, a `Close` call should be issued to close the file. A `Close` call takes one parameter: the 1-word ID number assigned to the file when it is opened.

Using the Standard File Tool Set

The Standard File Operations Tool Set, as noted, offers the IIGS user an easy and convenient method for loading and saving files—a collection of dialog boxes that can be programmed to appear on the screen when needed. These dialog boxes make loading and saving files as easy as clicking the mouse button. The Standard File Tool Set is even more of a timesaver for the IIGS programmer than it is for the IIGS user!

Before the Standard File Operations Tool Set is started up, the following tool sets must be loaded and initialized:

- Tool Locator (always loaded and active)
- Window Manager
- Control Manager
- Menu Manager
- LineEdit Tool Set
- Dialog Manager

When these tool sets are loaded and started up, the Standard File Tool Set can be initialized with the `SFStartup` call. Before a program that uses the tool set ends, `SFShutdown` should be called.

Table 12–2
ProDOS 16 File Types

Type	Name	Description
$00		Uncategorized file
$01	BAD	Bad block file
$02–03		Used by SOS (Apple III)
$04	TXT	ASCII text file
$05		Used by SOS (Apple III)
$06	BIN	Binary file
$07		Used by SOS (Apple III)
$08	FOT	Apple II graphics screen file
$09-$0E		SOS (Apple III) reserved
$0F	DIR	Directory file
$10-$18		Used by SOS (Apple III)
$19	ADB	AppleWorks database file
$1A	AWP	AppleWorks word-processor file
$1B	ASP	AppleWorks spreadsheet file
$1C-$AF		Reserved
$B0	SRC	APW source file
$B1	OBJ	APW object file
$B2	LIB	APW library file
$B3	S16	ProDOS 16 application program file
$B4	RTL	Run-time library
$B5	EXE	ProDOS 16 shell application file
$B6		ProDOS 16 permanent initialization file
$B7		ProDOS 16 temporary initialization file
$B8		New desk accessory (NDA)
$B9		Classic desk accessory (CDA)
$BA		Tool set file
$BB-$BE		Reserved for ProDOS 16 load files
$BF		ProDOS 16 document file
$C0-$EE		Reserved
$EF	PAS	Pascal area on a partitioned disk
$F0	CMD	ProDOS 8 CI added command file
$F1-$F8		ProDOS 8 user-defined files 1–8
$F9		ProDOS 8 reserved
$FA	INT	Integer BASIC program file
$FB	INV	Integer BASIC variable file
$FC	BAS	Applesoft BASIC program file
$FD	VAR	Applesoft BASIC variables file
$FE	REL	Relocatable code file (EDASM)
$FF	SYS	ProDOS 8 system program file

Table 12–3
File Storage Types

Value	Meaning
$00	Inactive entry
$01	Seedling file
$02	Sapling file
$03	Tree file
$04	Apple II Pascal region on a partitioned disk
$05	Directory file

Loading a File with the Standard File Tool Set

The easiest way to load a file using the Standard File Tool Set is with the `SFGetFile` call. The `SFGetFile` routine displays a standard, predesigned dialog box and allows the IIGS operator to use the dialog to open and load the selected file. With `SFGetFile`, the calling program can specify where the dialog box will be placed on the screen and the prompt that appears at the top of the box. The calling program can also filter the types of files to be displayed in the box. But the routine does not allow an application program to modify the appearance of the box. Programs that use a custom-designed dialog box must use another Standard File routine, `SFPGetFile`.

In the SF.S1 program, the `SFGetFile` call loads files into memory. Listing 12–3 shows the section of the program that uses the `SFGetFile` call.

The SFGetFile Call

As listing 12–3 illustrates, the `SFGetFile` call takes five parameters:

- A 1-word integer that specifies the horizontal screen coordinate of the upper left corner of the dialog box.
- Another 1-word integer that specifies the vertical screen coordinate of the upper left corner of the dialog box.
- A pointer to a Pascal-style string that is printed as a prompt inside the dialog box.
- A pointer to a "filter process" that can provide special instructions to the Dialog Manager about the handling of files. If such a process is used, it must be defined by the calling program. Instructions for designing a filter process are in the *Apple IIGS Toolbox Reference*. No filter process is used in the SF.S1 program.
- A pointer to a reply record, a specially designed record that the `SFGetFile` call fills with information before it returns. Listing 12–4 shows the reply record used in the SF.S1 program.

Listing 12–3
SFGetFile call in SF.S1

```
LoadIt          START
                using WindowData
                using IOData

                jsr Repaint

                PushWord #20              ; upper x coordinate
                PushWord #20              ; upper y coordinate
                PushLong #PromptPtr
                PushLong #0               ; no filter process
                PushLong #TypeListPtr     ; file types to display
                PushLong #ReplyRecord     ; defined in iodata
                _SFGetFile

                lda GoodFlag
                bne cont
                jmp return                ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda WinOHandle
                ldx WinOHandle+2
                jsr Deref
                sta PicDestIn
                stx PicDestIn+2
                jsr LoadOne

                PushLong NamePtr
                PushLong WinOPtr
                _SetWTitle               ; update window title

                lda WinOHandle
                ldx WinOHandle+2
                jsr Unlock

                PushLong NamePtr          ; update 'title' menu item
                PushWord #262             ; menu item number
                _SetMItemName             ; update name of item
                PushWord #0
                PushWord #0
                PushWord #3               ; menu number
                _CalcMenuSize             ; update width of items
```

```
return          rts

PromptPtr       str 'Load Picture:'

TypeListPtr     anop
NumEntries      dc i1'1'
Filetype1       dc h'c1'

                END
```

Listing 12–4
Reply record used by SFGetFile call

```
ReplyRecord     anop
GoodFlag        ds 2
FType           dc ds 2          ; in SF.S1, will always be $C1
AuxFType        dc i'0'          ; #0
FName           ds 15
FullPathName    ds 128
```

An `SFGetFile` reply record has five fields:

- A 1-word flag, called `GoodFlag` in the SF.S1 program, that holds a Boolean value. The flag is cleared to 0 if the user aborts the `SFGetFile` operation by pressing a Cancel button inside the dialog box. If the user does not press the Cancel button, the flag is set.
- A 1-word parameter that contains the type of file selected by the user. This parameter, like all other parameters in a reply record, is filled in by the `SFGetFile` call.
- A 1-word parameter that contains the auxiliary file type of the file selected by the user.
- A Pascal-style string that contains the name of the file selected by the user. The length of this parameter can be set by the application that calls `SFGetFile`. The most common length for this parameter is 15 bytes.
- Another Pascal-style string that contains the full pathname of the file selected by the user. The length of this parameter must be set by the application that calls `SFGetFile`. The recommended length for the parameter is 128 bytes.

All the information returned by the `SFGetFile` call is placed in its reply record; it does not push any values onto the stack.

In the SF.S1 program, a pointer to the file name returned by `SFGetFile` is loaded into the `NamePtr` variable. The handle of the screen buffer used in the program is then dereferenced (coverted into a pointer), and the `LoadOne` subroutine loads the file chosen by the user into the screen buffer.

Next, the program makes the Window Manager call `SetWTitle` to update the name of the window being displayed on the screen. Then the Menu Manager routines `SetMItemName` and `CalcMenuSize` replace the menu item `Untitled` with a menu item that displays the name of the selected window.

The SFPutFile Call

The simplest way to save a file using the Standard File Tool Set is with the call `SFPutFile`. The `SFPutFile` routine, like the `SFGetFile` routine, displays a standard, predesigned dialog box. The IIGS operator can then use the dialog to save the selected file on a disk. With `SFPutFile`, like `SFGetFile`, the calling program can specify the location of the dialog box on the screen, the prompt that appears at the top of the box, and the types of files to be displayed in the box. But it does not permit an application program to modify the design of the box. Programs that use a custom-tailored dialog box must use another Standard File routine, `SFPPutFile`.

In the SF.S1 program, files are saved using the `SFPutFile` call. Listing 12–5 shows how the call is used in the program.

Listing 12–5
SFPutFile call in SF.S1

```
SaveIt          START
                Using WindowData
                Using IOData

                PushWord #20              ; upper X coordinate
                PushWord #20              ; upper Y coordinate
                PushLong #TopMsg
                PushLong #WinOTitle
                PushWord #15              ; max length of filename
                PushLong #ReplyRecord     ; defined in iodata
                _SFPutFile

                lda GoodFlag
                bne cont
                jmp return                ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda WinOHandle
                ldx WinOHandle+2
                jsr Deref
                sta PicDestOut
                stx PicDestOut+2
```

```
                jsr SaveOne

                PushLong NamePtr
                PushLong WinOPtr
                _SetWTitle              ; update window title

                lda WinOHandle
                ldx WinOHandle+2
                jsr Unlock

                PushLong NamePtr        ; update 'title' menu item
                PushWord #262           ; menu item number
                _SetMItemName           ; update name of item

                PushWord #0
                PushWord #0
                PushWord #3             ; menu number
                _CalcMenuSize           ; update width of items

return          rts

TopMsg          str 'Type name of picture:'

                END
```

SFPutFile, like SFGetFile, takes five parameters. There are some differences, however, between the parameter sequences used by the two calls. The parameters that must be passed to the SFPutFile call are

- A 2-byte integer that specifies the horizontal screen coordinate of the upper left corner of the dialog box.
- Another 2-byte integer that specifies the vertical screen coordinate of the upper left corner of the dialog box.
- A pointer to a Pascal-style string that is printed as a prompt inside the dialog box.
- A pointer to a Pascal-type string that can be used to specify a default file name. If a pointer is specified, the string that is pointed to is printed in a line edit item inside the default box. You can then save that file by clicking the mouse button inside an OK box or pressing the Return key. If you want to save another file, the default string can be erased or edited using standard line edit techniques. If a 0 is passed in this parameter, a default string is not printed on the screen.
- A pointer to the same kind of five-field reply record used by the SFGetFile call.

After the SFPutFile routine is called in the SF.S1 program, the LoadOne subroutine loads the file selected by the user into the program's window buffer. The name of the window is updated, and the menu is modified so that it displays the new window's name.

The SF.S1 Program

The sample program in this chapter, SF.S1, is an expanded version of the DIALOG.S1 program created in chapter 11. To convert DIALOG.S1 into SF.S1, the following modifications are necessary:

1. Edit the heading of the program so that it looks like the one shown in listing 12–6.
2. Following the program segment labeled EventLoop, insert the segments shown in listing 12–7. These segments are the heart of the SF.S1 program. They load and save files and control the Standard File Tool Set.
3. Replace the data segment labeled MenuData with the segment shown in listing 12–8.
4. At the end of the program, add the data segment shown in listing 12–9.
5. Make sure that the latest version of INITQUIT.S1 is on the same disk that holds your SF.S1 source code. The COPY directive at the end of the SF.S1 combines the SF.S1 program and the INITQUIT.S1 program.

Listing 12–6
SF.S1 heading segment

```
*
*  SF.S1
*

*** A FEW ASSEMBLER DIRECTIVES ***

                Title 'SF'

                ABSADDR on
                LIST off
                SYMBOL off
                65816 on
                mcopy SF.macros

                KEEP SF
```

Listing 12–7
SF.S1 new segments

```
*
*  LOADIT: ROUTINE TO LOAD A PICTURE FROM DISK
*

LoadIt          START
                using WindowData
                using IOData

                jsr Repaint

                PushWord #20              ; upper x coordinate
                PushWord #20              ; upper y coordinate
                PushLong #PromptPtr
                PushLong #0               ; no filter process
                PushLong #TypeListPtr     ; file types to display
                PushLong #ReplyRecord     ; defined in iodata
                _SFGetFile

                lda GoodFlag
                bne cont
                jmp return                ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda WinOHandle
                ldx WinOHandle+2
                jsr Deref
                sta PicDestIn
                stx PicDestIn+2
                jsr LoadOne

                PushLong NamePtr
                PushLong WinOPtr
                _SetWTitle               ; update window title

                lda WinOHandle
                ldx WinOHandle+2
                jsr Unlock

                PushLong NamePtr          ; update 'title' menu item
                PushWord #262             ; menu item number
                _SetMItemName             ; update name of item
```

```
                PushWord #0
                PushWord #0
                PushWord #3             ; menu number
                _CalcMenuSize           ; update width of items

return          rts

PromptPtr       str 'Load Picture:'

TypeListPtr     anop
NumEntries      dc i1'1'
Filetype1       dc h'c1'

                END

*
*  SAVEIT: ROUTINE TO SAVE A PICTURE TO DISK
*

SaveIt          START
                Using WindowData
                Using IOData

                PushWord #20            ; upper X coordinate
                PushWord #20            ; upper Y coordinate
                PushLong #TopMsg
                PushLong #WinOTitle
                PushWord #15            ; max length of file name
                PushLong #ReplyRecord   ; defined in iodata
                _SFPutFile

                lda GoodFlag
                bne cont
                jmp return              ; user canceled operation

cont            lda #FName
                sta NamePtr
                lda #^FName
                sta NamePtr+2

                lda WinOHandle
                ldx WinOHandle+2
                jsr Deref
                sta PicDestOut
                stx PicDestOut+2
```

```
                jsr SaveOne

                PushLong NamePtr
                PushLong WinOPtr
                _SetWTitle              ; update window title

                lda WinOHandle
                ldx WinOHandle+2
                jsr Unlock

                PushLong NamePtr        ; update 'title' menu item
                PushWord #262           ; menu item number
                _SetMItemName           ; update name of item

                PushWord #0
                PushWord #0
                PushWord #3             ; menu number
                _CalcMenuSize           ; update width of items

return          rts

TopMsg          str 'Type name of picture:'

                END

*
* LoadOne
* Loads the picture whose pathname is passed in NamePtr to address
* passed in PicDestIN
*

LoadOne         START
                using IOData

                _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open picture file.'

cont1           anop
                lda OpenID
                sta ReadID
                sta CloseID

                _Read ReadParams
                bcc cont2
                ErrorCheck 'Could not read picture file.'
```

```
cont2           anop
                _Close CloseParams

                clc
                rts
                END

*
* SaveOne
* Saves the picture whose pathname is passed in NamePtr from address
* passed in PicDestOUT
*

SaveOne         START
                using IOData

                lda NamePtr
                sta NameC
                sta NameD
                lda NamePtr+2
                sta NameC+2
                sta NameD+2

                _Destroy DestParams

                lda #$c1                ; SuperHiRes picture type
                sta CType
                lda #$0                 ; standard type = 0
                sta CAux

                _Create CreateParams
                bcc cont0
                ErrorCheck 'Could not create pic file.'

cont0           _Open OpenParams
                bcc cont1
                ErrorCheck 'Could not open pic file.'

cont1           anop
                lda OpenID
                sta WriteID
                sta CloseID

                _Write WriteParams
                bcc cont2
                ErrorCheck 'Could not write to pic file.'
```

```
cont2           anop
                _Close CloseParams

                clc
                rts

                END
```

Listing 12–8
SF.S1 new MenuData segment

```
*
* Menu Data
*
MenuData        DATA

Return          equ 13

Menu1           dc c'>L@\XN1',i1'RETURN'
                dc c' LA Window Program \N257',i1'RETURN'
                dc c'.'

Menu2           dc c'>L File \N2',i1'RETURN'
                dc c' LNew \N258V',i1'RETURN'
                dc c' LLoad \N259',i1'RETURN'
                dc c' LSave \N260V',i1'RETURN'
                dc c' LQuit\N261',i1'RETURN'
                dc c'.'

Menu3           dc c'>L Windows \N3',i1'RETURN'
                dc c' LUntitled \N262',i1'RETURN'
                dc c'.'

                END

MenuTable       DATA

*               Menu 1 (apple)
                dc i'ignore'            ; one for the NDAs
                dc i'ignore'            ; 'a window program'

*               Menu 2 (file)
                dc i'Repaint'           ; 'doWin0' (new window)
                dc i'LoadIt'
                dc i'SaveIt'
                dc i'doQuit'            ; quit item selected
```

```
*               Menu 3 (windows)
                dc i'doWin0'            ; 'untitled'

                END

***
```

Listing 12–9
SF.S1 IOData segment

```
*
* IOData
*

IOData          DATA

ReplyRecord     anop
GoodFlag        ds 2
FType           dc i'193'               ; $c1
AuxFType        dc i'0'                 ; #0
FName           ds 15
FullPathName    ds 128

CreateParams    anop
NameC     dc    i4'0'
          dc    i2'$00C3'               ; DRNWR
CType     dc    i2'$00C1'               ; super high-res graphics
CAux      dc    i4'$00000000'           ; Aux
          dc    i2'$0001'               ; type
          dc    i2'$0000'               ; create date
          dc    i2'$0000'               ; create time

DestParams      anop
NameD           dc   i4'0'

OpenParams      anop
OpenID          ds 2
NamePtr         ds 4
                ds 4

ReadParams      anop
ReadID          ds 2
PicDestIN       ds 4
                dc i4'$8000'            ; this many bytes
                ds 4                    ; how many xfered
```

```
WriteParams     anop
WriteID         ds 2
PicDestOUT      ds 4
                dc i4'$8000'            ; this many bytes
                ds 4                    ; how many xfered

CloseParams     anop
CloseID         ds 2

                END
```

The SF.C Program

Listing 12–10 is a C language version of the SF.S1 program. Designed to be used with the `include` file INITQUIT.C, it works almost exactly like the SF.S1 program.

In the C version of the SF program, files are not loaded and saved using ProDOS calls, as they are in the assembly language version. Instead, SF.C uses four C library routines: `Open`, `Close`, `Read`, and `Write`. These routines are called in the `LoadIt` and `SaveIt` segments of the program.

The `Open` function returns an integer, known as a file descriptor, for each file successfully opened. If the call fails, it returns −1. In the SF.C program, you test the value returned by `Open`. If the value is −1, a dialog window appears on the screen and tells the user an I/O error has occurred. Then the user can try to continue or quit. This dialog is created and displayed in the `BadIO` segment of the program.

The event loop of the program is the same as the one that appeared in the DIALOG.C program in chapter 11. The `DoMenus` section is expanded to accommodate some new menu choices, but the changes need little explanation.

There are also changes in the way window titles are selected and displayed. These modifications are necessary because window titles can change in the SF.S1 program. Although there may be a more elegant way to accommodate the shifting of window titles, calling `HideWindow` and then `ShowWindow` does the job.

Also, the File menu selection in SF.S1 does not conform strictly to the usual conventions for saving and loading files. For example, in the SF.S1 program, you can use the menu selections New, Load, or Quit without saving first—and you can thus wipe out the picture currently on the screen without warning. Because SF.S1 is a tutorial program, we decided to forego fixing that bug to avoid adding more complexity to the program.

One feature we did add was to disable the menu selection Save when no window is open. Disabling an item lets the user know "that can't be done right now," and ensures that TaskMaster does not return the constant that represents the disabled item in the `wmTaskData` field.

Listing 12–10
SF.C program

```
#include "initquit.c"
#include <prodos.h>
#include <string.h>
#include <fcntl.h>

Boolean done = false;
WmTaskRec   myEvent;

/************************************/
/* Data and routine to create menus */
/************************************/

/*  Set up menu strings. Because C uses \ as an escape character, we use
two when we want a \ as an ordinary character. The \ at the end of each
line tells C to ignore the carriage return. This lets us set up our items
in an easy-to-read vertical alignment. */

char *menu1 = "\
>L@\\XN1\r\
 LA Standard File Program \\N257\r\
.";

char *menu2 = "\
>L File \\N2\r\
 LNew \\N258V\r\
 LOpen #\\N259\r\
 LSave \\N260V\r\
 LQuit #\\N261\r\
.";

char *menu3 = "\
>L Windows \\N3\r\
 LUntitled \\N262\r\
.";

#define NEW_ITEM 258
#define OPEN_ITEM 259
#define SAVE_ITEM 260
#define QUIT_ITEM 261    /* these will help us check menu item numbers */
#define TITLE_ITEM 262

BuildMenu()
{
   InsertMenu(NewMenu(menu3),0);
   InsertMenu(NewMenu(menu2),0);
```

```
    InsertMenu(NewMenu(menu1),0);
    FixMenuBar();
    DrawMenuBar();
    DisableMItem(SAVE_ITEM);/* save is disabled until a window is drawn */
}

/*****************************************************************/
/* Data structures and routines to set up and refresh            */
/* offscreen drawing environment                                 */
/*****************************************************************/

LocInfo picOLocInfo = { mode320,
                        NULL, /* space for pointer to pixel image */
                        160,  /* width of image in bytes = 320 pixels */
                        0,0,200,320  /* frame rect */
                      };

Rect screenRect = {0,0,200,320};
GrafPort picOPort;

#define IMAGE_ATTR attrLocked+attrFixed+attrNoCross+attrNoSpec+attrPage

PicOSetup()  /* called once by MakeWindow at start of program */
{
GrafPortPtr thePortPtr;

    picOLocInfo.ptrToPixImage = *(NewHandle(0x8000L,myID,IMAGE_ATTR,NULL));
    thePortPtr = GetPort();
    OpenPort(&picOPort);
    SetPort(&picOPort);
    SetPortLoc(&picOLocInfo);
    ClipRect(&screenRect);
    EraseRect(&screenRect);
    SetPort(thePortPtr);
}

ErasePicO()
{
GrafPortPtr oldPortPtr;

    oldPortPtr = GetPort();
    SetPort(&picOPort);
    ClipRect(&screenRect);
    EraseRect(&screenRect);
    SetPort(oldPortPtr);
}
```

```
/*********************************************************************/
/* Data and routines for handling Open and Save calls                */
/*********************************************************************/

#define O_PICLOAD O_RDONLY+O_BINARY
#define O_PICSAVE O_WRONLY+O_CREAT+O_BINARY+O_TRUNC

SFReplyRec file = {0,193}; /* intit 2 fields, rest are 0'd  */
char curpath[130] ;  /* place for C string version of pathname */
Byte typelist[2] = {1,193}; /* we only want to open hi-res pictures */
FileRec fileInfo = {file.fullPathname}; /* initialize first field  */

LoadIt()
{
int filedes;
char oldTitle[16];

   strncpy(oldTitle,file.filename,16); /* save title in case load fails */

   SFGetFile(20,20,"\pLoad Picture:",NULL,typelist,&file);
   if(file.good) {
      p2cstr(strncpy(curpath,file.fullPathname,(int)*file.fullPathname+ 1 ) ) ;

      if((filedes = open(curpath,O_PICLOAD)) != -1) {
         read(filedes,picOLocInfo.ptrToPixImage,0x8000);
         close(filedes);

         SetMItemName(file.filename,262);
         CalcMenuSize(0,0,3);
         RenewWind();
      }
      else {
         BadIO();       /* load failed, put up message and restore title */
         strncpy(file.filename,oldTitle,16);
      }
   }
}

SaveIt(winPtr)
GrafPortPtr winPtr;
{
int filedes;
char oldTitle[16];

   strncpy(oldTitle,file.filename,16); /* save title in case save fails */

   SFPutFile(20,20,"\pType name of picture:",file.filename,15,&file);
```

```
   if(file.good){
      p2cstr(strncpy(curpath,file.fullPathname,(int)*file.fullPathname+
1));
      if((filedes = open(curpath,O_PICSAVE)) ! 1= -1){
         write(filedes,picOLocInfo.ptrToPixImage,0x8000);
         close(filedes);

         GET_FILE_INFO(&fileInfo);  /* make file's type a hires picture */
         fileInfo.fileType = 0xC1;
         SET_FILE_INFO(&fileInfo);

         SetMItemName(file.filename,TITLE_ITEM);
         CalcMenuSize(0,0,3);
      }
      else {        /* save failed, put up message and restore title */
         BadIO();
         strncpy(file.filename,oldTitle,16);
      }
   }
   else strncpy(file.filename,oldTitle,16);
}

/*****************************************************/
/* Data structures and routines to create window     */
/*****************************************************/

/* Initialize template for NewWindow  */

#define FRAME fQContent+fMove+fZoom+fGrow+fBScroll+fRScroll+fClose+fTitle

ParamList template = { sizeof(ParamList),
                       FRAME,
                       file.filename,  /* Pointer to title in SFReplyRec */
                       0L,             /* RefCon           */
                       26,0,188,308,   /* Full size (0=default)  */
                         NULL,         /* use default ColorTable */
                       0,0,            /* origin           */
                       200,320,        /* data area height & width */
                       200,320,        /* max cont height & width  */
                       2,2,    /* vertical & horizontal scroll increment */
                       20,32,  /* vertical & horizontal page increment   */
                       NULL,   /* no info bar text string  */
                       0,      /* info bar height = none   */
                       NULL,   /* default def proc         */
                       NULL,   /* no info bar draw routine */
                       NULL,   /* draw content must be filled in at run time */
                       26,0,188,308, /* starting content rect */
```

```
                        -1L,    /* topmost plane */
                        NULL    /* let window manager allocate record */
                            };

/* Window's draw content routine */

pascal void DrawContent()
{
   PPToPort(&picOLocInfo,&(picOLocInfo.boundsRect),0,0,modeCopy);
}

GrafPortPtr winOPtr;

MakeWindow() /* Set default title str, complete template, make the window */
{
   strncpy(file.filename,"\pUntitled",9); /* default name for new window  */
   template.wContDefProc = DrawContent;
   winOPtr = NewWindow(&template);
}

RenewWind()  /* a way to restore a window to its default size and position */
{       /* will not affect the contents unless ErasePicO is called first */

   EnableMItem(SAVE_ITEM);
   HideWindow(winOPtr);
   CloseWindow(winOPtr);
   winOPtr = NewWindow(&template);
   SelectWindow(winOPtr);
   ShowWindow(winOPtr);
}

/*****************************************************************/
/* Data and routines to set up and display dialogs               */
/*****************************************************************/

char prompt[40] = "\pUnable to load or save ";

ItemTemplate item1 = { 1,{8,129,22,179},buttonItem,"\pStart\r",0,0,NULL };
ItemTemplate item2 = { 2,{8,8,22,58},buttonItem,"\pQuit\r",0,0,NULL };
ItemTemplate item3 = { 3,{8,67,22,117},buttonItem,"\pHelp\r",0,0,NULL };
ItemTemplate item4 = { 4,{30,8,55,259},statText,prompt,0,0,NULL };
ItemTemplate item5 = { 1,{8,129,22,179},buttonItem,"\pOK",0,0,NULL };

DialogTemplate dtemp = {{84,63,114,252},true,0L,&item1,&item2,&item3,NULL};
DialogTemplate iotemp = {{84,23,144,292},true,0L,&item5,&item2,&item4,NULL};

DoDialog()   /* Create and display an opening dialog box */
{
```

```
GrafPortPtr dlgPtr;
Word hit;

  dlgPtr = GetNewModalDialog(&dtemp);

  while ((hit = ModalDialog(NULL)) == 3);
  done = (hit == 2);
  CloseDialog(dlgPtr);
}

BadIO()
{
GrafPortPtr dlgPtr;

  strncat(prompt,file.filename + 1, *file.filename);
  *prompt = 23 + *file.filename;
  dlgPtr = GetNewModalDialog(&iotemp);

  done = (ModalDialog(NULL) == 2);
  CloseDialog(dlgPtr);
}

/*********************************************************************/
/* Main routine. Set up environment, call eventloop, and shut down  */
/*********************************************************************/

main()
{
   StartTools();
   DoDialog();
   BuildMenu();
   MakeWindow();
   PicOSetup();
   EventLoop();
   DisposeHandle(FindHandle(picOLocInfo.ptrToPixImage));
   ShutDown();
}

/************************************/
/* Event loop and supporting routines */
/************************************/

EventLoop()
{
   myEvent.wmTaskMask = 0xOFFF;
   while(!done)
      switch ( TaskMaster(everyEvent,&myEvent)) {
         case wInMenuBar:
```

```
            DoMenus();
            break;
         case wInGoAway:
            DisableMItem(SAVE_ITEM);
            HideWindow(winOPtr);
            break;
         case wInContent:
            Sketch();
      }
}

DoMenus()
{
Word *data = (Word *)&myEvent.wmTaskData; /* address of item id  */

    switch(*data) {
      case QUIT_ITEM:
         done = true;
         break;
      case OPEN_ITEM:
         LoadIt();
         break;
      case SAVE_ITEM:
         SaveIt();
         HideWindow(winOPtr);  /* Make sure the title gets updated */
         ShowWindow(winOPtr);
         break;
      case NEW_ITEM:
         ErasePicO();
         strncpy(file.filename,"\pUntitled",9);
         RenewWind();
         break;
      case TITLE_ITEM:
         EnableMItem(SAVE_ITEM);
         SelectWindow(winOPtr);
         ShowWindow(winOPtr);
         break;
    }
    HiliteMenu(false,*(data + 1)); /* data + 1 is address of menu id  */
}

Sketch() /* sketch into current port, and into offscreen port */
{
Point mouseLoc;
GrafPortPtr thePortPtr = (GrafPortPtr)myEvent.wmTaskData;
Rect theRect;
```

```
    mouseLoc = myEvent.wmWhere;

    StartDrawing(thePortPtr);  /* set up correct drawing coordinate system */
    GetPortRect(&theRect);     /* copy current port rect */
    GlobalToLocal(&mouseLoc);  /* get cursor pos in local coordinates */

    MoveTo(mouseLoc);          /* set pen position to mouse loc */
    SetPort(&picOPort);        /* switch to offscreen port */
    ClipRect(&theRect);        /* clip offscreen drawing to window's port rect */
    MoveTo(mouseLoc);          /* set offscreen pen to same location */
    SetPort(thePortPtr);       /* switch back to window's port   */

    while (StillDown(0)) {
       GetMouse(&mouseLoc);    /* get new mouse coordinates */

       LineTo(mouseLoc);       /* draw line in both ports */
       SetPort(&picOPort);
       LineTo(mouseLoc);
       SetPort(thePortPtr);
    }
    SetOrigin(0,0);            /* restore normal coordinates */
}
```

CHAPTER 13

The Sound of Music

The IIGS as a Sound and Music Synthesizer

One of the most remarkable features of the IIGS is its ability to synthesize music and sounds. Some reviewers have declared that the IIGS offers the finest sound-synthesizing capabilities of any computer in its class. So it's no wonder that the *s* in IIGS stands for *sound.*

You don't have to be a musician or an audio engineer to understand how the synthesizer built into the IIGS works. To write sound and music programs for the Apple IIGS, however, it doesn't hurt to know a little bit about how a music synthesizer produces sound. So, in the first part of this chapter, you take a brief look at some important facts about the science of sound and how the IIGS produces sound and music. Then you type, assemble, and run a program that turns your IIGS keyboard into a music synthesizer capable of producing an almost limitless variety of sounds.

The Characteristics of Sound

When you hear a sound from a musical instrument, four characteristics are combined to create the sound you perceive. These four characteristics are

- Volume, or loudness
- Frequency, or pitch
- Timbre, or sound quality

- Dynamic range, or the difference in level between the loudest sound that can be heard and the softest sound that can be heard during a given period of time. This time period can range between the time it takes to play a single note and the length of a much longer listening experience, such as a musical performance or a complete musical recording.

Sound Hardware in the IIGS

To produce sounds that have these four characteristics—volume, frequency, timbre, and dynamic range—the IIGS is equipped with a pair of special-purpose sound chips. One is the *digital oscillator chip,* or DOC, and the other is the *general logic unit,* or GLU. Let's take a closer look at these two processors.

The Digital Oscillator Chip

The digital oscillator chip, or DOC, is a sound-generating microprocessor designed by the Ensoniq sound synthesizer company. DOCs are used in Ensoniq synthesizers as well as in the IIGS.

The basic sound-generating unit used by the DOC is a component called an *oscillator.* To produce a sound, an oscillator must step through a table of sound samples stored as digital numbers. This table must be supplied by the application program using the oscillator. It can be created while a program is running, or it can be stored on a disk and loaded into memory in advance.

The DOC contains thirty-two oscillators, but two are unavailable for use in application programs. One is always used as a clock, and another is reserved for future use. That leaves thirty oscillators, each of which can function independently. In practice, however, the DOC's oscillators are used in pairs because it takes at least two oscillators to produce a continuous instrumental voice.

When two oscillators are used together to produce a sound, they form a functional unit called a *generator.* So, in normal use, the DOC has fifteen generators and thus is a 15-voice chip.

The DOC also has a component called an *analog-to-digital converter,* or ADC. The ADC makes it possible for the DOC to record a digital sample of an actual sound, so that the sound can be played back later from its digital sample. More information about this capability is in the *Apple IIGS Hardware Reference.*

The General Logic Unit

The general logic unit, or GLU, is a chip that interfaces the DOC processor and the IIGS system. It also enables the IIGS to produce sound in the same way as older Apple IIs: by toggling a single-bit switch that can make a speaker vibrate at various rates of speed. But thanks to the GLU, this method of producing sound is improved; its volume can now be software controlled.

In addition to its DOC and GLU chips, the IIGS has 64K of dedicated RAM used only for storing sound samples. Because this area of memory is used only by the DOC, it is sometimes referred to as DOC RAM.

Sound Tools in the Toolbox

The IIGS Toolbox contains three tool kits that make it possible to write sound and music programs without accessing the sound registers used by the DOC and the GLU directly. These three tool sets are the

- Sound Tool Set, which starts and stops sounds, sets sound volumes, performs read and write operations to and from DOC registers, and reads and writes data to and from DOC RAM.
- Note Synthesizer, a higher-level tool set that produces and controls musical notes. The Note Synthesizer can emulate the sound of virtually any musical instrument and can produce unique musical sounds with almost any characteristics desired.
- Note Sequencer, a still higher-level tool set that makes it easier to combine various notes, chords, note patterns, and rhythms into musical performances and compositions.

The sample program in this chapter, MUSIC.S1, uses the Sound Tool Set and the Note Synthesizer. It does not use the Note Sequencer because it is an interactive program. The MUSIC.S1 program appears at the end of this chapter.

More About the Science of Sound

Now that you know something about how the IIGS produces music and sound, you're ready to take a closer look at the four primary characteristics of every sound: volume, frequency, timbre, and dynamic range.

Volume

If you've ever turned a volume knob on a radio, you know just about all you'll need to know about volume to write sound and music programs for the IIGS.

In programs written using the Sound Tool Set, the volume of a sound is controlled using the Sound Tool call `SetSoundVolume`. In programs that use the Note Synthesizer, volume is expressed as a value ranging from 0 to 127 and is controlled by passing a parameter to the Note Synthesizer call `NoteOn`.

As you shall see later, the `NoteOn` call must be made every time a note is produced by the Note Synthesizer. In the MUSIC.S1 program, volume is controlled using the `NoteOn` call. You'll see how this is done later in this chapter.

Frequency

The pitch of a musical note is determined by its frequency. In programs written using the IIGS Note Synthesizer, frequency is measured in semitones, or halftones. A semitone value ranges from 0 to 127, with 60 representing middle C.

The frequency of a note, like the note's volume, can be established by passing a parameter to the Note Synthesizer call `NoteOn`. An example is provided later in this chapter.

Timbre

Timbre, or note quality, is sometimes illustrated with the help of a waveform. There are four basic varieties of waves: sine wave, square wave (or pulse wave), triangle wave, and sawtooth wave. But these four types of waves can be combined with each other, and with irregular wave patterns, in endless varieties.

To understand how waveforms work, you need to know a little about musical harmonics. So here is a crash course in music theory.

With the help of an electronic instrument, you can generate a tone that has just one pure frequency. But when a note is played on a musical instrument, more than one frequency is usually produced. In addition to a primary frequency, or a fundamental, there is usually a set of secondary frequencies called harmonics. It is this total harmonic structure that determines the timbre of a sound.

When a tone containing only a fundamental frequency is viewed on an oscilloscope, the pattern produced on the screen is a pure sine wave. When a flute is played, the waveform produced is very close to that of a pure sine wave. The waveform of a sine wave is shown in figure 13–1.

When harmonics are added to a tone, the result is a richer sound that produces what is sometimes called a triangle wave. Triangle waveforms, or waves that are close to triangle waveforms, are produced by instruments such as xylophones, organs, and accordians. Figure 13–2 is a triangle wave.

When still more harmonics are added to a note, other kinds of waves are formed. Harpsichords and trumpets, for example, produce a type of wave sometimes called a sawtooth wave. A piano generates a squarish kind of wave called a square wave or a pulse wave. A sawtooth wave is illustrated in figure 13–3, and a pulse wave is shown in figure 13–4.

Another kind of waveform that the DOC can produce is a noise waveform. A noise waveform creates a random sound output that varies with a frequency proportionate to that of an oscillator built into Voice 1. Noise waveforms are often used to imitate the sound of explosions, drums, and other nonmusical noises.

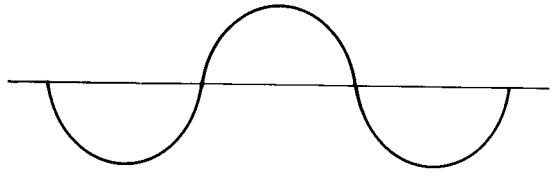


Figure 13–1
Sine waveform

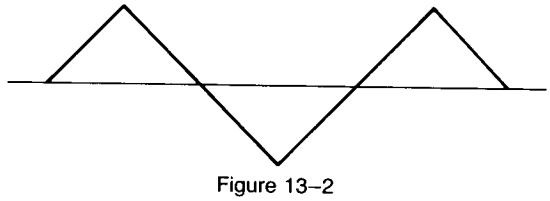


Figure 13–2
Triangle waveform

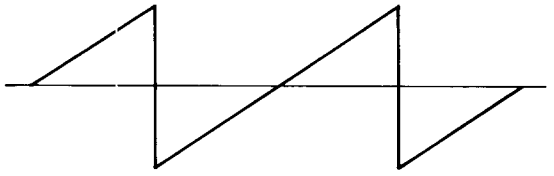


Figure 13–3
Sawtooth waveform

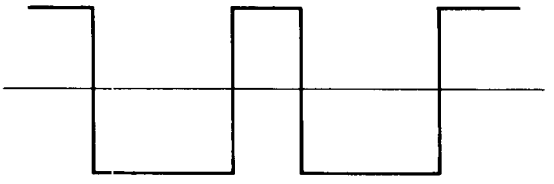


Figure 13–4
Pulse waveform

In programs written for the IIGS, waveforms can be created when needed—as they are in the MUSIC.S1 program—or they can be created and loaded into memory in advance. No matter how a waveform is created, though, it must be moved into DOC RAM before it can be used to produce a sound.

Dynamic Range

The dynamic range of a note—the difference in volume between its loudest sound level and its softest sound level—can be illustrated in many ways. To illustrate and control the dynamic ranges of notes, audio engineers sometimes use a device called an *ADSR envelope,* or attack-decay-sustain-release envelope. An ADSR envelope illustrates four distinct stages in the life of a note: four phases every note undergoes between the time it starts and the time it fades away. These four phases—attack, decay, sustain, and release—are shown in the ADSR envelope illustrated in figure 13–5.

A Close Look at an ADSR Envelope

As figure 13–5 shows, every note starts with an attack. The attack phase of a note is the length of time it takes for the volume of the note to rise from a level of zero to the note's peak volume.

As soon as a note reaches its peak volume, it begins to decay. The decay phase of a note is the length of time it takes for the note to decay from its peak volume to a predefined sustain volume.

When the decay phase of a note ends, the note is usually sustained for a certain period of time at a certain volume. Then a release phase begins. During this final phase, the volume of the note drops from its sustain level back down to zero.

When the IIGS Note Synthesizer is used in a program, the ADSR envelope of each sound in the program can be set up by creating a data structure called an instrument record. Then, when a note is played, the address of this record can be passed as a parameter to the Note Synthesizer call `NoteOn`.

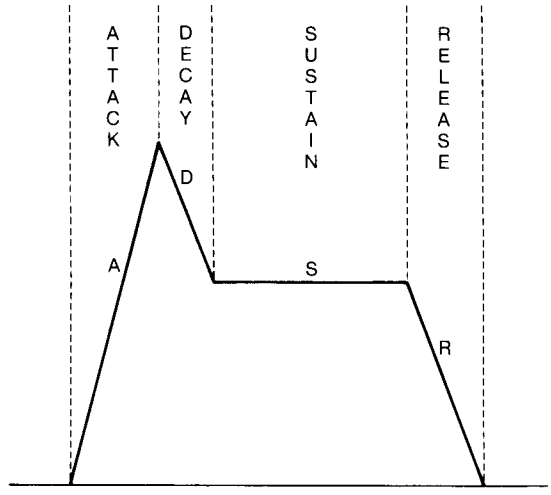


Figure 13–5
ADSR envelope

Initializing the Sound Tool Set and the Note Synthesizer

The Sound Tool Set and the Note Synthesizer, like most tools in the IIGS Toolbox, must be loaded and started before they can be used in a program. In programs that use both tool kits, the Sound Tool Set must be started first because the Note Synthesizer uses part of the Sound Tool Set's direct page.

In the MUSIC.S1 program, the Sound Tool Set is initialized in a program segment labeled `SoundStartUp`, and the Note Synthesizer is started in a segment labeled `NoteStartUp`.

`SoundStartUp`, the call that initializes the Sound Tool Set, is quite straightforward. It takes one parameter—a pointer to a direct page workspace—and returns with the carry clear if there is no error.

`NSStartup`, the call that initializes the Note Synthesizer, takes two parameters. The first parameter is a 2-byte update rate, which determines the rate at which sound envelopes are generated. Update rates are expressed in units of .4 cycles per second, or hertz. In the MUSIC.S1 program, the update rate passed to the `NSStartup` call is the decimal number 70, so the sound envelope used in the program is updated at a rate of 60 times a second, or 60 hertz.

The second parameter passed to the `NSStartup` call is a pointer to an interrupt-driven routine that can be used for note sequencing. No interrupts are used in the MUSIC.S1 program, so the value for this parameter is zero.

How the Note Synthesizer Works

When the Note Synthesizer is used in an application program, a sound generator must be allocated for each voice used in the program. The call to allocate a generator is `AllocGen`.

The `AllocGen` call takes two parameters: a 2-byte space to return a

result on the stack and a 1-word value to establish the priority of the generator being allocated.

This is how generator priorities work. Generator priorities can range from 0 to 128. When a generator has a priority of 0, it is free and thus can be allocated. If there are no free generators when a generator is to be allocated, the Note Synthesizer looks for the lowest-priority generator and "steals" it—if it has a priority of less than 128. If a generator has a priority of 128, it cannot be stolen.

When the `AllocGen` call returns, a generator number ranging from 0 to 13 is pushed onto the stack. Then, when a note is to be played by one of the DOC's fifteen generators, the generator can be referred to by its assigned number.

NoteOn Call

When all the generators needed by a program are allocated, the `NoteOn` call can be made each time a note is to begin, and the `NoteOff` call can be made each time a note is to end.

The `NoteOn` call takes four parameters:

- A 1-word generator number (the identification number assigned by the `AllocGen` call)
- A 1-word semitone number (a number ranging from 0 to 127, with the value 60 representing middle C)
- A 1-word volume parameter (a number ranging from 0 to 127)
- A 2-word pointer to an instrument record

The structure of an instrument record is described in the next section. The `NoteOn` call does not return any parameters.

The Structure of an Instrument Record

When the `NoteOn` call is used in a program, one of the parameters passed to it is an instrument record. The instrument record used in the MUSIC.S1 program is shown in table 13–1. The routine that plays notes using the instrument record is in the `PlayNote` segment of the program. The following paragraphs describe each of the fields shown in table 13–1.

The `Envelope` field of an instrument record is composed of up to eight linear segments. Each of these segments has a breakpoint value and an increment value, or slope. During each segment, the volume of the note being played ramps (increases or decreases) from its current value to its breakpoint value. The time that this process takes is determined by the increment value of the note's envelope.

The value of a breakpoint can range from 0 to 127. This range of values represents the level of a sound on a logarithmic scale, with each 16 steps changing the note's amplitude by 6 decibels (dB). The last breakpoint used in an envelope should have a value of 0.

Each increment value in the envelope field can range from 0 through 127. An increment is a value that is added to or subtracted from a note's current level at the update rate passed to the `NoteOn` call, thus changing its

Table 13–1
Instrument Record

Field Number	Field Name	Field Length
1	Envelope	24 bytes
2	ReleaseSegment	1 byte
3	PriorityIncrement	1 byte
4	PitchBlendRange	1 byte
5	VibratoDepth	1 byte
6	VibratoSpeed	1 byte
7	Spare	1 byte
8	AWaveCount	1 byte
9	BWaveCount	1 byte
10...	WaveLists	6 bytes each

frequency at a rate determined by its update rate. The sustain level of an envelope is created by setting an increment value to 0.

An increment is a 2-byte, fixed-point number, that is, a number that represents a fraction. Specifically, the fraction represented by an increment value is the value over 256. Thus, if an increment value is 1, it represents the fraction 1/256 and has to be added to a note's current volume 256 times—over a total elapsed time of 2.56 seconds—to cause the volume of the note to go up by 1.

The `ReleaseSegment` field of an instrument record is a number ranging from 0 to 7. This number determines how many segments it takes for the release of a note to go down to 0. When the release phase diminishes to 0, the note ends.

The `PriorityIncrement` field of an envelope is a number subtracted from the envelope's generator priority when the envelope reaches its sustain phase. Then, when the note reaches its release phase, its priority is cut in half. The priority of each allocated generator is also decremented by 1 each time a new generator is allocated. The purpose of this process is to ensure that the "oldest" active generators are "stolen" first when a new generator needs to be allocated.

The `PitchBlendRange` of an envelope is the number of semitones that a pitch is raised when its pitchwheel—a constantly incrementing value—reaches 127. The `PitchBlendRange` field controls a sound's vibrato effect. There are only three valid values for this field: 1, 2, and 4.

The `VibratoDepth` field defines the initial depth of a note's vibrato. Vibrato depth can range from 0 to 127, with a value of 0 meaning no vibrato will be used. The `VibratoSpeed` field, a value ranging from 0 to 255, controls the rate of vibrato oscillation. The next field, field 7, is reserved for future expansion.

Each of the digital oscillator chip's generators is made up of a pair of oscillators. Each oscillator in a pair can be used to synthesize as many different kinds of sound waves as desired. In an instrument record, field 8, `AWaveCount`, tells how many kinds of waves are defined for the first oscillator

in a pair. Field 9, `BWaveCount`, tells how many kinds of waves are defined for the second oscillator.

In an instrument record, a `WaveList` is a variable length array. Each element in a `WaveList` array has 6 bytes, divided into four fields. Fields 8 and 9 of an instrument record—the `AWaveCount` and `BWaveCount` fields—determine how many `WaveList` arrays the record contains.

The five fields in a `WaveList` array are:

- `TopKey` (1 byte). The highest semitone (ranging from 0 to 127) that a waveform will play. When a note is played by an instrument, the Note Synthesizer examines the `TopKey` field in each of the instrument's waveforms until it finds one that will play the requested note. Therefore, the waveforms listed in each wavelist should be arranged in an order of increasing `TopKey` values, and the last TopKey value in a wavelist should be 127.
- `WaveAddress` (1 byte). This field contains the high byte of the address of a waveform. This value is placed directly into a DOC register that holds a pointer to a waveform address. The waveform stored at the indicated address must be supplied by the program being executed.
- `WaveSize` (1 byte). This 1-byte field is placed directly in a DOC register that defines the size of the wave being accessed.
- `DOCMode` (1 byte). This field determines what mode the DOC uses to play the waveform listed. The most commonly used DOC mode is swap mode, in which two oscillators are used together to form a generator. DOC mode 0 is swap mode. More information on DOC modes are in the *Apple IIGS Hardware Reference*.
- `RelPitch` (2 bytes). This field is a 2-byte word that tunes the waveform in which it appears. The high byte of the word (the second byte of the field) is expressed in semitones, but can be a signed number. The low byte (the first byte of the field) is a value expressed in increments representing 1/256 of a semitone.

The MUSIC Program

Listing 13–1 is a complete listing of the MUSIC.S1 program. Listing 13–2, MUSIC.C, is a C language version of the program. INITQUIT.C, listing 13–3, is an `include` file that handles disk input and output for MUSIC.C. All three listings appear at the end of this chapter.

Type, assemble, and run the MUSIC program, and it will turn your IIGS keyboard into the keyboard of a real sound synthesizer. The keys on the Tab row are the synthesizer's white keys, and the keys on the numbers row are the black keys. The keyboard layout of the MUSIC synthesizer is illustrated in figure 13–6.

After you know how the IIGS produces sound, it isn't difficult to figure out how the MUSIC.S1 program works. It loads and starts up the Sound Tool

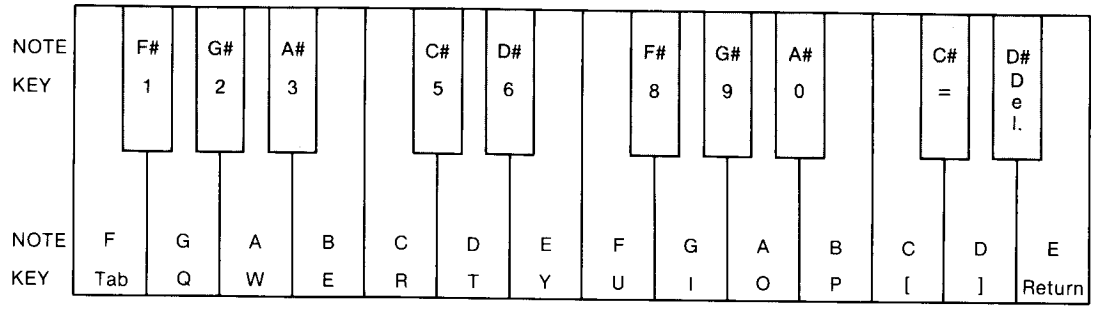


Figure 13–6
Key layout of the MUSIC synthesizer

Set and the Note Synthesizer, and then enters a loop that reads characters typed on the IIGS keyboard. In a segment labeled `GetKey`, the program constantly checks to see if the user has pressed a key on either of the top two rows of the keyboard. If such a key is pressed, the ASCII code of the typed character is converted into a musical semitone, and the program segment labeled `PlayNote` produces the appropriate musical sound. MUSIC.C is a fairly straightforward translation of the program into C.

Not the End

This brings us to the end of this book, but we have barely begun to explore the amazing capabilities of the Apple IIGS. If you have typed, assembled, and executed the Name Game program, and the programs designed to demonstrate the capabilities of the IIGS graphics and sound tools, you have all the supplies to hack your way into the IIGS jungle and see what lies beyond that first row of trees. So happy hunting!

MUSIC.S1, MUSIC.C, and INITQUIT.C Listings

Listing 13–1
MUSIC.S1 program

```
*
*  MUSIC.S1: Creating a Mini-Synthesizer
*

         keep music
         65816 on
         absaddr on
         mcopy music.macros
         longi on
         longa on
```

```
Music    START

         phk
         plb

         jsr SoundStartup

         jsr LoadSound

         jsr NoteStartup

         cli                              ; this seems to be necessary

         PrintLn ''
         PrintLn '   Your computer is now a mini-synthesizer.'
         PrintLn ''
         PrintLn '   The white keys are on the TAB row.'
         PrintLn '   The black keys are on the number row.'
         PrintLn ''
         PrintLn '   Keep shift lock down; press space bar to quit.'

loop     PushWord #0
         PushWord #0                      ; no echo
         _ReadChar                        ; read key the user typed
         pla
         and #$7F                         ; clear high bit
         cmp #$20                         ; space bar?
         beq exit

         jsr GetKey                       ; convert ASCII to a note
         bcs loop                         ; if carry set, no action

         jsr PlayNote                     ; call Note Synthesizer

         bra loop

exit     jsr Shutdown
         _Quit QuitParams

QuitParams anop
         dc i4'0'
         dc i2'0'

         END
```

```
***

GetKey          START

                short m,i
                ldx #23 ; 24 keys, starting from zero
loop            cmp Key,x  ; look for key in table
                beq foundit
                dex
                bpl loop
                jmp nonote ; search over--no note found

foundit         anop
                txa
                adc #$2A ; convert X reg content to a note
                clc      ; found note--clear carry
                jmp fini

nonote          sec      ; no note found--set carry
fini            long m,i
                rts

Key             dc h'09 31 51 32 57 33 45 52 35 54' ; ascii codes
                dc h'36 59 55 38 49 39 4f 30 50 5b'
                dc h'3d 5d 7f 0d'

                END

*
* Start up the tools we'll need
*

SoundStartup START

                _TLStartup
                PushWord #0
                _MMStartup
                ErrorCheck 'Could not call Memory Manager'

                pla
                sta MyID
                _MTStartup
                ErrorCheck 'Could not call Misc Tools'

                PushLong #ToolTable
                _LoadTools
                ErrorCheck 'Could not load sound tools'
```

```
*** GET SOME DIRECT PAGE SPACE AND START UP SOUND TOOLS ***

              PushLong #0                       ; room for handle
              PushLong #$100                    ; one page
              PushWord  MyID
              PushWord #$C001                   ; type: locked, fixed
              PushLong #0
              _NewHandle
              ErrorCheck 'Not enough memory!'
              pla
              sta   0                           ; using addresses $0000
              pla
              sta   2                           ; and $0002
              lda   [0]                         ; on direct page
              pha
              _SoundStartup
              ErrorCheck 'Could not start up sound tool'
              rts

MyID          ds 2
ToolTable dc i'1,25,0'                          ; one tool: #25, version 0
X
               end

*
*  Load Sound
*

LoadSound     START

*
* This routine creates a square wave, which approximates the
* waveform created by a piano.
*

              ldx #0
              lda #$40

              SetMode8                          ; use 8-bit accumulator

topedge       sta WaveForm,x                    ; draw top edge of wave
              inx
              cpx #128
              bne topedge
```

```
lowedge     anop                        ; draw low edge of wave
            lda #$C0
            sta WaveForm,x
            inx
            cpx #256
            bne lowedge

            SetMode16                   ; restore 16-bit accumulator

*
* Now we'll move the wave over to the DOC, using the sound tools.
*

            PushLong #WaveForm          ; arg1: src ptr
            PushWord #0                 ; doc start address
            PushWord #$100              ; byte count
            _WriteRamBlock
            ErrorCheck 'writing wave'
            rts

WaveForm    ds 256

            END

*
*  NoteStartup
*

NoteStartup START
            PushWord #70                ; 60 Hz updates
            PushLong #0                 ; no IRQ routine for me
            _NSStartup
            ErrorCheck 'Could not start up note synthesizer'
            rts

            END

*
* Now we play the note
*

PlayNote    START
            using NoteData
            sta   SemiTone

            PushWord #0                 ; space for result
            PushWord #64                ; medium priority
            _AllocGen
```

```
            ErrorCheck 'Could not allocate generator'
            pla
            sta GenNum

            PushWord GenNum
            PushWord SemiTone
            PushWord #112                ; medium volume
            PushLong #Piano              ; ptr to piano definition
            _NoteOn
            ErrorCheck 'Problem with NoteOn call'

*
* Normally, we would wait a while before issuing a note off. But
* because a piano has a fast attack and a long release, that
* isn't necessary in this case.
*

            PushWord GenNum
            PushWord SemiTone
            _NoteOff
            ErrorCheck 'Problem with NoteOff calloscillators
            rts

SemiTone    ds 2
GenNum      ds 2

            END

***

NoteData    DATA

Piano       dc i1'127,0,127'         ; env: sharp attack
            dc i1'112,20,1'          ; come down more slowly
            dc i1'0,48,0'            ; slow decay to 0
            dc i1'0,20,5'            ; and release in 112 steps
            dc i1'0,0,0'
            dc i1'0,0,0'
            dc i1'0,0,0'
            dc i1'0,0,0'             ; fill out 8 stages with 0's
            dc i1'3'                 ; release segment
            dc i1'32'                ; priority inc
            dc i1'2,0,0,0,1,1'       ; pbrange,vibdep,vibf,spare3
```

```
*
* Multi-sampled piano waveforms.
* First oscillator does the attack; second does loop.
*

AWavelist dc i1'127,0,0,0,0,12'              ; topkey,addr,size,ctrl,pitch

BWavelist dc i1'127,0,0,0,0,12'

              END

*
*  Routine that shuts down tools.
*

Shutdown START

              _NSShutdown
              ErrorCheck 'Problem with Note Synthesizer shutdown'
              _SoundShutdown
              rts

              END
```

Listing 13–2
MUSIC.C program

```
#include "initquit.c"

#define space ' '

EventRecord myEvent;
Word waveForm[257];
Instrument piano = {127,0x7F00, /* envelope */
                    112,0x0114,
                    0,0x0030,
                    0,0x0514,
                    0,0x0000,
                    0,0x0000,
                    0,0x0000,
                    0,0x0000,    /* end envelope */
                    3,32,
                    2,0,0,0,1,1,
                    127,0,0,0,0x0C00,    /* aWaveForm */
                    127,0,0,0,0x0C00     /* bWaveForm */
                   };

/* Keys contain the letters in "piano keyboard "#order. The backslashes */
/* are followed by the octal ASCII values of Tab, Delete, and Return.  */
```

```
char keys[] = "\0111Q2W3ER5T6YU8I90OP[=]\177\015";

main()
{
  StartTools();
  Prompt();
  LoadSound();
  asm {          /*it is necessary to clear interrupts */
      cli;
      }
  NSStartUp(70,nil);
 err("\pUnable to start up Note Synthesizer.\r\r");

  EventLoop ();
  NSShutDown();
  Shutdown();
}

LoadSound() /* it is more efficient to store words instead of bytes */
{
int i;

  for (i=0;i<64;i++)
    waveForm[i] = 0x4040;
  for (i=64;i<128;i++)
    waveForm[i] = 0xC0C0;

  WriteRamBlock (waveForm,0,0x100);
    err("Error in WriteRamBlock");
}

Prompt()
{
  GrafOff();
  printf("Your computer is now a mini-synthesizer\n\n");
  printf("The white keys are on the tab row.\n");
  printf("The black keys are on the number row.\n\n");
  printf("Keep shift lock down: Press space bar to quit.\n");
}

#define KEYMASK keyDownMask+autoKeyMask

EventLoop()
{
Boolean done = false;
Word i;
char theKey;
```

```
  while (!done)
    if(GetNextEvent(KEYMASK,&myEvent)) {
      theKey = (char)(myEvent.message);
      if (theKey == space)
        done = true;
      else
        if((i=findChar(keys,theKey)) < 24)
          PlayNote(i + 0x2A);
    }
}

int findChar(str,c) /* position of c in str, strlen if not present */
char *str;
char c;
{
int i=0;
  while(*str != c) {
    if(*(str++))
      i++;
    else
      break;
  }
  return i;
}

PlayNote(semiTone)
Word semiTone;
{
Word genNum;
  NoteOn((genNum = AllocGen(0,64)),semiTone,112,&piano);
  NoteOff(genNum,semiTone);
}
```

Listing 13–3
INITQUIT.C program

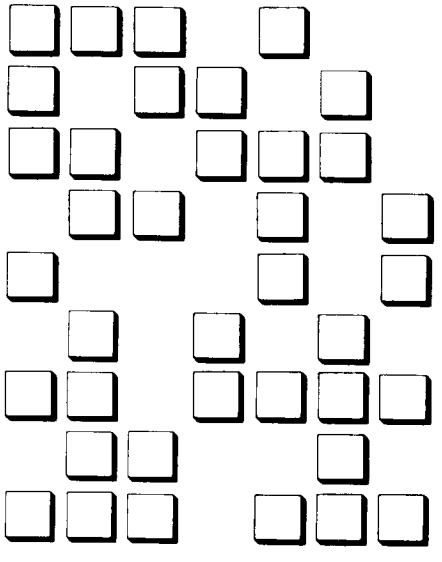
```
#include <TYPES.H>
#include <PRODOS.H>
#include <LOCATOR.H>
#include <MEMORY.H>
#include <MISCTOOL.H>
#include <QUICKDRAW.H>
#include <EVENT.H>
#include <SOUND.H>
#include <NOTESYN.H>
```

```
#define MODE mode320 /* 640 graphics mode def. from quickdraw.h */
#define MaxX 320      /* max X for cursor (for Event Mgr) */
#define dpAttr attrLocked+attrFixed+attrBank /* for
allocating direct page space */

#define err(str) if(_toolErr) SysFailMgr(_toolErr,str)

int  myID;            /* for Memory Manager. */
Handle zp;            /* handle for page 0 space for tools */
QuitRec qParms = {NULL,0};

int toolTable[] = {4,
                   4, 0x0100,  /* QD      */
                   6, 0x0100,  /* Event   */
                   8, 0x0100,  /* Sound   */
                  25, 0x0000   /* NoteSyn */
                  };

StartTools()        /* start up these tools: */
{
   TLStartUp();             /* Tool Locator */
   LoadEmUp();   /* load tools from disk */
   myID = MMStartUp();      /* Mem Manager  */
      err("\pUnable to start up Memory Mgr.\r\r");
   MTStartUp();             /* Misc Tools   */
      err("\pUnable to start up Misc. Tools\r\r");
   ToolInit();  /*start up the rest */
}

LoadEmUp()  /*Load tools, prompt for boot disk if not present */
{
Word response;
Pointer volName;

   GET_BOOT_VOL(&volName);

   LoadTools(toolTable);
   while(_toolErr == volumeNotFound) {
      response = TLMountVolume
(0,195,30,"\pPlease insert the disk",volName,"\pOK","\pCancel");
      if(response == 1)
         LoadTools(toolTable);
     else {
         TLShutDown();
         QUIT(&qParms);     /* try to exit gracefully */
      }
   }
```

```
    err("\pUnable to load tools\r\r");
}

ToolInit()      /* init the rest of needed tools */
{
    zp = NewHandle(0x500L,myID,dpAttr,0L);   /* reserve 6 pages */
      err("\pUnable to allocate DP space\r\r");

    QDStartUp((int) *zp, MODE, 160, myID);    /* uses 3 pages */
       err("\pUnable to start up QuickDraw.\r\r");
    EMStartUp((int) (*zp + 0x300), 20, 0, MaxX, 0, 200, myID);
       err("\pUnable to start up Event Mgr.\r\r");
    SoundStartUp((int) (*zp + 0x400));
       err("\pUnable to start up Sound Mgr.\r\r");
}

ShutDown()           /* shut down all of the tools we started */
{
    GrafOff();
    SoundShutDown();
    EMShutDown();
    QDShutDown();
    MTShutDown();
    DisposeHandle(zp); /* release our page 0 space */
    MMShutDown(myID);
    TLShutDown();
}
```

PART 3

Appendix



APPENDIX A

The 65C816 Instruction Set

This section is a complete listing of the 65C816 instruction set. It does not include pseudo-operations (also known as pseudo-ops, or directives), which vary from assembler to assembler. Tables A-1, A-2, and A-3 list the abbreviations used in this appendix.

Table A-1
Processor Status (P) Register Flags

Abbreviation	Flag
n	Negative (sign)
v	Overflow
b	Break
d	Decimal
i	Interrupt
z	Zero
c	Carry
m	Memory/accumulator select
e	Emulation

Table A-2
65C816 Registers

Abbreviation	Register
A	Accumulator or 8-bit accumulator
B	B register (high-order byte of 16-bit accumulator)
C	16-bit accumulator
X	X register
Y	Y register
P	Program counter
S	Stack pointer
M	Memory register
D	Direct page register
DBR (or B)	Data bank register
PBR (or K)	Program bank register

Table A-3
Addressing Modes

Abbreviation	Mode
#	Immediate
(a)	Absolute indirect
(a,x)	Absolute indexed indirect
(d)	Direct indirect
(d),y	Direct indirect indexed
(d,x)	Direct indexed indirect
(r,s),y	Stack relative indirect indexed
a	Absolute
a,x	Absolute indexed with X
a,y	Absolute indexed with Y
Acc	Accumulator
al	Absolute long
al,x	Absolute indexed long
d	Direct
d,x	Direct indexed with X
d,y	Direct indexed with Y
i	Implied
r	Program counter relative
r,s	Stack relative
rl	Program counter relative long
s	Stack
xya	Block move
[d]	Direct indirect long
[d],y	Direct indirect indexed long

adc — add with carry — 6502, 65C02, 65C816

Adds the contents of the accumulator to the contents of the effective address specified by the operand. If the P register's carry flag is set, a carry is also added to the result. The sum is stored in the accumulator.

If the accumulator is in 8-bit mode when the `adc` instruction is issued, two 8-bit numbers will be added, and the result of the operation is also 8 bits long. If the operation results in a carry, the carry flag is set.

If the accumulator is in 16-bit mode when the instruction is issued, two 16-bit numbers are added, and the result of the operation is also 16 bits long. If this operation results in a carry, the P register's carry flag is set.

The 65C816 has no instruction for adding without a carry. The `adc` instruction is the only addition instruction available. The carry flag can be cleared, however, with a `clc` instruction prior to an addition operation, and then no carry is added to the result.

It is considered good programming practice to issue a `clc` instruction before beginning any addition sequence. Then a carry bit will not be added to the result by mistake. If the first operation in an addition sequence results in a carry, the carry is added to the next higher-order operation, and each intermediate result correctly reflects the carry from the previous operation.

If the decimal flag is set when an `adc` instruction is issued, the addition operation is carried out in binary coded decimal (BCD) format. If the decimal flag is clear, binary addition is performed.

In emulation mode, `adc` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, v, z, c

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`adc (d)`	2	72
`adc (d),y`	2	71
`adc (d,x)`	2	61
`adc (r,s),y`	2	73
`adc d`	2	65
`adc d,x`	2	75
`adc r,s`	2	63
`adc [d]`	2	67
`adc [d],y`	2	77
`adc #`	2 (3)	69
`adc a`	3	6D
`adc a,x`	3	7D

adc a,y	3	79
adc al	4	6F
adc al,x	4	7F

and — logical AND — 6502, 65C02, 65C816

Performs a binary logical AND operation on the contents of the accumulator and the contents of the effective address specified by the operand. See figure A-1. Each bit in the accumulator is ANDed with the corresponding bit in the operand. The result of the operation is stored in the accumulator.

```
  0       0       1       1
AND 0   AND 1   AND 0   AND 1
  0       0       0       1
```

Figure A-1
Truth table for AND

The and instruction is often used as a mask, to clear specified bits in a memory location. When used as a mask, the instruction compares each bit in a memory location with the corresponding bit in the accumulator. Each bit cleared in the memory location clears the corresponding bit in the accumulator. Bits set in the memory location have no effect on their corresponding bits in the accumulator. For example, the sequence

```
lda #$00FF
and MEMLOC
sta MEMLOC
```

clears the high-order byte in MEMLOC, while leaving the low-order byte unchanged.

The and instruction conditions the P register's n and z flags. The n flag is set if the most significant bit of the result of the AND operation is set; otherwise, it is cleared. The z flag is set if the result is 0; otherwise, it is cleared.

In emulation mode, and is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z
Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
and (d)	2	32
and (d),y	2	31
and (d,x)	2	21
and (r,s),y	2	33
and d	2	25
and d,x	2	35

and r,s	2	23
and [d]	2	27
and [d],y	2	37
and #	2 (3)	29
and a	3	2D
and a,x	3	3D
and a,y	3	39
and al	4	2F
and al,x	4	3F

asl — arithmetic shift left — 6502, 65C02, 65C816

Shifts each bit in the accumulator or the effective address specified by the operand one position to the left. See figure A-2. A 0 is deposited into the bit 0 position, and the leftmost bit of the operand is forced into the carry bit of the P register. The result of the operation is left in the accumulator or the affected memory register. The `asl` instruction is often used in assembly language programs as an easy method for dividing by 2.

In emulation mode, `asl` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z, c

Registers affected: A, P, M

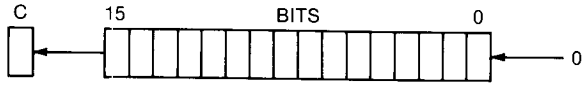


Figure A-2
ASL operation

Addressing Mode	Bytes	Opcode (hex)
asl Acc	1	0A
asl d	2	06
asl d,x	2	16
asl a	3	0E
asl a,x	3	1E

bcc — branch if carry clear — 6502, 65C02, 65C816

(Alias: `blt`.) Tests the P register's carry flag. Executes a branch if the carry flag is clear. Results in no operation if the carry flag is set.

The destination of the branch must be within a range of −128 to +127 memory addresses from the instruction immediately following the `bcc` instruction.

The bcc instruction is used for three main purposes:

- To test the carry flag after an arithmetic operation
- To test a bit that has been moved into the carry flag using a rotate, shift, or transfer operation
- To make a programming decision based on a comparison of two values

When bcc tests the result of a comparison operation, it comes after a comparison instruction (cmp, cpx, or cpy). When two values are compared with a comparison instruction, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The carry flag can then be tested using bcc. If the value in the accumulator is less than the value of the operand, the carry is clear and a branch is taken.

If bcc results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Because the meaning of bcc is not intuitively clear when the instruction 9is used as the result of a branch-after-compare operation, the APW assembler also accepts an alias: blt, which stands for *branch on less than* and assembles into the same machine language opcode as bcc.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	90

bcs — branch if carry set — 6502, 65C02, 65C816

(Alias: bge.) Tests the P register's carry flag. Executes a branch if the carry flag is set. Results in no operation if the carry flag is clear.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the bcs instruction.

The bcs instruction is used for three main purposes:

- To test the carry flag after an arithmetic operation
- To test a bit that has been moved into the carry flag using a rotate, shift, or transfer operation
- To make a programming decision based on a comparison of two values

When bcs tests the result of a comparison operation, it comes after a comparison instruction (cmp, cpx, or cpy). When two values are compared using a comparison instruction, data in memory is subtracted from data in

the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The carry flag can then be tested using `bcs`. If the value in the accumulator is greater than or equal to the value of the operand, the carry is set and a branch is taken.

If `bcs` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Because the meaning of `bcs` is not intuitively apparent when the instruction is used as the result of a branch-after-compare operation, the APW assembler also accepts an alias: `bge`, which stands for *branch on greater than or equal to* and assembles into the same machine language opcode as `bcs`.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	B0

beq — branch if equal — 6502, 65C02, 65C816

Tests the P register's zero flag. Executes a branch if the zero flag is set—that is, if the result of the last operation which affected the zero flag was 0. Results in no operation if the zero flag is clear.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `beq` instruction.

The `beq` instruction is used for several purposes:

- To test whether a value that has been pulled, shifted, incremented, or decremented is equal to 0
- To test the value of an index register to determine whether a loop has been completed
- To make a programming decision based on a comparison of two values

When `beq` tests the result of a comparison operation, it comes after a comparison instruction (`cmp`, `cpx`, or `cpy`). When two values are compared using a comparison instruction, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The zero flag can then be tested using `beq`. If the value in the accumulator is equal to the value of the operand, the zero flag is set and a branch is made.

If `beq` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	F0

bge — branch if carry set

`bge` is not a 65C816 instruction, but an alias recognized by the APW assembler. When assembled, it generates the same machine language opcode as the assembly language instruction `bcs`. For further details, see `bcs`.

bit — test memory bits against accumulator — 6502, 65C02, 65C816

Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory location. The contents of the accumulator are not affected, but three flags in the P register are affected.

If any bits in the accumulator and the value being tested match, the z flag is cleared. If no match is found, the z flag is set. Thus, a `bit` instruction followed by a `bne` instruction can determine if there is a bit match between the accumulator and the value of the operand. Similarly, a `bit` instruction followed by a `beq` instruction detects a no-match condition.

Another result of the `bit` instruction, in all of its addressing modes except immediate, is that bits 6 and 7 of the value in memory being tested are transferred directly into the v and n flags of the P register. This feature of the `bit` instruction is often used in signed binary arithmetic. If a `bit` operation results in the setting of the n flag, the value tested is negative. If the operation results in the setting of the v flag, that indicates an overflow condition when signed numbers are used.

In the immediate addressing mode, the only P register flag affected by the `bit` instruction is the z flag.

Flags affected in all modes except immediate addressing mode: n, v, z
Flags affected in immediate addressing mode: z
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
`bit d`	2	24
`bit d,x`	2	34

`bit #`	2 (30)	89
`bit a`	3	2C
`bit a,x`	3	3C

blt — branch if less than

`blt` is not a 65C816 instruction, but an alias recognized by the APW assembler. When assembled, it generates the same opcode as the assembly language instruction `bcc`. For further details, see `bcc`.

bmi — branch on minus — 6502, 65C02, 65C816

Tests the P register's n flag. Executes a branch if the n flag is set. Results in no operation if the n flag is clear.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bmi` instruction.

In operations involving two's complement arithmetic, `bmi` is often used to determine whether a value is negative. In logical operations, it is used to determine if the high bit of a value is set. It is sometimes used to detect whether short loops have counted down past 0.

If `bmi` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	30

bne — branch if not equal — 6502, 65C02, 65C816

Tests the P register's zero flag. Executes a branch if the zero flag is clear (if the result of the last operation which affected the zero flag was not zero). Results in no operation if the zero flag is set.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bne` instruction.

The `bne` instruction is used for several purposes:

- To test whether a value that has been pulled, shifted, incremented, or decremented is equal to zero

- To test the value of an index register to determine whether a loop has been completed
- To make a programming decision based on a comparison of two values

When `bne` tests the result of a comparison operation, it is used after a comparison instruction (`cmp`, `cpx`, or `cpy`). When two values are compared using a comparison instructions, data in memory is subtracted from data in the accumulator. This does not affect the value of the accumulator, but it conditions the carry flag as a result of the comparison. The zero flag can then be tested using `bne`. If the value in the accumulator is not equal to the value of the operand, the zero flag is set and a branch is made.

If `bne` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	D0

bpl — branch on plus — 6502, 65C02, 65C816

Tests the P register's n flag. Executes a branch if the n flag is clear. Results in no operation if the n flag is set.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bmi` instruction.

In operations involving two's complement arithmetic, `bpl` is often used to determine whether a value is negative. In logical operations, it is used to determine if the high bit of a value is clear.

If `bpl` results in a branch, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. When the address of the branch is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	10

bra — branch always — 65C02, 65C816

The `bra` instruction always results in a branch; no testing is done. There are three major differences between `bra` and the unconditional jump instruction `jmp`.

Because signed displacements are used, a statement that uses the `bra` instruction is only 2 bytes long, compared with the 3-byte length of a statement containing a `jmp` instruction. Second, the `bra` instruction uses displacements from the program counter and is thus relocatable. Last, the destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bra` instruction.

When the branch instruction is used, a 1-byte signed displacement, fetched from the second byte of the instruction, is sign-extended to 16 bits and added to the program counter. After the branch address is calculated, the result is loaded into the program counter, transferring control to that location.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	80

brk — break, or software interrupt — 6502, 65C02, 65C816

Forces a software interrupt, usually passing control of the Apple IIGS to the monitor. In programs written for the 65C816, a `brk` instruction can be handled in two ways, depending on whether the processor is in native mode or emulation mode.

If the 65C816 is in native mode, the program bank register is pushed onto the stack. Next, the program counter is incremented by 2 and pushed onto the stack. This incrementation takes place so that a break instruction can be followed by a signature byte identifying which break in a program caused the program to halt.

After the program counter is incremented by 2 and placed on the stack, the program bank register is cleared to 0, and the program counter is loaded from a special `brk` vector situated at $00FFE6 and $00FFE7. (This vector exists only in native mode, not in emulation mode, and that is why there is no need for the P register to have a break flag when the 65C816 is configured for emulation mode. In emulation mode, a `brk` instruction sends a program to vector $00FE6-$00FE7 instead of setting a special flag.) After the break is executed, the P register's decimal flag is cleared to 0.

If the 65C816 is in emulation mode when a `brk` instruction is given, the program counter is incremented by 2 and then pushed onto the stack, just as in native mode. Next, the processor status register, with the b (break) flag set, is pushed onto the stack. The interrupt disable flag is then set, and the program counter is loaded from an interrupt vector at $FFFE and $FFFF.

This is a different interrupt vector from the one the `brk` instruction uses when the 65C816 is in native mode. In native mode, the `brk` instruction does not have its own interrupt vector, as it does in emulation mode, but shares

one with hardware interrupts (IRQs). This shared vector is at memory addresses $FFFE and $FFFF. So, after an interrupt occurs, the interrupt handling routine at $FFFE-$FFFF must pull the processor status register off the stack and check the b flag to determine whether program processing was halted by a software interrupt (`brk`) instruction or a hardware interrupt.

If the break was caused by a software interrupt, the flag is set. But hardware IRQs push the P register onto the stack with the b flag clear. So, if the b flag is not set, the program was halted by a hardware IRQ.

When the 65C816 is in native mode, the P register's decimal flag is not modified by the `brk` instruction.

Flags affected in native mode: b and i

Flags affected in emulation mode: b, d, and i

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	2†	00

†`brk` is a 1-byte instruction, but increments the program counter by 2 before pushing it onto the stack.

brl — branch always long — 65C816

The `brl` instruction, like the `bra` instruction, always causes a branch. But `brl` is a 3-byte instruction. The 2 bytes immediately following the opcode form a 16-bit signed displacement from the program counter. Thus, the destination of a `brl` instruction can be anywhere within the current 64K program bank.

After the destination address of the branch is calculated, the result is loaded into the program counter, transferring control to that address.

There are two major differences between the `brl` instruction and the jump instruction `jmp`. The `brl` instruction (like any other branch instruction) is relocatable, but the `jmp` instruction is not. Also, `jmp` executes one cycle faster than `brl`.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	3	82

bvc — branch if overflow clear — 6502, 65C02, 65C816

Tests the overflow (v) flag in the 65C816 P register. Executes a branch if the overflow flag is clear. Results in no operation if the overflow flag is set. This instruction is used primarily in operations involving signed numbers.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bvc` instruction.

The most common use for `bvc` is to detect whether there is an overflow from bit 6 to bit 7 in a calculation involving signed numbers. The instruction can also test bit 6 in a value that has been moved into the v flag by the `bit` instruction.

The v flag can be altered by the instructions `adc`, `sbc`, `clv`, `bit` (in all but immediate mode), `sep`, and `rep`. It is also one of the flags restored from the stack by the `plp` and `rtl` instructions.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	50

bvs — branch if overflow set — 6502, 65C02, 65C816

Tests the overflow (v) flag in the 65C816 P register. Executes a branch if the overflow flag is set. Results in no operation if the overflow flag is clear. This instruction is used primarily in operations involving signed numbers.

The destination of the branch must be within a range of −128 to +127 memory addresses from the address immediately following the `bvs` instruction.

The most common use for `bvs` is to detect if there is an overflow from bit 6 to bit 7 in a calculation involving signed numbers. The instruction can also test bit 6 in a value that is moved into the v flag by the `bit` instruction.

The v flag can be altered by the instructions `adc`, `sbc`, `clv`, `bit` (in all but immediate mode), `sep`, and `rep`. It is also one of the flags restored from the stack by the `plp` and `rti` instructions.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
r	2	70

clc — clear carry — 6502, 65C02, 65C816

Clears the carry bit of the processor status register. The `clc` instruction should be used prior to addition (`adc`) operations to make sure that the carry flag is clear before addition begins. It should also be used prior to the `xce` (exchange carry flag with emulation bit) instruction when the intent of the instruction is to put the 65C816 into native mode.

Flags affected: c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	18

cld clear decimal mode

Puts the computer into binary mode (its default mode) so that binary operations (the kind most often used) can be carried out properly. When the decimal flag is set, `adc` and `sbc` calculations are carried out in binary coded decimal (BCD) mode.

It is a good practice to clear the decimal flag before beginning arithmetic operations that should be carried out in binary mode, in case the flag has been left in decimal mode following some previous decimal mode operation.

Flags affected: d

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	D8

cli clear interrupt disable flag — 6502, 65C02, 65C816

Enables hardware interrupts (IRQs) by clearing the P register's interrupt disable (i) flag. (If the i flag is set, hardware interrupts are ignored.) When the 65C816 starts servicing an interrupt, it finishes the instruction currently executing and then pushes the program counter and the P register on the stack. It then sets the i flag and jumps to one of ten interrupt vectors on page $FF of bank 0. The routine that it finds there must determine the nature of the interrupt and handle it accordingly.

When the interrupt service routine ends with `rti`, the `rti` instruction pulls the P register off the stack and returns to the instruction following the one that was executed just before the interrupt began. The restored P register contains a cleared i flag, so `cli` is ordinarily not necessary. However, if the interrupt service routine is designed to service interrupts that occur while a previous interrupt is still being handled, other interrupt handling routines must be reenabled with a `cli` instruction.

The `cli` instruction is also used to reenable interrupts if they have been disabled to allow the execution of time critical code or other code that cannot be interrupted.

Flags affected: i

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	58

clv — clear overflow flag — 6502, 65C02, 65C816

Clears the P register's overflow (v) flag by setting it to 0. Because the v flag is cleared by a nonoverflow result of an `adc` instruction, it is not usually necessary to clear it before an addition operation. So, until the advent of the `bra` (branch always) and `brl` (branch always—long) instructions, the most common use of the `clv` instruction was to force an unconditional branch with a sequence of code such as

```
clv
bvc SOMEPLACE
```

Now, the `bra` and `brl` instructions have made such sequences as this one unnecessary. It is up to you to find some useful function for the `clv` instruction.

Incidentally, there is no specific instruction for setting the v flag. It can, however, be set with the 65C02/65C816 instruction `rep` or by using a `bit` instruction with a mask that has bit 6 set.

Flags affected: v

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	B8

cmp — compare with accumulator — 6502, 65C02, 65C816

Compares a specified literal number or the contents of a specified memory location with the contents of the accumulator. The n, z, and c flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends on what branch instruction is used and whether the value in the accumulator is less than, equal to, or more than the value tested.

When a `cmp` instruction is issued, the contents of the specified memory location are subtracted from the accumulator. The result is not stored in the acummulator, but the n, z, and c flags are conditioned as follows.

The z flag is set if the result of the comparison is 0 and cleared otherwise. The n flag is set or cleared by the condition of the sign bit (bit 7) of the result. The c flag is set if the value in the accumulator is greater than or equal to the value in memory. A `bcc` instruction can then be used to detect if the

value in the accumulator is greater than the value in memory. The `beq` instruction can detect if the two values are equal. The `bcs` instruction can detect if the value in the accumulator is greater than or equal to the value in memory. A `beq` followed by `bcs` can detect if the value in the accumulator is greater than the value in memory.

Flags affected: n, z, c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
`cmp (d)`	2	D2
`cmp (d),y`	2	D1
`cmp (d,x)`	2	C1
`cmp (r,s),y`	2	D3
`cmp d`	2	C5
`cmp d,x`	2	D5
`cmp r,s`	2	C3
`cmp [d]`	2	C7
`cmp [d],y`	2	D7
`cmp #`	2 (3)	C9
`cmp a`	3	CD
`cmp a,x`	3	DD
`cmp a,y`	3	D9
`cmp al`	4	CF
`cmp al,x`	4	DF
`cmp i`	2	02

cop — coprocessor enable — 65C816

The `cop` instruction allows the 65C816 to turn control over to another processor, such as a math, graphics, or music chip. When the coprocessor completes its assignment, it can return control to the 65C816.

The `cop` instruction, much like a `brk` instruction, causes a software interrupt, but through a different vector: $00FFF4 and $00FFF5.

When a `cop` instruction is issued, the program counter is incremented by 2 and pushed onto the stack. This operation allows the programmer to follow `cop` with a signature byte that specifies which coprocessor handling routine to execute. Unlike the `brk` instruction, which makes a signature byte optional, the `cop` instruction requires a signature byte. Signature bytes from $80 through $FF are reserved by the Western Design Center, which designed the 65C816. Signature bytes in the range $00 through $7F are available for use in application programs.

There are some differences between the way `cop` works in emulation mode and native mode. When a `cop` instruction is used in emulation mode, the program counter is incremented by 2 and pushed onto the stack, the status register is pushed onto the stack, the interrupt disable flag is set, and the

program counter is loaded from the emulation mode coprocessor vector at $FFF4-FFF5. Then, after the command is executed, the P register's d (decimal) flag is cleared.

When a `cop` instruction is issued in native mode, the program counter bank register is pushed onto the stack, the program counter is incremented by 2 and pushed onto the stack, the status register is pushed onto the stack, the interrupt disable flag is set, the program bank register is cleared to 0, and the program counter is loaded from the native mode coprocessor vector at $00FFE4–00FFE5. Then, after the instruction is issued, the d (decimal) flag is cleared.

Flags affected: d, i

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	2†	02

†`cop` is a 1-byte instruction, but the program counter is incremented by 2 before it is pushed onto the stack, allowing (in fact requiring) a signature byte to be used following the instruction.

cpa

`cpa` is not a 65C816 instruction, but an alias that the APW assembler recognizes as an alternate for the assembly language statement `cmp a`. For further details, see `cmp`.

cpx compare with X register 6502, 65C02, 65C816

Compares a specified literal number or the contents of a specified memory location with the contents of the X register. The n, z, and c flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used and whether the value in the X register is less than, equal to, or more than the value tested.

When a `cpx` instruction is issued, the contents of the specified memory location are subtracted from the value of the X register. The result is not stored in the X register, but the n, z, and c flags are conditioned as follows.

The z flag is set if the result of the comparison is 0 and cleared otherwise. The n flag is set or cleared by the condition of the sign bit (bit 7) of the result. The c flag is set if the value in the X register is greater than or equal to the value in memory. A `bcc` instruction can then be used to detect if the value in the X register is greater than the value in memory. A `beq` instruction can detect if the two values are equal. A `bcs` instruction can detect if the value in the X register is greater than or equal to the value in memory. A `beq`

instruction followed by `bcs` can detect if the value in the X register is greater than the value in memory.

Flags affected: n, z, c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
`cpx d`	2	E4
`cpx #`	2 (3)	E0
`cpx a`	3	EC

cpy — compare with Y register 6502, 65C02, 65C816

Compares a specified literal number or the contents of a specified memory location with the contents of the Y register. The n, z, and c flags of the status register are affected by this operation, and a branch instruction usually follows. The result of the operation thus depends upon what branch instruction is used and whether the value in the Y register is less than, equal to, or more than the value tested.

When a `cpy` instruction is issued, the contents of the specified memory location are subtracted from the value of the Y register. The result is not stored in the Y register, but the n, z, and c flags are conditioned as follows.

The z flag is set if the result of the comparison is 0 and cleared otherwise. The n flag is set or cleared by the condition of the sign bit (bit 7) of the result. The c flag is set if the value in the Y register is greater than or equal to the value in memory. A `bcc` instruction can then be used to detect if the value in the Y register is greater than the value in memory. A `beq` instruction can detect if the two values are equal. A `bcs` instruction can detect if the value in the Y register is greater than or equal to the value in memory. A `beq` instruction followed by `bcs` can detect if the value in the Y register is greater than the value in memory.

Flags affected: n, z, c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
`cpy d`	2	C4
`cpy #`	2 (3)	C0
`cpy a`	3	CC

dea

`dea` is not a 65C816 instruction, but an alias that the APW assembler recognizes as an alternate for the assembly language statement `dec a`. For further details, see `dec`.

dec — decrement a memory location — 6502, 65C02,65C816

Decrements the contents of a specified memory location by 1. It is important to note that `dec` does not affect the carry flag. Thus, if the value to be decremented is $00, the result of the `dec` operation is $FF.

Because `dec` does not change the carry flag, the carry flag cannot be used to test the outcome of a `dec` operation. A `dec` instruction does condition the n and z flags, however, so they can be used to test a value decremented by `dec`.

Flags affected: n, z

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
`dec Acc`	1	3A
`dec d`	2	C6
`dec d,x`	2	D6
`dec a`	3	CE
`dec a,x`	3	DE

dex — decrement the X register — 6502, 65C02, 65C816

Decrements the contents of the X register by 1. It is important to note that `dex` does not affect the carry flag. Thus, if the value to be decremented is $00, the result of `dex` is $FF.

Because `dex` does not change the carry flag, the carry flag cannot be used to test the outcome of a `dex` operation. The `dex` instruction does condition the n and z flags, however, so they can be used to test a value decremented by `dex`.

Flags affected: n, z

Registers affected: P, M

Addressing Mode	Bytes	Opcode (hex)
`i`	1	CA

dey — decrement the Y register — 6502, 65C02, 65C816

Decrements the contents of the Y register by 1. It is important to note that `dey` does not affect the carry flag. Thus, if the value to be decremented is $00, the result of `dey` is $FF.

Because `dey` does not change the carry flag, the carry flag cannot be used to test the outcome of a `dey` operation. The `dey` instruction does condition the n and z flags, however, so they can be used to test a value decremented by `dey`.

Flags affected: n, z
Registers affected: P, M

Addressing Mode	Bytes	Opcode (hex)
i	1	88

eor — exclusive-OR with accumulator — 6502, 65C02, 65C816

Performs an exclusive-OR operation on the contents of the accumulator and a specified literal value or memory location. Each bit in the accumulator is EORed with the corresponding bit in the operand, and the result of the operation is stored in the accumulator. See figure A-3.

The `eor` instruction is often used as a mask, to set specified bits in a memory location. When used as a mask, the instruction compares each bit in a memory location with the corresponding bit in the accumulator. If one and only one of the two bits being compared is set, the corresponding bit in the accumulator is set. Otherwise, the corresponding bit in the accumulator is cleared.

When `eor` is used with a mask consisting of all ones—that is, a mask of $FFFF in native mode or a mask of $FF in emulation mode—each bit in the operand is reversed; that is, each set bit is cleared, and each cleared bit is set. So `eor` is used quite often to reverse the settings of the bits in a word or a byte.

Here is another useful characteristic of `eor`. When it is used on a value twice in succession and with the same operand, the value is changed to another value the first time the instruction is used, and it is converted back into its original value the second time the instruction is used. Because of this characteristic, the `eor` instruction is often used to encode values and then to restore them to their original states. To encode a value using `eor`, just perform an EOR operation on it using an arbitrary 1-byte key. Later, the value can be restored to its original state by performing another EOR operation using the same key.

The `eor` instruction conditions the P register's n and z flags. The n flag is set if the most significant bit of the result of the EOR operation is set; otherwise, it is cleared. The z flag is set if the result is 0; otherwise, it is cleared.

In emulation mode, `eor` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

```
  0       0       1       1
EOR 0   EOR 1   EOR 0   EOR 1
  0       1       1       0
```

Figure A-3
Truth table for EOR

Flags affected: n, z
Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`eor (d)`	2	52
`eor (d),y`	2	51
`eor (d,x)`	2	41
`eor (r,s),y`	2	53
`eor d`	2	45
`eor d,x`	2	55
`eor r,s`	2	43
`eor [d]`	2	47
`eor [d],y`	2	57
`eor #`	2 (3)	49
`eor a`	3	4D
`eor a,x`	3	5D
`eor a,y`	3	59
`eor al`	4	4F
`eor al,x`	4	5F

ina

`ina` is not a 65C816 instruction, but an alias that the APW assembler recognizes as an alternate for the assembly language statement `inc a`. For further details, see `inc`.

inc — increment memory — 6502, 65C02, 65C816

Increments the contents of a specified memory location by 1. The `inc` instruction neither affects nor is affected by the carry flag. So, if a value being incremented is $FF, the result of the `inc` operation is $00. Because `inc` does not affect the carry flag, the result of an `inc` operation cannot be tested by checking the carry flag. It does condition the n and z flags, however, so they can be used to test the result of an `inc` operation.

Flags affected: n, z
Registers affected: M, P

Addressing Mode	Bytes	Opcode (hex)
`inc Acc`	1	1A
`inc d`	2	E6

`inc d,x`	2	F6
`inc a`	3	EE
`inc a,x`	3	FE

inx — increment X register — 6502, 65C02, 65C816

Increments the contents of the X register by 1. The `inx` instruction neither affects nor is affected by the carry flag. So, if a value being incremented is $FF, the result of the `inx` operation is $00. Because `inx` does not affect the carry flag, the result of an `inx` operation cannot be tested by checking the carry flag. It does condition the n and z flags, however, so they can be used to test the result of an `inx` operation.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
i	1	E8

iny — increment Y register — 6502, 65C02, 65C816

Increments the contents of the Y register by 1. The `iny` instruction neither affects nor is affected by the carry flag. So, if a value being incremented is $FF, the result of the `iny` operation is $00. Because `iny` does not affect the carry flag, the result of an `iny` operation cannot be tested by checking the carry flag. It does condition the n and z flags, however, so they can be used to test the result of an `iny` operation.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
i	1	C8

jmp — jump to address — 6502, 65C02, 65C816

Causes program execution to jump to the address specified. When a `jmp` instruction is issued, the program counter is loaded with the target address, causing control of the program in progress to be shifted to that address. When `jmp` is used in the absolute addressing mode, its operand can be either 16 bits or 24 bits. If a 16-bit address is used, the destination of the jump can be anywhere within the current program bank. If a 24-bit address is used, the jump is referred to as a long jump, and its destination address can be anywhere within the address space of the IIGS. When `jmp` carries out a long jump, it has the same result as the `jml` instruction.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`jmp (a)`	3	6C
`jmp (a,x)`	3	7C
`jmp a`	3	4C
`jmp al`	4	5C

jsl jump to subroutine—long 65C816

Jumps to a subroutine using long (24-bit) addressing. The `jsl` instruction takes a 24-bit operand. It pushes a 24-bit (long) return address onto the stack, then transfers control to the subroutine at the 24-bit address that is the operand. This return address is the address of the last instruction byte (the fourth instruction byte, or the third operand byte), not the address of the next instruction. It is the return address minus 1.

When you issue a `jsl` instruction, the current program counter bank is pushed onto the stack first. Then the high-order byte and the low-order byte of the address are pushed onto the stack in standard 6502/65C816 order, low byte first. The program bank register and the program counter are then loaded with the effective address specified by the operand, and control is transferred to the specified address.

Flags affected: None
Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`al`	4	22

jsr jump to subroutine 6502, 65C02, 65C816

Causes program execution to jump to the address that follows the instruction. That address should be the starting address of a subroutine that ends with the `rts` instruction. When the program reaches the `rts` instruction, execution of the program returns to the next instruction after the `jsr` instruction that caused the jump to the subroutine.

When a `jsr` instruction is issued, the high-order byte and the low-order byte of the address are pushed onto the stack in standard 6502/65C816 order,

low byte first. The program counter is then loaded with the effective address specified by the operand, and control is transferred to the specified address.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`jsr (a,x)`	3	FC
`jsr a`	3	20

lda — load the accumulator — 6502, 65C02, 65C816

Loads the accumulator with the contents of the effective address of the operand. The n flag is set if a value with the high bit set is loaded into the accumulator. The z flag is set if the value loaded into the accumulator is 0.

In emulation mode, `lda` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`lda (d)`	2	B2
`lda (d),y`	2	B1
`lda (d,x)`	2	A1
`lda (r,s),y`	2	B3
`lda d`	2	A5
`lda d,x`	2	B5
`lda r,s`	2	A3
`lda [d]`	2	A7
`lda [d],y`	2	B7
`lda #`	2 (3)	A9
`lda a`	3	AD
`lda a,x`	3	BD
`lda a,y`	3	B9
`lda al`	4	AF
`lda al,x`	4	BF

ldx — load the X register — 6502, 65C02, 65C816

Loads the X register with the contents of the effective address of the operand. The n flag is set if a value with the high bit set is loaded into the X register. The z flag is set if the value loaded into the X register is 0.

In emulation mode, `ldx` is an 8-bit operation. In native mode, it is a

16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
ldx d	2	A6
ldx d,y	2	B6
ldx #	2 (3)	A2
ldx a	3	AE
ldx a,y	3	BE

ldy — load the Y register — 6502, 65C02, 65C816

Loads the Y register with the contents of the effective address of the operand. The n flag is set if a value with the high bit set is loaded into the Y register. The z flag is set if the value loaded into the Y register is 0.

In emulation mode, ldy is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
ldy d	2	A4
ldy d,x	2	B4
ldy #	2 (3)	A0
ldy a	3	AC
ldy a,x	3	BC

lsr — logical shift right — 6502, 65C02, 65C816

Moves each bit in the accumulator one position to the right. See figure A-4. A 0 is deposited into the leftmost position (bit 15 in native mode and bit 7 in emulation mode), and bit 0 is deposited into the carry. The result is left in the accumulator or in the affected memory register.

In emulation mode, lsr is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

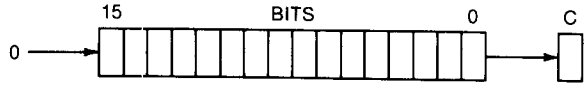


Figure A-4
LSR operation

Flags affected: n, z, c
Registers affected: A, P, M

Addressing Mode	Bytes	Opcode (hex)
`lsr Acc`	1	4A
`lsr d`	2	46
`lsr d,x`	2	56
`lsr a`	3	4E
`lsr a,x`	3	5E

mvn — move block next, or move block negative — 65C816

Copies a block of memory from one RAM address to another. Both `mvn` and the 65C816's other block move instruction, `mvp` (move block previous, or move block positive), can copy blocks from one bank to another and can copy memory blocks that overlap. When overlapping blocks are moved, however, `mvn` should be used only if the starting address of the block to be moved is higher than the starting address of the destination. If the blocks overlap and the starting address of the destination is higher than the starting address of the source, use the `mvp` instruction. Otherwise, part of the block being copied may be overwritten.

The `mvn` instruction takes two operands, each consisting of 1 byte. In programs written using the APW assembler-editor, the operands are separated by a comma. The first operand specifies the bank containing the block to be moved, and the second specifies the bank to which the block will be moved.

The source address, destination address, and length of the move are passed to the `mvn` instruction in the X, Y, and C (double accumulator) registers. The X register holds the source address, the Y register holds the destination address, and the C register holds the length of the block being moved, minus 1. For example, if the C register holds the value $00FF, 256 bytes (or $FF bytes in hexadecimal notation) are moved. The complete C register is always used, regardless of the setting of the m flag.

When you issue an `mvn` instruction, the first byte to be moved is copied from the source address stored in the X register to the destination address stored in the Y register. Then the X and Y registers are incremented. Next, the C register is decremented, and the next byte is moved. This sequence of operations continues until the number of bytes originally stored in the C register, plus 1, are moved (until the value in C is $FFFF).

When the execution of an `mvn` operation is complete, the X and Y registers point to addresses that lie 1 byte beyond the ends of the blocks to which they originally pointed. The data bank register holds the value of the destination bank value (the value of the first byte of the operand).

If the source and destination blocks do not overlap, the source block remains intact after it is copied to the destination.

The operand field of the `mvn` instruction must be coded as two addresses:

first the source, then the destination. When the instruction is assembled into machine code, however, this order is reversed.

If the 65C816 receives an interrupt while an `mvn` move is in progress, the copying of the byte being moved is completed and then the interrupt is serviced. If the interrupt handling routine restores all registers or leaves them intact and ends with an `rti` instruction, the block move is resumed automatically when the interrupt ends.

The `mvn` instruction is useful when blocks of code are moved from one bank to another. For moves that take place within one bank, however, operations that use other algorithms may be faster and more efficient.

If the 65C816 is in emulation mode or the A, X, and Y registers are in 8-bit mode when the `mvn` instruction is issued, both addresses specified in the operand must be on page 0 because the high bytes of the index registers contain zeros.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
`xya`	3	54

mvp — move block previous, or move block positive — 65C816

Copies a block of memory from one RAM address to another. Both `mvp` and the 65C816's other block move instruction, `mvn` (move block next, or move block negative), can copy blocks from one bank to another and can copy memory blocks that overlap. When overlapping blocks are moved, however, `mvp` should be used only if the starting address of the block to be moved is lower than the starting address of the destination. If the blocks overlap and the starting address of the destination is higher than the starting address of the source, use the `mvn` instruction. Otherwise, part of the block being copied may be overwritten.

The `mvp` instruction takes two operands, each consisting of 1 byte. In programs written using the APW assembler-editor, the operands are separated by a comma. The first operand specifies the bank containing the block to be moved, and the second specifies the bank to which the block will be moved.

The source address, destination address, and length of the move are passed to the `mvp` instruction in the X, Y, and C (double accumulator) registers. The X register holds the address of the last byte of the block to be moved, the Y register holds the last byte of the destination block, and the C register holds the length of the block being moved, minus 1. For example, if the C register holds the value $00FF, 256 bytes (or $FF bytes in hexadecimal notation) are moved. The complete C register is always used, regardless of the setting of the m flag.

When you issue an `mvp` instruction, the first byte to be moved is copied from the source address stored in the X register to the destination address

stored in the Y register. Then the X and Y registers are decremented. Next, the C register is decremented, and the next byte is moved. This sequence of operations continues until the number of bytes originally stored in the C register, plus 1, are moved (until the value in C is $FFFF).

When the execution of an `mvp` operation is complete, the X and Y registers point to addresses that lie 1 byte past the starting addresses of the blocks to which they originally pointed. The data bank register holds the value of the destination bank value (the value of the first byte of the operand).

If the source and destination blocks do not overlap, the source block remains intact after it is copied to the destination.

The operand field of the `mvp` instruction must be coded as two addresses: first the address of the last byte of the source block, then the address of the last byte of the destination block. When the instruction is assembled into machine code, however, this order is reversed.

If the 65C816 receives an interrupt while an `mvp` move is in progress, the copying of the byte being moved is completed and then the interrupt is serviced. If the interrupt handling routine restores all registers or leaves them intact and ends with an `rti` instruction, the block move is resumed automatically when the interrupt ends.

The `mvp` instruction is useful when blocks of code are moved from one bank to another. For moves that take place within one bank, however, operations that use other algorithms may be faster and more efficient.

If the 65C816 is in emulation mode or the A, X, and Y registers are in 8-bit mode when the `mvp` instruction is issued, both addresses specified in the operand must be on page 0 because the high bytes of the index registers contain zeros.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
xya	3	44

nop — no operation — 6502, 65C02, 65C816

Causes the 65C816 to wait, and do nothing, for one or more cycles. The `nop` instruction does not affect any registers except the program counter, which is incremented once to point to the next instruction.

The `nop` instruction is often used to indicate spots in a program where more code may be inserted. For example, in a sequence such as

```
LAB1  nop
      lda #$FF
```

you could insert more lines of source code between the `nop` and `lda` instructions, without retyping the line containing the label `LAB1`.

The nop instruction can also be used to take up time. Every nop in a program takes two cycles, so nop instructions are often used in delay loops and to adjust the speeds of loops in which timing is important.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	1	EA

ora

OR accumulator with memory — 6502, 65C02, 65C816

Performs a binary inclusive-OR operation on the value in the accumulator and a literal value or the contents of a specified memory location or immediate value. See figure A-5. Each bit in the accumulator is ORed with the corresponding bit in the operand, and the result of the operation is stored in the accumulator.

The ora instruction is often used as a mask, to set specified bits in a memory location. When used as a mask, the instruction compares each bit in a memory location with the corresponding bit in the accumulator. Each bit set in the memory location sets the corresponding bit in the accumulator. Bits cleared in the accumulator have no effect on their corresponding bits in the memory location. For example, the sequence

```
lda #$00FF
ora MEMLOC
sta MEMLOC
```

sets all bits in the the low-order byte of MEMLOC, while leaving the high-order byte of MEMLOC unchanged.

The ora instruction conditions the P register's n and z flags. The n flag is set if the most significant bit of the result of the ORA operation is set; otherwise, it is cleared. The z flag is set if the result is 0; otherwise it is cleared.

In emulation mode, ora is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: A, P

0	0	1	1
ORA 0	ORA 1	ORA 0	ORA 1
0	1	1	1

Figure A-5
Truth table for ORA

Addressing Mode	Bytes	Opcode (hex)
ora (d)	2	12
ora (d),y	2	11
ora (d,x)	2	01
ora (r,s),y	2	13
ora d	2	05
ora d,x	2	15
ora r,s	2	03
ora [d]	2	07
ora [d],y	2	17
ora #	2 (3)	09
ora a	3	0D
ora a,x	3	1D
ora a,y	3	19
ora al	4	0F
ora al,x	4	1F

pea — push effective address — 65C816

Pushes a 16-bit operand, always expressed in absolute addressing mode, onto the stack. This operation always pushes 16 bits of data, regardless of the settings of the m and x mode select flags, and the stack pointer is decremented twice.

Although the mnemonic `pea` would seem to suggest that the value pushed onto the stack must be an address, the instruction can actually be used to place any 16-bit value on the stack. For instance, the instruction

```
pea 0
```

pushes a 0 on the stack. Notice, however, that when `pea` places a literal value on the stack, the context is unusual. The operand of the instruction is interpreted by the assembler as a literal value. Thus, it does not require the prefix # to designate it as a literal value. So, in this example, a literal 0, not the value of memory address $0000, is pushed onto the stack.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	3	F4

pei — push effective indirect address — 65C816

Pushes the 16-bit value located at the address formed by adding the direct page offset specified by the operand to the direct page register. Although the

mnemonic `pei` may seem to suggest that the instruction's operand must be an address, it actually can be any kind of 16-bit data. The instruction always pushes 16 bits of data, regardless of the settings of the m and x mode select flags.

The first byte pushed is the byte at the direct page offset plus 1 (the high byte of the double byte stored at the direct page offset). The byte of the direct page offset itself (the low byte) is pushed next. The stack pointer then points to the next available stack location, directly below the last byte pushed.

The syntax of the `pei` instruction is that of direct page indirect. Unlike other instructions that use this syntax, however, the effective indirect address, rather than the data stored at that address, is pushed onto the stack.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	2	D4

per — push effective PC relative indirect address — 65C816

Adds the current value of the program counter to the value of a 2-byte operand and pushes the result on the stack. When the program counter is added to the operand, it contains the address of the next instruction (the instruction following the `per` instruction).

After the program counter and the operand are added, the high byte of their sum is pushed onto the stack first, followed by the low byte. After the instruction is completed, the stack pointer points to the next available stack location, immediately below the last byte pushed. The `per` instruction always pushes 16 bits of data, regardless of the settings of the m and x mode select flags.

The syntax used with the `per` instruction is similar to that used with branch instructions; that is, the data to be referenced is used as an operand. The address referred to must be in the current program bank because `per`'s displacement is relative to the program counter.

The `per` instruction is useful when you write self-relocatable code in which a given address (typically the address of a data area) must be accessed. In this kind of application, the address pushed onto the stack is the run time address of the data area, regardless of where the program was loaded in memory. It could be pulled into a register, stored in an indirect pointer, or used on the stack with the stack relative indirect indexed addressing mode to access the data at that location.

The `per` instruction can also be used to push return addresses on the stack, either as part of a simulated branch-to-subroutine or to place the return address beneath the stacked parameters to a subroutine call. When `per` is

used in this way, it should be noted that a pushed return address should be the desired return address minus 1.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	3	62

pha — push accumulator

Pushes the contents of the accumulator on the stack. The accumulator and the P register are not affected.

In emulation mode, pha is an 8-bit operation. The contents of an 8-bit accumulator are pushed on the stack, and the stack pointer is decremented by 1.

In native mode, pha is a 16-bit operation. The high byte in the accumulator is pushed first, then the low byte. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
s	1	48

phb — push data bank register — 65C816

Pushes the value of the data bank register (DBR) onto the stack. The stack pointer then points to the next available stack location, directly below the byte pushed. The data bank register itself is left unchanged.

The 65C816 data bank register is an 8-bit register, so only 1 byte is pushed onto the stack, regardless of the settings of the m and x (mode select) flags.

The phb instruction allows the programmer to save the current value of the data bank register before changing the data bank's value. It is therefore useful when a program in one bank must access data in another. After the data in the other bank is accessed, the original value of the data bank register can be restored.

Flags affected: None

Register affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	8B

phd — push direct page register — 65C816

Pushes the contents of the direct page register (D) onto the stack. The most important use of the phd instruction is to save the value of the D register temporarily, prior to starting an operation that may change its value. After the contents of the D register are saved, a subroutine may specify its own direct page. Then, after the subroutine ends, the original value of the D register can be restored.

Because the direct page register is always a 16-bit register, phd is always a 16-bit operation, regardless of the settings of the m and x (mode select) flags. When you use this instruction, the high byte of the direct page register is pushed first, then the low byte. The direct page register itself is unchanged. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None

Register affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	0B

phk — push program bank register — 65C816

Pushes the current value of the program bank register onto the stack. The phk instruction is often used to set the data bank register and the program bank register so that they contain the same values. A program can then access data in its own bank.

To make the program bank register and the data bank register the same, the following sequence is often used:

```
phk          ; push contents of PBR on stack
plb          ; pull PBR value into data bank register
```

When the phk instruction is used, the program bank register itself is unchanged. The stack pointer then points to the next available stack location, directly below the byte pushed. Because the program bank register is an 8-bit register, only 1 byte is pushed onto the stack, regardless of the settings of the m and x (mode select) flags.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	48

php — push processor status — 6502, 65C02, 65C816

Pushes the contents of the P register on the stack. The P register itself is left unchanged, and no other registers are affected.

Because the program bank register is an 8-bit register, only 1 byte is pushed onto the stack, regardless of the settings of the m and x (mode select) flags.

Note that the P register's e flag, a "hanging flag," is not pushed onto the stack by the `php` instruction. The only way to access the e flag is with the `xce` instruction.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
s	1	08

phx — push X register — 65C02, 65C816

Pushes the contents of the X index register onto the stack. The X register itself is unchanged.

When the 65C816 is in emulation mode or when the X and Y registers are set to 8-bit lengths, the 8-bit contents of the X register are pushed onto the stack. The stack pointer then points to the next available stack location, directly below the byte pushed.

When the 65C816 is in native mode and the X and Y registers are set to 16-bit lengths, the 16-bit contents of the X register are pushed onto the stack. The high byte is pushed first, then the low byte. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	DA

phy — push Y register — 65C02, 65C816

Pushes the contents of the Y index register onto the stack. The Y register itself is unchanged.

When the 65C816 is in emulation mode or when the X and Y registers are set to 8-bit lengths, the 8-bit contents of the Y register are pushed onto

the stack. The stack pointer then points to the next available stack location, directly below the byte pushed.

When the 65C816 is in native mode and the X and Y registers are set to 16-bit lengths, the 16-bit contents of the X register are pushed onto the stack. The high byte is pushed first, then the low byte. The stack pointer then points to the next available stack location, directly below the last byte pushed.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
s	1	5A

pla — pull accumulator — 6502, 65C02, 65C816

Removes 1 byte from the stack and deposits it in the accumulator. The n and z flags are conditioned, just as if an `lda` operation had been carried out.

When the 6C816 is in emulation mode or when the accumulator is set to an 8-bit length, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the accumulator.

When the 65C816 is in native mode and the accumulator is set to a 16-bit length, the low-order byte of the accumulator is pulled first, followed by the high-order byte.

Flags affected: n, z

Registers affected: A, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	68

plb — pull data bank register — 6502, 65C02, 65C816

Pulls the 8-bit value on top of the stack into the data bank register (B) and changes the value of the data bank to that value. All instructions referencing data that specifies only 16-bit addresses will then get their bank address from the value pulled into the data bank register. This is the only instruction that can modify the data bank register.

The `plb` instruction is often used with `phk` (push program bank register) to set the data bank register and the program bank register so that they contain the same values. A program can then access data that is in its own bank. To make the program bank register and the data bank register the same, the following sequence is often used:

```
phk          ; push contents of PBR on stack
plb          ; pull PBR value into data bank register
```

When plb is used in a program, the stack pointer is incremented, then the byte pointed to by the stack pointer is loaded into the register. Because the bank register is an 8-bit register, plb pulls only 1 byte from the stack, regardless of the settings of the m and x (mode select) flags.

Flags affected: n, z

Registers affected: B, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	AB

pld — pull direct page register — 65C816

Pulls the 16-bit value on top of the stack into the direct page register (D), giving the D register a new value.

The most common use of pld is to restore the direct page register to a previous value. When a program calls a subroutine that has its own direct page, the program can save its direct page by using the instruction phd (push direct page) before the subroutine is called. When the subroutine ends and control returns to the program that called it, the original state of the D register can be restored with a pld instruction.

The direct page register is a 16-bit register, so 2 bytes are pulled from the stack, regardless of the settings of the m and x (mode select) flags. The low byte of the direct page register is pulled first, then the high byte. The stack pointer then points to where the high byte just pulled was stored, and that is the next available stack location.

Flags affected: n, z

Register affected: D, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	2B

plp — pull processor status register — 6502, 65C02, 65C816

Pulls the 8-bit value on top of the stack into the processor status register (P), changing the value of the P register. plp is often used to restore flag settings previously saved on the stack with a php (push processor status register) instruction.

It should be noted, however, that the P register's e flag (the emulation mode flag) cannot be retrieved from the stack with a plp instruction. That is because it is a "hanging flag" that is not pushed on the stack by the php instruction. The only way to set the e flag is with the xce instruction.

The status register is an 8-bit register, so only 1 byte is pulled from the stack by the plp instruction, regardless of the settings of the m and x (mode select) flags. When the instruction is used in a program, the stack pointer is

first incremented. Then the byte pointed to by the stack pointer is loaded into the status register.

Flags affected: All except e

Registers affected: S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	28

plx — pull X register from stack — 65C02, 65C816

Pulls the value on top of the stack into the X index register, destroying the register's previous contents. This operation conditions the n and z flags.

When the 65C816 is in emulation mode or when the X register is set to an 8-bit length, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the X register.

When the 65C816 is in native mode and the X register is set to a 16-bit length, the low-order byte of the X register is pulled first, followed by the high-order byte.

Flags affected: n, z

Registers affected: X, S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	FA

ply — pull Y register from stack — 65C02, 65C816

Pulls the value on top of the stack into the Y index register, destroying the register's previous contents. This operation conditions the n and z flags.

When the 65C816 is in emulation mode or the Y register is set to an 8-bit length, the stack pointer is first incremented. Then the byte pointed to by the stack pointer is loaded into the Y register.

When the 65C816 is in native mode and the Y register is set to a 16-bit length, the low-order byte of the Y register is pulled first, followed by the high-order byte.

Flags affected: n, z

Registers affected: S, Y, P

Addressing Mode	Bytes	Opcode (hex)
s	1	7A

rep — reset status bits — 65C816

Clears flags in the status register according to the contents of an 8-bit operand. For each bit set to 1 in the operand, `rep` resets, or clears, the corresponding

bit in the status register to 0. For example, if bit 5 in the operand byte is set, bit 5 in the P register is cleared to 0. Zeros in the operand byte have no effect on their corresponding status register bits.

The `rep` instruction allows the programmer to reset any flag or combination of flags in the status register with a single 2-byte instruction. It is the only direct means of clearing the m flag and the x flag (although instructions that pull the P register affect the m and x flags).

When the 65C816 is in emulation mode, `rep` does not affect the break flag or bit 5, the 6502's undefined flag bit. In native mode, however, all flags except the e flag (the "hanging" flag) can be cleared with the `rep` instruction. The only way to access the e flag is with the `xce` instruction.

Flags affected in native mode: All flags except e

Flags affected in emulation mode: All flags except b

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
#	2	C2

rol — rotate left — 6502, 65C02, 65C816

Moves each bit in the accumulator or a specified memory location one position to the left. See figure A-6.

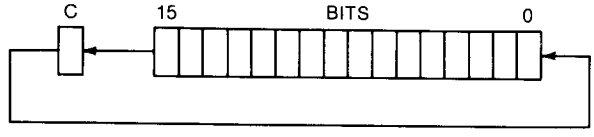


Figure A-6
ROL operation

The carry bit is deposited into the bit 0 location and is replaced by the leftmost bit (bit 15 in native mode and bit 7 in emulation mode) of the accumulator or the affected memory register. The n, z, and c flags are conditioned according to the result of the rotation operation.

Flags affected: n, z, c

Registers affected: A, P, M

Addressing Mode	Bytes	Opcode (hex)
`rol Acc`	1	2A
`rol d`	2	26
`rol d,x`	2	36
`rol a`	3	2E
`rol a,x`	3	3E

ror — rotate right — 6502, 65C02, 65C816

Moves each bit in the accumulator or a specified memory location one position to the right. See figure A-7.

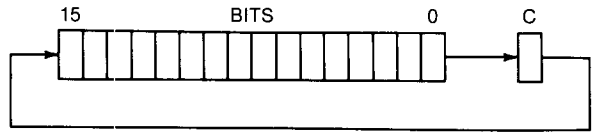


Figure A-7
ROR operation

The carry bit is deposited into the leftmost location (bit 15 in native mode and bit 7 in emulation mode) and is replaced by bit 0 of the accumulator or the affected memory register. The n, c, and z flags are conditioned according to the result of the rotation operation.

Flags affected: n, z, c

Registers affected: A, P, M

Addressing Mode	Bytes	Opcode (hex)
`ror Acc`	1	6A
`ror d`	2	66
`ror d,x`	2	76
`ror a`	3	6E
`ror a,x`	3	7E

rti — return from interrupt — 6502, 65C02, 65C816

The status of both the program counter and the P register are pulled from the stack and restored to their original values in preparation for resuming the routine in progress when an interrupt occurred. If the 65C816 is in native mode, the program bank register is also pulled from the stack. The `rti` instruction is used to end interrupt handling routines and return control to the program in progress when the interrupt occurred.

The `rti` instruction pulls values off the stack in the reverse order from the way they were pushed onto the stack by a hardware interrupt (IRQ) or a software interrupt (`brk` or `cop`). It is up to the interrupt handling routine to ensure that the values pulled off the stack by `rti` are valid.

When the 65C02 is in native mode, 4 bytes are pulled from the stack: the 8-bit status register, the 16-bit program counter, and the 8-bit program bank register.

In emulation mode, 3 bytes are pulled from the stack: the status register and the program counter.

Flags affected: n, v, b, d, i, z, c
Registers affected: S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	40

rtl return from subroutine long 65C816

Returns to the program in progress from a subroutine that was called using the instruction `jsl` (jump to subroutine—long).

When you call a subroutine using `jsl`, the 8-bit value of the program bank register is pushed onto the stack, followed by the 16-bit value of the program counter.

When you use an `rtl` instruction to end a subroutine, the instruction pulls the value of the program counter from the stack, increments it by 1, and loads the incremented value into the program counter. Then it pulls the program bank register off the stack and loads that into the program bank register.

Flags affected: all except e
Register affected: S, P

Addressing Mode	Bytes	Opcode (hex)
s	1	6B

rts return from subroutine 6502, 65C02, 65C816

At the end of a subroutine, `rts` returns execution of a program to the next address after the `jsr` (jump to subroutine) instruction that caused the program to jump to the subroutine. At the end of an assembly language program, the `rts` instruction returns control of the IIGS to the utility that was in control before the program began.

When a subroutine is called in a 65C816 program with a `jsr` instruction, the contents of the program counter (a 16-bit value) are pushed onto the stack. When the subroutine ends with an `rts` instruction, the `rts` instruction pulls the return address from the stack, increments it, and places it in the program counter, transferring control back to the instruction immediately following the `jsr` instruction.

The instructions `jsr` and `rts` do not push or pull the contents of the program bank register. Therefore, they cannot be used to jump across bank boundaries. When a program must cross a bank boundary to jump to a subroutine, it must use the instructions `jsl` (jump to subroutine—long) and `rtl` (return from subroutine—long).

Flags affected: None
Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
i	1	60

sbc — subtract with carry — 6502, 65C02, 65C816

Subtracts the content of the effective address of the operand from the contents of the accumulator. The opposite of the carry flag is also subtracted; because subtraction is really reverse addition, the carry flag in a subtraction operation is treated as a borrow.

Because of the way the carry flag is used in subtraction operations, you should set it before a subtraction takes place. Then, if there is a borrow by a lower-order word (or byte in emulation mode) from a higher-order word (or byte in emulation mode), the carry flag is cleared. That causes a borrow, and the result of the subtraction will be accurate.

In emulation mode, `sbc` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

The n, v, z, and c flags are all conditioned by the `sbc` instruction, and its result is deposited in the accumulator.

Flags affected: n, v, z, c

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
`sbc (d)`	2	F2
`sbc (d),y`	2	F1
`sbc (d,x)`	2	E1
`sbc (r,s),y`	2	F3
`sbc d`	2	E5
`sbc d,x`	2	F5
`sbc r,s`	2	E3
`sbc [d]`	2	E7
`sbc [d],y`	2	F7
`sbc #`	2 (3)	E9
`sbc a`	3	ED
`sbc a,x`	3	FD
`sbc a,y`	3	F9
`sbc al`	4	EF
`sbc al,x`	4	FF

sec — set carry — 6502, 65C02, 65C816

Sets the carry flag. The `sec` instruction is often used before the `sbc` instruction so that there is not an extra borrow in the subtraction operation. `sec` is also used prior to an `xce` (exchange carry flag with emulation bit) instruction if the intent of the instruction is to put the 65C816 into 8-bit emulation mode.

Flags affected: c

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	38

sed — set decimal mode — 6502, 65C02, 65C816

Sets the P register's d flag, taking the 65C816 out of normal binary mode and preparing it for operations using BCD (binary coded decimal) numbers. BCD arithmetic is more accurate than binary arithmetic—the usual type of 6510 arithmetic—but it is slower and more difficult to use and consumes more memory. BCD arithmetic is most often used in accounting programs, bookkeeping programs, and floating-point arithmetic.

The decimal flag can be cleared, returning the 65C816 to its default binary mode, with a `cld` (clear decimal flag) instruction.

Flags affected: d

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	F8

sei — set interrupt disable — 6502, 65C02, 65C816

Sets the P register's i (interrupt disable) flag, disabling the processing of hardware interrupts (IRQs). When the i bit is set, maskable hardware interrupts are ignored.

When the 65C816 begins servicing an interrupt, it sets the i flag, so interrupt handling routines that are themselves intended to be interruptable must reenable interrupts with a `cli` (clear interrupt) instruction. If other interrupts are to remain disabled during the interrupt being serviced, a `cli` instruction is not necessary, because the `rti` (return from interrupt) instruction automatically restores the status register with the i flag clear, reenabling interrupts.

Flags affected: i

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	78

sep — set status bits — 65C816

Sets bits in the processor status register according to the value of an 8-bit operand. For each bit set in the operand, `sep` sets the corresponding bit in the status register to 1. For example, if bit 5 is set in the operand byte, bit 5 in the status register is set to 1. Zeros in the operand byte have no effect on their corresponding bits in the P register.

The sep instruction enables the programmer to set any flag or combination of flags in the status register with a single 2-byte instruction. Also, it is the only direct means of setting the m and x (mode select) flags, although instructions that pull the P status register indirectly affect the m and x mode select flags.

When the 65C816 is in emulation mode, sep does not affect the break flag or bit 5, the 6502's non-flag bit.

Flags affected in native mode: n, v, m, x, d, i, z, c
Flags affected in emulation mode: n, v, d, i, z, c
Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
sep #	2	E2

sta store accumulator 6502, 65C02, 65C816

Stores the contents of the accumulator in a specified memory location. The contents of the accumulator are not affected.

In emulation mode, sta is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None
Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
sta (d)	2	92
sta (d),y	2	91
sta (d,x)	2	81
sta (r,s),y	2	93
sta dta	2	85
sta d,x	2	95
sta r,s	2	83
sta [d]	2	87
sta [d],y	2	97
sta a	3	8D
sta a,x	3	9D
sta a,y	3	99
sta al	4	8F
sta al,x	4	9F

stp stop the processor 6502, 65C02, 65C816

Puts the 65C816 into a dormant state until a hardware reset occurs, that is, until the processor's RES pin is pulled low.

The stp instruction is designed for use in battery-powered computers and other systems engineered to support a low-power mode. It can reduce

power consumption to almost 0 by putting the 65C816 out of action while it is not actively in use.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	1	DB

stx — store X register — 6502, 65C02, 65C816

Stores the contents of the X register in a specified memory location. The contents of the X register are not affected.

In emulation mode, `stx` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None

Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
`stx d`	2	86
`stx d,y`	2	96
`stx a`	3	8E

sty — store Y register — 6502, 65C02, 65C816

Stores the contents of the Y register in a specified memory location. The contents of the Y register are not affected.

In emulation mode, `sty` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None

Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
`sty d`	2	84
`sty d,x`	2	94
`sty a`	3	8C

stz — store zero to memory — 65C02, 65C816

Stores a 0 in the effective address specified by the operand. The `stz` instruction does not affect any of the flags in the P register.

In emulation mode, `stz` is an 8-bit operation. In native mode, it is a

16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: None

Registers affected: M

Addressing Mode	Bytes	Opcode (hex)
`stz d`	2	64
`stz d,x`	2	74
`stz a`	3	9C
`stz a,x`	3	9E

tax — transfer accumulator to X register — 6502, 65C02, 65C816

Deposits the value in the accumulator into the X register. The n and z flags are conditioned according to the result of this operation. The contents of the accumulator are not changed.

In emulation mode, `tax` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
`i`	1	AA

tay — transfer accumulator to Y register — 6502, 65C02, 65C816

Deposits the value in the accumulator into the Y register. The n and z flags are conditioned according to the result of this operation. The contents of the accumulator are not changed.

In emulation mode, `tay` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: n, z

Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
i	1	A8

tcd — transfer 16-bit accumulator to direct page register — 65C816

Transfers the value in the 16-bit accumulator (C) to the direct page register (D). The value of C is not changed.

When the `tcd` instruction is issued, both bytes in the 16-bit accumulator are copied into the direct page register, regardless of the setting of the m flag. If the accumulator is in 8-bit mode, the low-order byte of the 16-bit accumulator (A) is transferred to the low byte of the direct page register, and the value in the accumulator's "hidden" high-order byte (B) is transferred to the high byte of the direct page register.

Flags affected: n, z

Registers affected: D, P

Addressing Mode	Bytes	Opcode (hex)
i	1	5B

tcs — transfer accumulator to stack pointer — 65C816

Transfers the value in the accumulator to the stack pointer. The accumulator's value is unchanged.

If the 65C816 is in native mode, `tcs` transfers both bytes in the 16-bit accumulator (C) to the stack pointer, regardless of the setting of the m flag. The accumulator's low-order byte (A) is transferred to the low byte of the stack pointer, and the value in the accumulator's "hidden" high-order byte (B) is transferred to the high byte of the stack pointer. If the 65C816 is in emulation mode, only the 8-bit accumulator (A) is transferred.

The `tcs` and `txs` (transfer the X register to the stack pointer) instructions are the only instructions for changing the value in the stack pointer. They are also the only two transfer instructions that do not alter the n and z flags.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
i	1	1B

tdc — transfer direct page register to 16-bit accumulator — 65C816

Transfers the value of the direct page register (D) to the 16-bit accumulator (C). The value of the D register is not changed.

The `tdc` instruction transfers 16 bytes, regardless of the setting of the m (accumulator/memory mode) flag. If the accumulator is in 8-bit mode, the accumulator's low-order byte (A) takes the value of the low byte of the direct page register, and the accumulator's "hidden" B register takes the value of the high byte of the direct page register.

Flags affected: n, z

Registers affected: A, B, C, P

Addressing Mode	Bytes	Opcode (hex)
i	1	7B

trb — test and reset memory bits against accumulator — 65C02, 65C816

Logically ANDs the value in the accumulator with the complement of the value in a memory location. This operation clears each memory bit that corresponds to a set bit in the accumulator, while leaving unchanged each memory bit that corresponds to a cleared bit in the accumulator. The result of the operation is stored in the memory location.

In addition, the P register's z flag is conditioned by the result of the AND operation. It sets the z flag if the result of the operation is zero and clears it if the result is not zero. This is the same way that the `bit` instruction conditions the zero flag. But `trb`, unlike `bit`, is a read-modify-write instruction. It not only calculates a result and modifies a flag, but also stores the result in memory.

In emulation mode, `trb` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: z

Registers affected: M, P

Addressing Mode	Bytes	Opcode (hex)
`trb d`	2	14
`trb a`	3	1C

tsb — test and set memory bits against accumulator — 65C02, 65C816

Logically ORs the value in the accumulator with the value stored in a memory location. This operation sets each memory bit that corresponds to a set bit in the accumulator, while leaving unchanged each memory bit that corresponds to a cleared bit in the accumulator. The result of the operation is stored in the memory location.

In addition, the P register's z flag is conditioned by the result of the OR operation. It sets the z flag if the result of the operation is zero and clears it if the result is not zero. This is the same way that the `bit` instruction conditions the zero flag. But `tsb`, unlike `bit`, is a read-modify-write instruction. It not only calculates a result and modifies a flag, but also stores the result in memory.

In emulation mode, `tsb` is an 8-bit operation. In native mode, it is a 16-bit operation, with the high-order byte situated in the effective address plus 1.

Flags affected: z

Registers affected: M, P

Addressing Mode	Bytes	Opcode (hex)
`tsb d`	2	04
`tsb a`	3	0C

tsc — transfer stack pointer to 16-bit accumulator — 65C816

Transfers the value in the stack pointer (S) to the accumulator. The stack pointer's value is unchanged.

If the 65C816 is in native mode, `tsc` transfers both bytes in the stack pointer to the 16-bit accumulator (C), regardless of the setting of the m flag. The accumulator's low-order byte (A) takes the value of the low byte of the stack pointer, and the value in the accumulator's "hidden" high-order byte (B) takes the value of the high byte of the stack pointer. If the 65C816 is in emulation mode, B always takes a value of 1 because the stack is always page 1 in 8-bit emulation mode.

Flags affected: None

Registers affected: A, B, C

Addressing Mode	Bytes	Opcode (hex)
i	1	3B

tsx — transfer stack to X register — 6502, 65C02, 65C816

Deposits the value in the stack pointer into the X register. The n and c flags are conditioned according to the result of this operation. The value of the stack pointer is not changed.

When the 65C816 is in emulation mode, `tsx` is an 8-bit operation. If the 65C816 is in native mode and the X register is in 16-bit mode, `tsx` is a 16-bit operation. If the 65C816 is in native mode and the X register is in 8-bit mode, only the low-order byte of the stack pointer is transferred to the X register.

Flags affected: n, c

Registers affected: X, P

Addressing Mode	Bytes	Opcode (hex)
i	1	BA

txa — transfer X register to accumulator — 6502, 65C02, 65C816

Deposits the value in the X register into the accumulator. The n and z flags are conditioned according to the result of this operation. The value of the X register is not changed.

If the 65C816 is in native mode and the A and X registers are both in 16-bit mode, both bytes of the X register are transferred to the accumulator.

If the 65C816 is in emulation mode and the A and X registers are both in 8-bit mode, the 8-bit X register is transferred to the 8-bit accumulator.

If the 65C816 is in native mode and the accumulator is in 8-bit mode and the X register is in 16-bit mode, the low byte of the X register is moved into the accumulator's low byte (A) and the accumulator's high byte (the "hidden" register B) is not affected by the transfer.

If the 65C816 is in native mode and the accumulator is in 16-bit mode and the X register is in 8-bit mode, the X register is moved into the accumulator's low byte (A) and the accumulator's high byte (B) takes a value of 0.

Flags affected: n, z

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
i	1	8A

txs — transfer stack to X register — 6502, 65C02, 65C816

Deposits the value in the X register into the stack pointer. No flags are conditioned by this operation. The value of the X register is not changed.

When the 65C816 is in emulation mode, `txs` is an 8-bit operation. If the 65C816 is in native mode and the X register is in 16-bit mode, `txs` is a 16-bit operation. If the 65C816 is in native mode and the X register is in 8-bit mode, the X register is transferred to the low byte of the stack pointer and the high byte of the stack pointer is zeroed.

Flags affected: None

Registers affected: S

Addressing Mode	Bytes	Opcode (hex)
i	1	9A

txy — transfer X register to Y register — 6502, 65C02, 65C816

Transfers the value of the X register to the Y register. The value of the X register is not changed.

When the 65C816 is in emulation mode, `txy` is an 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in native mode, `txy` is a 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in 16-bit mode, `txy` is a 16-bit operation.

Flags affected: n, z

Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
i	1	9B

tya — transfer Y register to accumulator — 6502, 65C02, 65C816

Deposits the value in the Y register into the accumulator. The n and z flags are conditioned according to the result of this operation. The value of the Y register is not changed.

If the 65C816 is in native mode and the A and Y registers are both in 16-bit mode, both bytes of the Y register are transferred to the accumulator.

If the 65C816 is in emulation mode and the A and X registers are both in 8-bit mode, the 8-bit Y register is transferred to the 8-bit accumulator.

If the 65C816 is in native mode and the accumulator is in 8-bit mode and the Y register is in 16-bit mode, the low byte of the Y register is moved into the accumulator's low byte (A) and the accumulator's high byte (the "hidden" register B) is not affected by the transfer.

If the 65C816 is in native mode and the accumulator is in 16-bit mode and the Y register is in 8-bit mode, the Y register is moved into the accumulator's low byte (A) and the accumulator's high byte (B) takes a value of 0.

Flags affected: n, z

Registers affected: A, P

Addressing Mode	Bytes	Opcode (hex)
i	1	98

tyx — transfer Y register to X register — 6502, 65C02, 65C816

Transfers the value of the Y register to the X register. The value of the Y register is not changed.

When the 65C816 is in emulation mode, `tyx` is an 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in native mode, `tyx` is an 8-bit operation. When the 65C816 is in native mode and the X and Y registers are in 16-bit mode, `tyx` is an 16-bit operation.

Flags affected: n, z

Registers affected: Y, P

Addressing Mode	Bytes	Opcode (hex)
i	1	BB

wai — wait for interrupt — 65C816

The `wai` instruction puts the 65C816 in a dormant condition during an external event to reduce its power consumption or to provide an immediate response to interrupts so that the proccessor can be synchronized with the external event.

After an interrupt is received, control is generally vectored through one of the hardware interrupt vectors, and an `rti` instruction in an interrupt handling routine returns control to the instruction following the `wai` instruction. But if interrupts are disabled by setting the P register's i flag and a hardware interrupt takes place, the 65C816's wait condition is terminated and control resumes with the next instruction, rather than through the interrupt

vectors. This system provides a very fast response to an interrupt, allowing synchronization with external events.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	1	CB

wdm — reserved for future expansion — 65C816

The letters *wdm* are the initials of William D. Mensch, Jr., the designer of the 65C02 and the 65C816. The `wdm` instruction uses opcode $42, the only one of the 65C816's 256 possible machine language opcodes that is not used. It is left unused so that it can be a gateway to any new assembly language instructions that may be added to the 65C816's instruction set. If new instructions are added, they have to take 2-byte opcodes, and the `wdm` instruction will signify that the next byte is an opcode in the processor's expanded instruction set.

If the `wdm` instruction is used in a IIGS program, it has no effect except to consume time. It behaves like a 2-byte `nop` instruction. But you should not use `wdm` in a program because it would make the program incompatible with any future 65C02 family chips.

Flags affected: None

Registers affected: None

Addressing Mode	Bytes	Opcode (hex)
i	2†	42

†Subject to change in future processors.

xba — swap the B and A accumulators

Swaps the contents of the 8-bit A register (the low-order byte of the 16-bit accumulator C) with the contents of the 8-bit B register (the high-order byte of the 16-bit accumulator C). When the 65C816 is in emulation mode, this is the only way to access the accumulator's "hidden" B register. The transfer conditions the P register's n and z flags.

The `xba` instruction can be used to invert the low-order, high-order arrangement of a 16-bit value or to store an 8-bit value in the B register. Because it is an exchange, the previous contents of both accumulators are changed, replaced by the previous contents of the other.

Neither the m (mode select) flag nor the e (emulation mode) flag affects this operation.

Flags affected: n, z

Registers affected: A, B, C, P

Addressing Mode	Bytes	Opcode (hex)
i	1	EB

xce — exchange carry and emulation bits — 65C816

Swaps the P register's carry flag with the e (emulation mode) flag. The `xce` instruction is the only method for toggling the 65C816 between 16-bit native mode and 8-bit emulation mode.

If the processor is in emulation mode, it can be switched to native mode by clearing the carry bit and then executing the `xce` instruction. If the processor is in native mode, it can be switched to emulation mode by setting the carry bit and then executing the `xce` instruction.

Flags affected: c, e

Registers affected: P

Addressing Mode	Bytes	Opcode (hex)
i	1	FB

APPENDIX B

Apple IIGS Toolbox Calls

This appendix contains most of the calls in the Apple IIGS Toolbox. The calls are listed alphabetically.

Tool Abbreviations

Abbreviation	Meaning
ADB	Apple Desktop Bus
CM	Control Manager
DM	Desk Manager
DLM	Dialog Manager
EM	Event Manager
FM	Font Manager
IM	Integer Math Tool Set
LE	LineEdit Tool Set
LM	List Manager
MM	Memory Manager
MUM	Menu Manager
MTS	Miscellaneous Tool Set
PM	Print Manager

Tool Abbreviations (cont.)

Abbreviation	Meaning
QD	QuickDraw II
SAN	SANE Tool Set
SK	Scheduler
ST	Sound Tool Set
SF	Standard File Operations Tool Set
TT	Text Tool Set
WM	Window Manager

Toolbox Calls

Call	Tool	Call Number	Function
AbsOff	ADB	$1009	Disables automatic polling of an absolute device.
AbsOn	ADB	$0F09	Enables automatic polling from an absolute device.
AddFamily	FM	$0D1B	Allows a family to be added to the Font Manager's list of font families.
AddFontVar	FM	$141B	Allows a pre-existing family to be added to the available font list.
AddPt	QD	$8004	Adds two points and leaves their sum in the destination point.
Alert	DLM	$1715	Invokes an alert defined by a specified alert template.
ASynchADBReceive	ADB	$0D09	Receives data from a ADB device.
AutoAbsPoll	ADB	$1109	Reads flags to determine if automatic polling is on or off.
BeginUpdate	WM	$1E0E	Starts the window drawing procedure when a window is updated.
BlockMove	MM	$2B02	Copies a specified number of bytes from a source to a destination.
BringToFront	WM	$240E	Brings a window to the front and redraws other windows as necessary.
Button	EM	$0D06	Returns the current state of the specified mouse button.
CalcMenuSize	MUM	$1C0F	Sets menu dimensions, either manually or automatically.

Call	Tool	Call Number	Function
CautionAlert	DLM	$1A15	Performs functions similar to those of the Alert routine.
CharBounds	QD	$AC04	Sets a specified rectangle to be the bounds of a specified character.
CharWidth	QD	$A804	Returns the width in pixels of a specified character.
CheckHandle	MM	$1E02	Checks a handle to see if it's valid.
CheckMItem	MUM	$320F	Displays or removes a check mark to the left of a menu item.
CheckUpdate	WM	$0A0E	Checks to see if any windows need updating.
ChooseCDA	DM	$1105	Activates the Desk Manager and displays the CDA menu.
ChooserFont	FM	$161B	Displays a dialog for selection of a new font, size, and/or style.
CIrHeartBeat	MTS	$1403	Removes all tasks from the heartbeat interrupt task queue.
ClampMouse	MTS	$1C03	Sets mouse clamp values and places the mouse at the minimum values.
ClearMouse	MTS	$1B03	Sets the mouse's X and Y axis positions to $0000 or clamp minimums.
ClearScreen	QD	$1504	Sets the words in screen memory to a specified value.
ClearSRQTable	ADB	$1609	Clears the SRQ list of all entries.
ClipRect	QD	$2604	Makes the current port's clip rectangle equal to a given rectangle.
CloseAIINDAs	DM	$1D05	Closes all open NDAs.
CloseDialog	DLM	$0C15	Removes a dialog from the screen and deletes it from the window list.
CloseNDA	DM	$1605	Closes a specified new desk accessory.
CloseNDAbyWinPtr	DM	$1C05	Closes the NDA whose window pointer is passed.
ClosePoly	QD	$C204	Completes the polygon creation started with OpenPoly.
ClosePort	QD	$1A04	Deallocates the regions in a port.

Call	Tool	Call Number	Function
CloseRgn	QD	$6E04	Stops processing of a region and returns the created region.
CloseWindow	WM	$0V0E	Removes a window from the screen and deletes it from the window list.
CompactMem	MM	$1F02	Compacts memory.
CopyRgn	QD	$6904	Copies the contents of a region from one region to another.
CountFamilies	FM	$091B	Returns the number of font families available.
CountFonts	FM	$101B	Returns the number of fonts available that fit a certain description.
CountMItems	MUM	$140F	Returns the number of items in a specified menu.
CreateList	LM	$091C	Creates a list control and returns its handle.
CStringBounds	QD	4AE04	Sets a specified rectangle to be the bounds of a specified C string.
CStringWidth	QD	$AA04	Returns the width of a specified C string.
CtlBootInit	CM	$0110	Called only by the Tool Locator when the system is booted.
CtlNewRes	CM	$1210	Reinitializes resolution and mode.
CtlReset	CM	$0510	Called on system reset.
CtlShutDown	CM	$0310	Deactivates the Control Manager.
CtlStartUp	CM	$0210	Starts up the Control Manager for use by an application.
CtlStatus	CM	$0610	Checks the current status of the Control Manager.
CtlTextDev	TT	$160C	Passes a control code to a specified text device.
CtlVersion	CM	$0410	Returns the version number of the Control Manager.
Dec2Int	IM	$280B	Converts an ASCII string into a 16-bit signed or unsigned integer.
Dec2Long	IM	$290B	Converts an ASCII string into a 32-bit integer.
DefaultFilter	DLM	$3615	Calls a modal or an alert dialog's standard default filter.

Call	Tool	Call Number	Function
DeleteID	MTS	$2103	Deletes all references to a specified user ID.
DeleteMenu	MUM	$0E0F	Removes a specified menu from the menu list.
DeleteMItem	MUM	$100F	Removes a specified item from the current menu.
DelHeartBeat	MTS	$1303	Deletes a specified task from the heartbeat interrupt task queue.
DeskBootInit	DM	$0105	Internal routine called at boot time to initialize the Desk Manager.
DeskReset	DM	$0505	Resets the Desk Manager.
DeskShutDown	DM	$0305	Shuts down the Desk Manager.
DeskStartUp	CM	$0205	Starts up the Desk Manager.
DeskStatus	DM	$0605	Tells if the Desk Manager is active.
Desktop	WM	$0C0E	Keeps track of regions on the desktop and controls desktop pattern.
DeskVersion	DM	$0405	Returns the version number of the Desk Manager.
DialogBootInit	DLM	$0115	Called by the Tool Locator at initialization.
DialogReset	DLM	$0515	Resets the Dialog Manager.
DialogSelect	DLM	$1115	Handles modeless dialog events.
DialogShutDown	DLM	$0315	Shuts down the Dialog Manager.
DialogStartUp	DLM	$0215	Starts up the Dialog Manager.
DialogStatus	DLM	$0615	Indicates if the Dialog Manager is active.
DialogVersion	DLM	$0415	Returns the version number of the Dialog Manager.
DiffRgn	QD	47304	Returns a region that is the difference between two regions.
Disable Increment	ST		Disables auto-increment mode.
DisableDItem	DLM	$3915	Disables a specified item in a specified dialog.
DisableMItem	MUM	$310F	Displays an item in dimmed characters and makes it unselectable.
DisposeAll	MM	$1102	Discards all the handles belonging to a specified user ID.

Call	Tool	Call Number	Function
DisposeControl	CM	$0A10	Deletes a specified control from its window's control list.
DisposeHandle	MM	$1002	Disposes of a specified block and deallocates its handle.
DisposeMenu	MUM	$2E0F	Frees the memory allocated by NewMenu.
DisposeRgn	QD	$6804	Deallocates space for a specified region.
DlgCopy	DLM	$1315	Applies the LineEdit procedure LECopy to an EditLine item.
DlgCut	DLM	$1215	Applies the LineEdit procedure LECut to an EditLine item.
DlgDelete	DLM	$1515	Applies the LineEdit procedure LEDelete to an EditLine item.
DlgPaste	DLM	$1415	Applies the LineEdit procedure LEPaste to an EditLine item.
DoWindows	EM	$0906	Returns the address of the Event Manager's direct page work area.
DragControl	CM	$1710	Pulls a dotted outline of a control around the screen.
DragRect	CM	$1D10	Pulls a dotted outline of a rectangle around the screen.
DragWindow	WM	$1A0E	Pulls around the outline of a window, following mouse movements.
DrawChar	QD	$A404	Draws a specified character at the current pen location.
DrawControls	CM	$1010	Draws all controls currently visible in a specified window.
DrawCString	QD	$A604	Draws a specified C string at the current pen location.
DrawDialog	DLM	$1615	Draws the contents of a specified dialog box.
DrawIcon	QD	$0B12	Draws an icon on the screen.
DrawMember	LM	$0C1C	Redraws a member of the list whose state may have changed.
DrawMenuBar	MUM	$2A0F	Draws the current menu bar, along with any menu titles on the bar.
DrawOneCtl	CM	$2510	Draws a specified control.

Call	Tool	Call Number	Function
DrawString	QD	$A504	Draws a specified string at the current pen location.
DrawText	QD	$A704	Draws specified text at the current pen location.
EMBootInit	EM	$0106	Called at boot time by the Tool Locator.
EmptyRect	QD	$5204	Returns whether or not a specified rectangle is empty.
EmptyRgn	QD	$7804	Checks to see if a specified region is empty.
EMReset	EM	$0506	Returns an error if the Event Manager is active.
EMShutDown	EM	$0306	Shuts down the Event Manager and releases any workspace allocated to it.
EMStartUp	EM	$0206	Initializes the Event Manager and sets the size of the event queue.
EMStatus	EM	$0606	Indicates a nonzero value if the Event Manager is active.
EMVersion	EM	$0406	Returns the version of the Event Manager.
EnableDItem	DLM	$3A15	Enables a specified item in a specified dialog.
EnableMItem	MUM	$300F	Displays an item normally and allows it to be selected.
EndInfoDrawing	WM	$510E	Puts the Window Manager back into a global coordinate system.
EndUpdate	WM	$1F0E	Ends the window drawing procedure started by BeginUpdate.
EqualPt	QD	$8304	Indicates whether two points are equal.
EqualRect	QD	$5104	Compares two rectangles and indicates if they are equal.
EqualRgn	QD	$7704	Compares two regions and tells if they are equal.
EraseArc	QD	$6404	Erases an arc by filling it with the background pattern.
EraseControl	CM	$2410	Makes a specified control invisible.
EraseOval	QD	$5A04	Erases an oval by filling it with the background pattern.
ErasePoly	QD	$BE04	Erases a specified polygon.

Call	Tool	Call Number	Function
EraseRect	QD	$5504	Erases a rectangle by filling it with the background pattern.
EraseRgn	QD	$7B04	Fills the interior of a specified region with the background pattern.
EraseRRect	QD	45F04	Erases the interior of a round rectangle.
ErrorSound	DLM	$0915	Sets the sound procedure for alerts to a specified procedure.
ErrWriteBlock	TT	$1F0C	Writes a block of text to the error output text device.
ErrWriteChar	TT	$190C	Writes a character to the error output text device.
ErrWriteCString	TT	$210C	Writes a C-style string to the error output text device.
ErrWriteLine	TT	$1B0C	Writes a string, plus a carriage return, to the error output text device.
ErrWriteString	TT	$1D0C	Writes a string to the error output text device.
EventAvail	EM	$0B06	Accesses the next available event but leaves it in the queue.
FakeMouse	EM	$1906	Allows an application to use an alternative pointing device.
FamNum2ItemID	FM	$171B	Translates a font family number into a menu item ID.
FamNum2ItemID	FM	$1B1B	Tells if a menu item is displayed in a specified font family.
FFGeneratorStatus	ST	$1108	Reads the first 2 bytes of a block corresponding to a generator.
FFSoundDoneStatus	ST	$1408	Returns the free-form synthesizer sound-playing status.
FFSoundStatus	ST	$1008	Returns the status of all fifteen generators.
FFStartSound	ST	$0E08	Enables the DOC to start generating sound on a particular generator.
FFStopSound	ST	$0F08	Stops sound generators that may be running.
FillArc	QD	$6604	Fills the interior of an arc.
FillOval	QD	$5C04	Fills an oval with a specified pattern.

Call	Tool	Call Number	Function
FillPoly	QD	$C004	Fills a specified polygon with a specified pen pattern.
FillRect	QD	$5704	Fills the interior of a specified rectangle with a specified pattern.
FillRgn	QD	$7D04	Fills the interior of a specified region with a specified pattern.
FillRRect	QD	$6104	Fills a round rectangle with a specified pattern.
FindControl	CM	$1310	Tells in which control the mouse button was pressed.
FindDItem	DLM	$2415	Returns the ID of the item located at a specified point in a dialog.
FindFamily	FM	$0A1B	Returns the family number and name of a particular font family.
FindFontStats	FM	$111B	Places a FontID and a FontStatBits in a specified FontStat record.
FindHandle	MM	$1A02	Returns the handle of the block containing a specified address.
FindWindow	WM	$170E	Tells if the mouse was clicked inside a window, and where.
Fix2Frac	IM	$1C0B	Converts fixed to fraction.
Fix2Long	IM	$1B0B	Converts fixed to long integer.
Fix2X	IM	$1E0B	Converts fixed to extended.
FixAppleMenu	DM	$1E05	Adds the names of new desk accessories to the specified menu.
FixATan2	IM	$170B	Returns a fixed arc tangent of the coordinates of two like inputs.
FixDiv	IM	$110b	Divides two like inputs and returns a rounded fixed result.
FixFontMenu	FM	$151B	Appends the names of available font families onto a specified menu.
FixMenuBar	MUM	$130F	Computes standard sizes for the menu bar and menus.
FixMul	IM	$0F0B	Multiplies two 32-bit fixed inputs and returns a 32-bit fixed result.
FixRatio	IM	$0E0B	Returns a 32-bit fixed-number ratio of a numerator and a denominator.

Call	Tool	Call Number	Function
FixRound	IM	$130B	Takes a fixed input and returns a rounded integer result.
FlashMenuBar	MUM	$0C0F	Flashes the current menu bar using colors set by NewInvertColor.
FlushEvents	EM	$1506	Removes specified queue events until a stop mask is encountered.
FMBootInit	FM	$011B	Called at boot time by the Tool Locator.
FMGetCurFID	FM	$1A1B	Returns the FontID of the current font.
FMGetSysFID	FM	$191B	Returns the FontID of the system font.
FMReset	FM	$051B	Returns an error if the Font Manager is active.
FMSetSysFont	FM	$181B	Loads a specified font into memory, makes it unpurgeable.
FMShutDown	FM	$031B	Shuts down the Font Manager.
FMStartUp	FM	$021B	Initializes the Font Manager for use by an application.
FMStatus	FM	$061B	Returns a nonzero value if the Font Manager is active.
FMVersion	FM	$041B	Returns the version number of the Font Manager.
ForceBufDims	QD	$CC04	Works like SetBufDims, but does not pad MaxFBRExtent.
Frac2Fix	IM	$1D0B	Converts fraction to fixed.
Frac2X	IM	$1F0B	Converts fraction to extended.
FracCos	IM	$150B	Takes a fixed input and returns its fractional cosine.
FracDiv	IM	$120B	Divides two like inputs and returns a rounded fractional result.
FracMul	IM	$100B	Multiplies two fractional inputs and returns a rounded fractional result.
FracSart	IM	$140B	Takes a fractional input and returns a rounded fractional square root.
FracSin	IM	$160B	Takes a fixed input and returns its fractional sine.

Call	Tool	Call Number	Function
FrameArc	QD	$6204	Draws the boundary of an arc using the current pen state and pattern.
FrameOval	QD	$5804	Frames an oval using the current pen state and pen pattern.
FramePoly	QD	$BC04	Frames a specified polygon.
FrameRect	QD	$5304	Frames a rectangle using the current pen state and pen pattern.
FrameRgn	QD	$7904	Frames a specified region using the current pen state and pattern.
FrameRRect	QD	$5D04	Frames a round rectangle using the current pen state and pen pattern.
FreeMem	MM	$1B02	Returns the total number of free bytes in memory.
FrontWindow	WM	$150E	Returns a pointer to the first visible window in the window list.
FWEntry	MTS	$2403	Allows some Apple II entry points to be supported from native mode.
GetAbsClamp	MTS	$2B03	Returns the current values for the absolute device clamps.
GetAbsScale	ADB	$1309	Reads absolute device scaling values.
GetAddr	MTS	$1603	Returns the address of a parameter referenced by the firmware.
GetAddress	QD	$0904	Returns a pointer to a specified table.
GetAlertStage	DLM	$3415	Returns the stage of the last occurrence of an alert.
GetBackColor	QD	$A304	Returns the value of the background color field from the GrafPort.
GetBackPat	QD	$3504	Returns the current background pattern.
GetBarColors	MUM	$180F	Returns the colors for the current menu bar.
GetCaretTime	EM	$1206	Returns the time between blinks of the caret.
GetCharExtra	QD	$D504	Returns the chExtra field from the GrafPort.

Call	Tool	Call Number	Function
GetClip	QD	$2504	Copies the ClipRgn to a specified region.
GetClipHandle	QD	$C704	Returns a copy of the handle to the ClipRgn.
GetColorEntry	QD	$1104	Returns the value of a color in a specified color table.
GetColorTable	QD	$0F04	Fills a color table with the contents of another color table.
GetContentDraw	WM	$480E	Returns a pointer to the routine that draws a window's contents.
GetContentOrgin	WM	$3E0E	Returns values used to set the origin of a window's port.
GetContentRgn	WM	$2F0E	Returns a handle to a specified window's content region.
GetControlDItem	DLM	$1E15	Returns a handle to the control for a specified item.
GetCtlAction	CM	$2110	Returns the current value of a specified control's CtLAction field.
GetCtlDpage	CM	$1F10	Returns the value of the Control Manager's direct page.
GetCtlParams	CM	$1C10	Returns a specified control's additional parameter settings.
GetCtlRefCon	CM	$2310	Returns the current value of a specified control's CtlRefCon field.
GetCtlTitle	CM	$0D10	Returns the value in a specified control's CtlData field.
GetCtlValue	CM	$1A10	Returns a specified control's current CtlValue field.
GetCursorAdr	QD	$8F04	Returns a pointer to the current cursor record.
GetDAStrPtr	DM	$1405	Returns the pointer to a table of desk accessory strings.
GetDataSize	WM	$400E	Returns the height and width of the data area of a specified window.
GetDbITime	EM	$1106	Sets the time required between mouse clicks for a double click.
GetDefButton	DLM	$3715	Returns the ID of the default button item in a specified dialog.
GetDefProc	WM	$310E	Returns the address of a routine that controls a window's behavior.

Call	Tool	Call Number	Function
GetDItemBox	DLM	$2815	Returns the display rectangle of a specified item.
GetDItemType	DLM	$2615	Returns the type of a specified item.
GetDItemValue	DLM	$2E15	Returns the current value of a specified item.
GetErrGlobals	TT	$0E0C	Returns the current values for the error output device's global parameters.
GetErrorDevice	TT	$140C	Returns the type of driver installed as the error output device.
GetFamInfo	FM	$0B1B	Returns the name of a font family that has a specified family number.
GetFamNum	FM	$0C1B	Returns a family number corresponding to a given font family name.
GetFGSize	QD	$CF04	Returns the size of the font globals record.
GetFirstDItem	DLM	$2A15	Returns the ID of the first item in a specified dialog.
GetFirstWindow	WM	$520E	Returns the first window in the Window Manager's window list.
GetFont	QD	$9504	Returns a handle to the current font.
GetFontFlags	QD	$9904	Returns the current font flags.
GetFontGlobals	QD	$9704	Returns information about the current font in the specified record.
GetFontID	QD	$D104	Returns the FontID in the GrafPort.
GetFontInfo	QD	$9604	Returns information about the current font in the specified record.
GetFontLore	QD	$D904	Returns information about the current font in the specified record.
GetForeColor	QD	$A104	Returns the current foreground color from the GrafPort.
GetFrameColor	WM	$100E	Returns the color of a specified window's frame.
GetFuncPtr	TL	$0B01	Returns a pointer, less 1, to a specified tool function.

Call	Tool	Call Number	Function
GetGrafProcs	QD	$4504	Returns a pointer to the current port's GrafProcs record.
GetHandleSize	MM	$1802	Returns the size of a specified block.
GetInfoDraw	WM	$4A0E	Returns a pointer to a window's information bar drawing procedure.
GetInfoRefCon	WM	$350E	Returns a value associated with an information bar drawing routine.
GetInputDevice	TT	$120C	Returns the type of driver installed as the input device.
GetIRQEnable	MTS	$2903	Returns the interrupt enable states of certain interrupt sources.
GetIText	DLM	$1F15	Returns the text of a specified StatText or EditLine item.
GetListDefProc	LM	$0E1C	Returns a pointer to the list control's definition procedure.
GetInGlobals	TT	$0C0C	Returns current values for the input device's global parameters.
GetMasterSCB	QD	$1704	Returns a copy of the master SCB.
GetMaxGrow	WM	$420E	Returns the maximum values to which a window's content can grow.
GetMenuBar	MUM	$0A0F	Returns the handle of the current menu bar.
GetMenuFlag	MUM	$200F	Returns the menu flag for a specified menu.
GetMenuMgrPort	MUM	$1B0F	Returns a pointer to the Menu Manager's port.
GetMenuTitle	MUM	$220F	Returns a pointer to the title of a menu.
GetMHandle	MUM	$160F	Returns a handle to a menu record.
GetMItem	MUM	$250F	Returns a pointer to the name of an item.
GetMItemFlag	MUM	$270F	Returns information about a specified menu item.
GetMItemMark	MUM	$340F	Returns the character displayed to the left of a specified item.

Call	Tool	Call Number	Function
GetMItemStyle	MUM	$360F	Returns the text style for a specified menu item.
GetMouse	EM	$0C06	Returns the current mouse location.
GetMouseClamp	MTS	$1D03	Returns the current mouse clamp values.
GetMTitleStart	MUM	$1A0F	Returns the starting position for the leftmost title within the current menu bar.
GetMTitleWidth	MUM	$1E0F	Returns the width of a menu title.
GetNewDItem	DLM	$3315	Adds a new item to a specified dialog's item list using a template.
GetNewID	MTS	$2003	Creates a new user ID.
GetNewModalDialog	DLM	$3215	Creates a modal dialog and returns a pointer to its GrafPort.
GetNextDItem	DLM	$2B15	Returns the ID of the next item in a specified dialog.
GetNextEvent	EM	$0A06	Returns the next available event of a specified type or types.
GetNextWindow	WM	$2A0E	Returns a pointer to the next window in the window list.
GetNumNDAs	DM	$1B05	Returns the total number of new desk accessories currently installed.
GetOSEvent	EM	$1606	Removes a specified type of event from the queue.
GetOutGlobals	TT	$0D0C	Returns current values for the output device's global parameters.
GetOutputDevice	TT	$130C	Returns the type of driver installed as the output device.
GetPage	WM	$460E	Returns the number of pixels to be scrolled by a scroll bar's "page" region.
GetPen	QD	$2904	Returns the pen location.
GetPenMask	QD	$3304	Returns the pen mask at the specified location.
GetPenMode	QD	$2F04	Returns the pen mode from the current port.
GetPenPat	QD	$3104	Copies the current port's pen pattern into a specified location.

Call	Tool	Call Number	Function
GetPenSize	QD	$2D04	Returns the current pen size at the place indicated.
GetPenState	QD	$2B04	Returns the pen state from the GrafPort.
GetPicSave	QD	$3F04	Returns the contents of the PicSave field in the GrafPort.
GetPixel	QD	$8804	Returns the pixel below and to the right of a specified point.
GetPolySave	QD	$4304	Returns the contents of the PicSave field in the GrafPort.
GetPort	QD	$1C04	Returns a pointer to the current port.
GetPortLoc	QD	$1E04	Returns the current port's map information structure.
GetPortRect	QD	$2004	Returns the current port's port rectangle.
GetRectInfo	WM	$4F0E	Sets a rectangle in which objects can be drawn in an information bar.
GetRgnSave	QD	$4104	Returns the contents of the RgnSave field in the GrafPort.
GetRomFont	QD	$D804	Fills a specified record with information about the font in ROM.
GetSCB	QD	$1304	Returns the value of a specified scan-line control byte (SCB).
GetScrap	SK	$0D16	Copies scrap information to the specified handle.
GetScrapCount	SK	$1216	Returns the current scrap count.
GetScrapHandle	SK	$0E16	Returns a copy of the handle for the scrap of the specified type.
GetScrapPath	SK	$1016	Returns a pointer to the pathname used for the clipboard file.
GetScrapSize	SK	$0F16	Returns the size of the specified scrap.
GetScrapState	SK	$1316	Returns a flag indicating the current state of the scrap.
GetScroll	WM	$440E	Returns the number of pixels scrolled by the arrows in a scroll bar.
GetSoundVolume	ST	$0C08	Reads the volume setting for a generator.

Call	Tool	Call Number	Function
GetSpaceExtra	QD	$9F04	Returns the value of the SpExtra field from the GrafPort.
GetStandardSCB	QD	$0C04	Returns a copy of the standard SCB in the low-order byte of the word.
GetStructRgn	WM	$2E0E	Returns a handle to a specified window's structure region.
GetSysBar	MUM	$110F	Returns the handle of the current system menu bar.
GetSysField	QD	$4904	Returns the contents of the SysField in the GrafPort.
GetSysFont	QD	$B304	Returns a handle to the current system font.
GetTableAddress	ST	$0B08	Returns the jump table address for low-level routines.
GetTextFace	QD	$9B04	Returns the current text face.
GetTextMode	QD	$9D04	Returns the current text mode.
GetTextSize	QD	$D304	Not yet implemented at the time of this writing.
GetTick	MTS	$2503	Returns the current value of the tick counter.
GetTSPtr	TL	$0901	Returns a pointer to the function pointer table of a specified tool set.
GettSysWFlag	WM	$4C0E	Indicates if a specified window is a system window.
GetUpdateRgn	WM	$300E	Returns a handle to a specified window's update region.
GetUserField	QD	$4704	Returns the contents of the UserField field in the GrafPort.
GetVector	MTS	$1103	Returns the vector address for a specified vector reference number.
GetVisHandle	QD	$C904	Returns a copy of the handle to the VisRgn.
GetVisRgn	QD	$B504	Copies the contents of the VisRgn into a specified region.
GetWAP	TL	$0C01	Gets the pointer to the work area for a specified tool set.
GetWControls	WM	$330E	Returns the handle of the first control in the window's control list.

Call	Tool	Call Number	Function
GetWFrame	WM	$2C0E	Returns a bit array that describes a window's frame type.
GetWKind	WM	$2B0E	Tells if a window is a system or application window.
GetWMgrPort	WM	$200E	Returns a pointer to the Window Manager's port.
GetWRefCon	WM	$290E	Returns a value passed to NewWindow or WRefCon by an application.
GetWTitle	WM	$0E0E	Returns a pointer to a specified window's title.
GetZoomRect	WM	$370E	Returns a pointer to a rectangle representing a window's zoomed size.
GlobalToLocal	QD	$8504	Converts a point from global coordinates to local coordinates.
GrafOff	QD	$0B04	Turns off super high-resolution graphics mode.
GrafOn	QD	$0A04	Turns on super high-resolution graphics mode.
GrowSize	CM	$1E10	Returns the height and width of the grow box control.
GrowWindow	WM	$1B0E	Expands or shrinks a window, corresponding to mouse movements.
HandToHand	MM	$2A02	Copies a block of bytes from a source handle to a destination handle.
HandToPtr	MM	$2902	Copies a block of bytes from a handle to a pointer.
Hex2Int	IM	$240B	Converts a hexadecimal string into a 16-bit unsigned integer.
Hex2Long	IM	$250B	Converts a hexadecimal string into a 32-bit unsigned integer.
Hexlt	IM	$2A0B	Converts a 16-bit unsigned integer into a hexadecimal string.
HideControl	CM	$0E10	Makes a specified control invisible.
HideCursor	QD	$9004	Hides the cursor, that is, decrements the cursor level.
HideDItem	DLM	$2215	Erases a specified item from a specified dialog.
HidePen	QD	$2704	Decrements the pen level.
HideWindow	WM	$120E	Makes a specified window invisible.

Call	Tool	Call Number	Function
HiLiteControl	CM	$1110	Changes the way a specified control is highlighted.
HiliteMenu	MUM	$2C0F	Highlights or unhighlights the title of a specified menu.
HiliteWindow	WM	$220E	Highlights or unhighlights a window's title bar, as appropriate.
HiWord	IM	$180B	Returns the high-order word of a long input.
HLock	MM	$2002	Locks a block specified by a handle.
HLockAll	MM	$2102	Locks all the blocks belonging to a specified user ID.
HomeMouse	MTS	$1A03	Positions mouse at the minimum clamp position.
HUnlock	MM	$2202	Unlocks a block specified by a handle.
HUnLockAll	MM	$2302	Unlocks all the blocks for a specified user ID.
IMBootInit	IM	$010B	Called at boot time by the Tool Locator.
IMReset	IM	$050B	Called when a system reset occurs.
IMShutDown	IM	$030B	Standard tool call.
IMStartUp	IM	$020B	Standard tool call.
IMStatus	IM	$060B	Returns a nonzero value indicating that the Integer Math Tool Set is active.
IMVersion	IM	$040B	Returns the version of the Integer Math Tool Set.
InflateText-Buffer	QD	$D704	Inflates the text buffer to a specified size, if necessary.
InitColorTable	QD	$0D04	Returns a copy of the standard color table for the current mode.
InitCursor	QD	$CA04	Reinitializes the cursor.
InitMouse	MTS	$1803	Sets mouse clamp values to $000 minimum and $3FF maximum.
InitPalette	MUM	$2F0F	Reinitializes the palettes used to draw the apple on the menu bar.
InitPort	QD	$1904	Initializes specified memory locations as a standard port.
InitTextDev	TT	$150C	Initializes a specified text device.

Call	Tool	Call Number	Function
InsertMenu	MUM	$0D0F	Inserts a menu into the menu list.
InsertMItem	MUM	$0F0F	Inserts an item into a menu.
InsetRect	QD	$4C04	Inserts a specified rectangle by specified displacements.
InsetRgn	QD	$7004	Shrinks or expands a specified region.
InstallCDA	DM	$0F05	Installs a specified classic desk accessory in the system.
InstallFont	FM	$0E1B	Loads a given font into memory and makes it current and unpurgeable.
InstallNDA	DM	$0E05	Installs a specified new desk accessory in the system.
Int2Dec	IM	$260B	Returns a string representing a 16-bit signed or unsigned integer.
Int2Hex	IM	$220B	Converts a 16-bit unsigned integer into a hexadecimal string.
IntSource	MTS	$2303	Enables or disables certain interrupt sources.
InvalRect	WM	$3A0E	Accumulates a rectangle into the current window port's update region.
InvalRgn	WM	$3B0E	Accumulates a region into the current window port's update region.
InvertArc	QD	$6504	Inverts the pixels inside a specified arc.
InvertOval	QD	$5B04	Inverts the pixels inside a specified oval.
InvertPoly	QD	$BF04	Inverts a specified polygon.
InvertRect	QD	$5604	Inverts the pixels in the interior of a specified rectangle.
InvertRgn	QD	$7C04	Inverts the pixels in the interior of a specified region.
InvertRRect	QD	$6004	Inverts the pixels inside a specified round rectangle.
IsDialogEvent	DLM	$1015	Determines if an event should be handled as part of a dialog.
ItemID2FamNum	FM	$171B	Translates a menu item ID into a font family number.
KillControls	CM	$0B10	Disposes of all controls associated with a specified window.

Call	Tool	Call Number	Function
KillPoly	QD	$C304	Disposes of a specified polygon.
LEActivate	LE	$0F14	Highlights current selection range in specified text.
LEBootInit	LE	$0114	Called at boot time by the Tool Locator.
LEClick	LE	$0D14	Using mouse clicks, draws a caret and highlights selected text.
LECopy	LE	$1314	Copies selected text into the LineEdit scrap.
LECut	LE	$1214	Removes selected text and places it in the LineEdit scrap.
LEDeactivate	LE	$1014	Unhighlights current selection range in specified text.
LEDelete	LE	$1514	Removes selected text and redraws the remaining text.
LEDispose	LE	$0A14	Releases the memory allocated for a specified edit record.
LEFromScrap	LE	$1914	Copies the desk scrap to the LineEdit scrap.
LeGetScrapLen	LE	$1C14	Returns the size of the LineEdit scrap in bytes.
LEGetTextHand	LE	$2214	Returns a handle to the text of a specified edit record.
LEGetTextLen	LE	$2314	Returns the length of the text of a specified edit record.
LEIdle	LE	$0C14	Places a blinking caret at the insertion point in a specified line.
LEInsert	LE	$1614	Inserts specified text into other text, and redraws the updated text.
LEKey	LE	$1114	Places a character in text and leaves an insertion point after it.
LENew	LE	$0914	Allocates text space and returns a handle to a new edit record.
LEPaste	LE	$1414	Replaces selected text with the contents of the LineEdit scrap.
LEReset	LE	$0514	Returns an error if LineEdit is active.
LEScrapHandle	LE	$1B14	Returns a handle to the LineEdit scrap.
LESetCaret	LE	$1F14	Sets the CaretHook field in the edit record to a specified pointer.

Call	Tool	Call Number	Function
LESetHilite	LE	$1E14	Sets the HiliteHook field in the edit record to a specified pointer.
LESetJust	LE	$2114	Sets up the LineEdit Tool Set record for left, right, or center justification.
LESetScrapLen	LE	$1D14	Sets the size of the LineEdit scrap to a specified number of bytes.
LESetSelect	LE	$0E14	Sets the selection range in the specified text.
LESetText	LE	$0B14	Incorporates a copy of specified text into a specified edit record.
LEShutDown	LE	$0314	Shuts down the LineEdit Tool Set and discards the LineEdit scrap.
LEStartUp	LE	$0214	Initializes the LineEdit Tool Set and allocates a handle for the LineEdit scrap.
LEStatus	LE	$0614	Indicates whether or not the LineEdit Tool Set is active.
LETextBox	LE	$1814	Draws specified text in a specified rectangle.
LETextBox2	LE	$2014	Draws specified text in a specified rectangle.
LETextBox2	LE	$2014	Draws text in a specified rectangle, with specified justification.
LEToScrap	LE	$1A14	Copies the LineEdit scrap to the desk scrap.
LEVersion	LE	$0414	Returns version number of the LineEdit Tool Set.
Line	QD	$3D04	Draws a line from the current pen location to the specified displacements.
LineTo	QD	$3C04	Draws a line from the current pen location to a specified point.
ListBootInit	LM	$011C	Called at boot time by the Tool Locator.
ListReset	LM	$051C	Called when a system reset occurs.
ListShutDown	LM	$031C	Standard tool call.
ListStartUp	LM	$021C	Standard tool call.
ListStatus	LM	$061C	Returns a nonzero value indicating that the List Manager is active.
ListVersion	LM	$041C	Returns the version of the List Manager.

Call	Tool	Call Number	Function
LLDBitMap	PM	$1C13	Prints part or all of a specified QuickDraw II bit map.
LLDControl	PM	$1B13	Resets the printer and generates linefeeds and formfeeds.
LLDShutDown	PM	$1A13	Deallocates any memory allocated by LLDStartUp.
LLDStartUp	PM	$1913	Sets up the necessary environment for low-level drivers.
LLDText	PM	$1D13	Prints a stream of text using the native facilities of the printer.
LoadFont	FM	$121B	Finds a specified font, loads it, and makes it current.
LoadOneTool	TL	$0F01	Loads a specified tool from disk and checks its version.
LoadScrap	SK	$0A16	Reads the desk scrap from the scrap file into memory.
LoadSysFont	FM	$131B	Makes the system font current without requiring its font ID.
LoadTools	TL	$0E01	Loads specified RAM-based tool sets from disk into memory.
LocalToGlobal	QD	$8404	Converts a point from local coordinates to global coordinates.
Long2Dec	IM	$270B	Returns a string representing a 32-bit signed or unsigned integer.
Long2Fix	IM	$1A0B	Converts long integer to fixed.
Long2Hex	IM	$230B	Converts a 32-bit unsigned integer into a hexadecimal string.
LongDivide	IM	$0D0B	Divides two 32-bit inputs, producing a quotient and a remainder.
LongMul	IM	$0C0B	Multiplies two 32-bit inputs and produces a 64-bit result.
LoWord	IM	$190B	Returns the low-order word of a long input.
MapPoly	QD	$C504	Maps a polygon from a source rectangle to a destination rectangle.
MapPt	QD	$8A04	Maps a point from a source rectangle to a destination rectangle.
MapRect	QD	$8B04	Maps a rectangle from a source rectangle to a destination rectangle.

Call	Tool	Call Number	Function
MapRgn	QD	$8C04	Maps a region from a source rectangle to a destination rectangle.
MaxBlock	MM	$1C02	Returns the size of the largest free block in memory.
MenuBootInit	MUM	$010F	Called at boot time.
MenuKey	MUM	$090F	Allows the user to type a character to select a menu item.
MenuNewRes	MUM	$290F	Restyles the menu after the screen resolution changes.
MenuRefresh	MUM	$0B0F	Called when the application is not using the Window Manager.
MenuReset	MUM	$050F	This call does nothing.
MenuSelect	MUM	$2B0F	Controls highlighting and pull-down action when an item is selected.
MenuShutDown	MUM	$030F	Closes the Menu Manager's port and frees any allocated menus.
MenuStartUp	MUM	$020F	Initializes the Menu Manager at application startup.
MenuStatus	MUM	$060F	Checks the current status of the Menu Manager.
MenuVersion	MUM	$040F	Returns the version of the Menu Manager.
MMBootInit	MM	$0102	Initializes the Memory Manager at boot time.
MMReset	MM	$0502	Used by the system at reset time.
MMShutDown	MM	$0302	An application makes this call when it is terminating.
MMStartUp	MM	$0202	An application makes this call when starting up.
MMStatus	MM	$0602	Returns status indicating the Memory Manager is active.
MMVersion	MM	$0402	Returns the version of the Memory Manager.
ModalDialog	DLM	$0F15	Repeatedly gets and handles events in a modal dialog's window.
ModalDialog2	DLM	$2C15	Repeatedly gets and handles events in a modal dialog's window.
Move	QD	$3B04	Moves the current pen location by specified X and Y displacements.
MoveControl	CM	$1610	Moves a specified control to a new location within its window.

Call	Tool	Call Number	Function
MovePortTo	QD	$2204	Changes the location of the current GrafPort's PortRect.
MoveTo	QD	$3A04	Moves the current pen location to the specified point.
MoveWindow	WM	$190E	Moves a window to another part of the screen, not changing its size.
MTBootInit	MTS	$0103	Called at boot time.
MTReset	MTS	$0503	Clears the heartbeat task pointer and sets the mouse flag to "not found."
MTShutDown	MTS	$0303	This call is not used in this tool set.
MTStartUp	MTS	$0203	This call is not used in this tool set.
MTSTATUS	MTS	$0603	Returns status indicating the Miscellaneous Tool Set is active.
MTVersion	MTS	$0403	Returns the version of the Miscellaneous Tool Set.
Multiply	IM	$090B	Multiplies two 16-bit inputs and produces a 32-bit result.
Munger	MTS	$2803	Manipulates bytes in a string of bytes.
NewControl	CM	$0910	Creates a control and returns a handle to it.
NewDItem	DLM	$0D15	Adds a new item to a dialog's item list.
NewHandle	MM	$0902	Creates a new block and returns the handle to the block.
NewList	LM	$101C	Resets the list control according to a specified list record.
NewMenu	MUM	$2D0F	Allocates space for a menu list and its items.
NewMenuBar	MUM	$150F	Creates a default menu bar with no menus.
NewModalDialog	DLM	$0A15	Creates a modal dialog and returns a pointer to its port.
NewModeless-Dialog	DLM	$0B15	Creates a modeless dialog and returns a handle to its port.
NewRgn	QD	$6704	Allocates space for a new region.
NewWindow	WM	$090E	Creates a window and returns a pointer to its GrafPort.
NextMember	LM	$0B1C	Searches a list record for a specified member and returns its value.

Call	Tool	Call Number	Function
NoteAlert	DLM	$1915	Performs the same functions as the Alert routine.
ObscureCursor	QD	$9204	Hides the cursor until the mouse moves.
OffsetPoly	QD	$C404	Offsets a polygon by specified X and Y displacements.
OffsetRect	QD	$4B04	Offsets a specified rectangle by specified displacements.
OffsetRgn	QD	$6F04	Moves a region a distance specified by X and Y displacements.
OpenNDA	DM	$1505	Opens a specified new desk accessory.
OpenPoly	QD	$C104	Opens a polygon structure for updating, and returns its handle.
OpenPort	QD	$1804	Initializes specified memory locations as a standard port.
OpenRgn	QD	$6D04	Allocates memory to hold information about a region being created.
OSEventAvail	EM	$1706	Accesses the next event of a given type but leaves it in the queue.
PackBytes	MTS	$2603	Packs bytes into a special format that uses less storage space.
PaintArc	QD	$6304	Paints the interior of an arc using the current pen state and pattern.
PaintOval	QD	$5904	Paints the interior of an oval using the current pen state and pattern.
PaintPixels	QD	$7F04	Transfers a region of pixels.
PaintPoly	QD	$BD04	Paints the interior of a polygon using the current pen state and pattern.
PaintRect	QD	$5404	Paints the interior of a rectangle using the current pen state and pattern.
PaintRgn	QD	$7A04	Paints the interior of a region using the current pen state and pattern.
PaintRRect	QD	$5E04	Paints the interior of a round rectangle using the current pen state and pattern.
ParamText	DLM	$1B15	Substitutes text in StatText and LongStatText items.
PenNormal	QD	$3604	Sets the pen state to the standard state.

Call	Tool	Call Number	Function
PinRect	WM	$210E	Pins a specified point inside a specified rectangle.
PMBootInit	PM	$0113	Called at boot time by the Tool Locator.
PMReset	PM	$0513	Internal routine called only at system reset.
PMShutDown	PM	$0313	Shuts down the Print Manager.
PMStartUp	PM	$0213	Initializes the Print Manager for use by an application.
PMStatus	PM	$0613	Indicates whether or not the Print Manager is active.
PMVersion	PM	$0413	Returns the version number of the Print Manager.
PosMouse	MTS	$1E03	Positions mouse at specified coordinates.
PostEvent	EM	$1406	Posts an event at the end of the event queue.
PPToPort	QD	$D604	Transfers pixels from a source pixel map to the current port.
PrChoosePrinter	PM	$1613	Displays a dialog for selecting a printer and port driver.
PrCloseDoc	PM	$0F13	Closes the GrafPort being used for printing.
PrClosePage	PM	$1113	Finishes the printing of the current page.
PrDefault	PM	$0913	Sets a print record to default values for the appropriate printer.
PrError	PM	$1413	Returns the result code left by the last Print Manager routine.
PrJobDialog	PM	$0C13	Displays a dialog for setting print quality, pages to print, and so on.
PrOpenDoc	PM	$0E13	Initializes a GrafPort for use in printing and returns its pointer.
PrOpenPage	PM	$1013	Begins a new page.
PrPicFile	PM	$1213	Prints a spooled document.
PrSetError	PM	$1513	Given an error number, performs a corresponding function.
PrStlDialog	PM	$0B13	Displays a dialog for inputting page setup information.
PrValidate	PM	$0A13	Checks if a print record is compatible with the Print Manager.

Call	Tool	Call Number	Function
Pt2Rect	QD	$5004	Creates a rectangle using an upper left point and a lower right point.
PtInRect	QD	$4F04	Detects if a specified point is in a specified rectangle.
PtlnRgn	QD	$7504	Determines where a specified point is within a specified region.
PtrToHand	MM	$2802	Copies a specified number of bytes from a source to a destination.
PurgeAll	MM	$1302	Purges all of the purgeable blocks for a specified user ID.
PurgeHandle	MM	$1202	Purges a specified purgeable handle.
PutScrap	SK	$0C16	Appends specified data to data in the scrap of the same type.
QDBootlnit	QD	$0104	Initializes QuickDraw II at boot time.
QDReset	QD	$0504	Resets QuickDraw II.
QDShutDown	QD	$0304	Frees up any buffers allocated for QuickDraw II.
QDStartUp	QD	$0204	Starts up QuickDraw II.
QDStatus	QD	$0604	Returns if QuickDraw II is active.
QDVersion	QD	$0404	Returns the version of QuickDraw II.
Random	QD	$8604	Returns a pseudorandom number in the range −32768 to +32767.
Read Next	ST		Reads the next address pointed to by the GLU address register.
Read RAM	ST		Reads any specified Ensoniq RAM location.
Read Register	ST		Reads any register within the DOC.
ReadASCIITime	MTS	$0F03	Reads elapsed time since 00:00:00, Jan. 1, 1904.
ReadBParam	MTS	$0C03	Reads date from a specified parameter in battery RAM.
ReadBRam	MTS	$0A03	Reads 252 bytes of data, plus 4 checksum bytes, from battery RAM.
ReadChar	TT	$220C	Reads a character from an input text device; returns it on the stack.

Call	Tool	Call Number	Function
ReadKeyMicroData	ADB	$0A09	Receive data from the microcontroller.
ReadKeyMicro-Memory	ADB	$0B09	Reads a data byte from the microcontroller ROM.
ReadLine	TT	$240C	Reads an input string and writes it to a buffer.
ReadMouse	MTS	$1703	Returns mouse position, status, and mode.
ReadRamBlock	ST	$0A08	Reads any number of locations from DOC RAM into a buffer.
ReadTimeHex	MTS	$0D03	Returns current time in hexadecimal format.
ReAllocHandle	MM	$0A02	Reallocates a block that was purged.
RectInRgn	QD	$7604	Checks whether a specified rectangle intersects a specified region.
RectRgn	QD	$6C04	Sets a specified region to a rectangle described by the input.
RefreshDesktop	WM	$390E	Redraws the entire desktop and all windows.
RemoveDItem	DLM	$OE15	Removes an item from a dialog and erases it from the screen.
ResetAlertStage	DLM	$3515	Resets a dialog so that its next stage is treated as its first stage.
ResetMember	LM	$0F1C	Searches a list record for a member and clears its select flag.
RestAll	DM	$0C05	Restores variables that were saved in calling a desk accessory.
RestoreBufDims	QD	$CE04	Restores QuickDraw's internal buffers to the sizes described in a record.
RestoreHandle	MM	$0B02	Reallocates a purged handle.
RestScrn	DM	$0A05	Restores the screen area saved by the Desk Manager.
SANEBootInit	SAN	$010	Not used in this tool set.
SANEDecStr816	SAN	$0A0A	Contains numeric scanners and formatter.
SANEDecStr816	SAN	$0B0A	Contains elementary, financial, and random number functions.
SANEFP816	SAN	$090A	Contains basic arithmetic operations and IEEE auxiliary operations.

Call	Tool	Call Number	Function
SANEReset	SAN	$050A	Not used in this tool set.
SANEShutDown	SAN	$030A	Zeros out the work area pointer for the SANE Tool Set.
SANEStartup	SAN	$020A	Starts up the SANE Tool Set for use by an application.
SANEStatus	SAN	$060A	Returns true, indicating the SANE Tool Set is active.
SANEVersion	SAN	$040A	Returns the version number of the SANE Tool Set.
SaveAll	DM	$0B05	Saves all variables preserved in activating a desk accessory.
SaveBufDims	QD	$CD04	Saves QuickDraw II's buffer sizing information in an 8-byte record.
SaveScrn	DM	$0905	Saves the 80-column text screens in banks $00, 01, EO, and E1.
ScalePt	QD	$8904	Scales a point from a source rectangle to a destination rectangle.
SchAddTask	SK	$0907	Adds a task to Scheduler's queue.
SchBootInit	SK	$0107	Initializes the flags and counters used by the Scheduler.
SchFlush	SK	$0A07	Flushes all tasks in the Scheduler's queue.
SchReset	SK	$0507	Reinitializes flags and counters.
SchShutDown	SK	$0307	Not used in this tool set.
SchStartUp	SK	$0207	Not used in this tool set.
SchStatus	SK	$0607	Returns true, indicating the Scheduler is active.
SchVersion	SK	$0407	Returns the version number of the Scheduler.
ScrapBootInit	SK	$0116	Internal routine called at load time to initialize the Scrap Manager.
ScrapReset	SK	$0516	Internal routine to reset the Scrap Manager.
ScrapShutDown	SK	$0316	Shuts down the Scrap Manager.
ScrapStartUp	SK	$0216	Starts up the Scrap Manager.
ScrapStatus	SK	$0616	Always returns true: if the Scrap Manager is loaded, it is active.

Call	Tool	Call Number	Function
ScrapVersion	SK	$0416	Returns the version number of the Scrap Manager.
ScrollRect	QD	$7E04	Scrolls a rectangle inside certain boundaries.
SDivide	IM	$0A0B	Divides two 16-bit inputs and produces two 16-bit signed results.
SectRect	QD	$4D04	Places the intersection of two rectangles in a third rectangle.
SectRgn	QD	$7104	Places the intersection of two regions in a third region.
SelectMember	LM	$0D1C	Selects a list member and scrolls the list so it is at the top.
SelectWindow	WM	$110E	Makes a specified window the active window.
SelIText	DLM	$2115	Sets the selection range or the insertion point for an EditLine item.
SendBehind	WM	$140E	Places a window behind a specified window, redrawing as appropriate.
SendInfo	ADB	$0909	Sends data to the microcontroller.
ServeMouse	MTS	$1F03	Returns the mouse interrupt status.
SetAbsClamp	MTS	$2A03	Sets clamp values for an absolute device to new values.
SetAbsScale	ADB	$1209	Sets up scaling for absolute devices.
SetAllSCBs	QD	$1404	Sets all scan-line control bytes (SCBs) to a specified value.
SetBackColor	QD	$A204	Sets a GrafPort's background color field to a specified value.
SetBackPat	QD	$3404	Sets the background pattern to a specified pattern.
SetBarColors	MUM	$170F	Sets the normal, inverse, and outline colors of the current menu bar.
SetBufDims	QD	$CB04	Sets the size of the QuickDraw II clipping and text buffers.
SetCharExtra	QD	$D404	Sets the chExtra field in the GrafPort to the specified value.
SetClip	QD	$2404	Copies a specified region into the ClipRgn.
SetClipHandle	QD	$C604	Sets the ClipRgn handle field in the GrafPort to a specified value.

Call	Tool	Call Number	Function
SetColorEntry	QD	$1004	Sets the value of a color in a specified color table.
SetColorTable	QD	$0E04	Sets a color table to specified values.
SetContentDraw	WM	$490E	Sets the pointer to a routine that redraws a window's content region.
SetContentOrigin	WM	$3F0E	Sets the origin of the window's port when handling an update event.
SetCtlAction	CM	$2010	Sets a specified control's CtlAction field to a new action.
SetCtlIcons	CM	$1810	Provides a handle to a specified new icon font.
SetCtlParams	CM	$1B10	Sets new parameters to a control's definition procedure.
SetCtlRefCon	CM	$2210	Sets a specified control's CtlReCon field to a new value.
SetCtlTitle	CM	$0C10	Sets a control's title to a given string and redraws the control.
SetCtlValue	CM	$1910	Sets a control's CtlValue field and redraws the control.
SetCursor	QD	$8E04	Sets the cursor to an image passed in a specified cursor record.
SetDAFont	DLM	$1C15	Sets the font of a given window's port to a specified font number.
SetDAStrPtr	DM	$1305	Allows a program to change the built-in classic desk accessories.
SetDataSize	WM	$410E	Sets the height and width of the data area of a specified window.
SetDefButton	DLM	$3815	Sets the ID of the default button to a specified ID.
SetDefProc	WM	$320E	Sets the address of the routine that defines a window's behavior.
SetDItemBox	DLM	$2915	Changes the display rectangle of an item to a new display rectangle.
SetDItemType	DLM	$2715	Changes the specified item to the new desired type.
SetDItemValue	DLM	$2F15	Sets the value of an item to a new value and redraws the item.
SetEmptyRgn	QD	$6A04	Sets a specified region to the empty region.

Call	Tool	Call Number	Function
SetErrGlobals	TT	$0B0C	Sets the global parameters for the error output device.
SetErrorDevice	TT	$110C	Sets the error output device to a specified type and location.
SetEventMask	EM	$1806	Sets the system event mask to the specified event mask.
SetFont	QD	$9404	Sets the current font to the specified font.
SetFontFlags	QD	$9804	Sets the font flags to the specified value.
SetFontID	QD	$D004	Sets the FontID field in the GrafPort.
SetForeColor	QD	$A004	Sets a GrafPort's foreground color field to a specified value.
SetFrameColor	WM	$0F0E	Sets the color of a specified window's frame.
SetGrafProcs	QD	$4404	Sets a GrafPort's GrafProcs field to a specified value.
SetHandleSize	MM	$1902	Changes the size of a specified block.
SetHeartBeat	MTS	$1203	Installs a specified task into the heartbeat interrupt task queue.
SetInfoDraw	WM	$160E	Sets the pointer to a window's information bar drawing procedure.
SetInfoRefCon	WM	$360E	Sets a value associated with a window's information bar drawing routine.
SetInputDevice	TT	$0F0C	Sets the input device to a specified type and location.
SetIntUse	QD	$B604	Tells if the cursor should be drawn using scan-line interrupts.
SetIText	DLM	$2015	Fetches a string for an item that contains text and redraws the item.
SetInGlobals	TT	$090C	Sets the global parameters for the input device.
SetMasterSCB	QD	$1604	Sets the master SCB to a specified value.
SetMaxGrow	WM	$430E	Sets the maximum values to which a window's content region can grow.
SetMenuBar	MUM	$390F	Sets the current menu bar.
SetMenuFlag	MUM	$1F0F	Sets the menu to a specified state.
SetMenuID	MUM	$370F	Specifies a new menu number.

Call	Tool	Call Number	Function
SetMenuTitle	MUM	$210F	Specifies the title for a menu.
SetMItem	MUM	$240F	Specifies the name for a menu item.
SetMItemBlink	MUM	$280F	Determines how many times all menu items should blink when selected.
SetMItemFlag	MUM	$260F	Controls the style of an item's highlighting and underlining.
SetMItemID	MUM	$380F	Specifies the ID number of a menu item.
SetMItemMark	MUM	$330F	Sets a specified character to display or not display to the left of a menu item.
SetMItemName	MUM	$3A0F	Specifies the name for a menu item.
SetMItemStyle	MUM	$350F	Sets the text style for a specified menu item.
SetMouse	MTS	$1903	Sets the mode value for the mouse.
SetMTitleStart	MUM	$190F	Sets the starting point for the leftmost menu on the menu bar.
SetMTitleWidth	MUM	$1D0f	Sets the width of a title.
SetOrigin	QD	$2304	Sets the upper left corner of the PortRect to a given point.
SetOriginMask	WM	$340E	Specifies the mask used to put the horizontal origin on a grid.
SetOutGlobals	TT	$0A0C	Sets the global parameters for the error output device.
SetOutputDevice	TT	$100C	Sets the output device to a specified type and location.
SetPage	WM	$470E	Sets the number of pixels that define a "page" for scrolling.
SetPenMask	QD	$3204	Sets the pen mask to the specified mask.
SetPenMode	QD	$2E04	Sets the current pen mode to the specified pen mode.
SetPenPat	QD	$3004	Sets the current pen pattern to the specified pen pattern.
SetPenSize	QD	$2C04	Sets the current pen size to the specified pen size.
SetPenState	QD	$2A04	Sets the pen state in the GrafPort to the specified values.
SetPicSave	QD	$3E04	Sets the PicSave field to a specified value.
SetPolySave	QD	$4204	Sets the PolySave field to a specified value.

Call	Tool	Call Number	Function
SetPort	QD	$1B04	Makes the specified port the current port.
SetPortLoc	QD	$1D04	Sets the current port's map information structure.
SetPortRect	QD	$1F04	Sets the current port's rectangle to the specified rectangle.
SetPortSize	QD	$2104	Changes the size of the current GrafPort's PortRect.
SetPt	QD	$8204	Sets a point to specified horizontal and vertical values.
SetPurge	MM	$2402	Sets the purge level of a block specified by a handle.
SetPurgeAll	MM	$2502	Sets the purge level of all blocks for a specified user ID.
SetPurgeStat	FM	$0F1B	Makes a specified font in memory unpurgeable or purgeable.
SetRandSeed	QD	$8704	Sets the seed value for the random number generator.
SetRect	QD	$4A04	Sets a specified rectangle to specified values.
SetRectRgn	QD	$6B04	Sets a region to a specified rectangle.
SetRgnSave	QD	$4004	Sets the RgnSave field to a specified value.
SetSCB	QD	$1204	Sets the scan-line control byte (SCB) to a specified value.
SetScrapPath	SK	$1116	Sets the clipboard file pointer to the specified value.
SetScroll	WM	$450E	Sets the number of pixels that will be scrolled by scroll bar arrows.
SetSolidBackPat	QD	$3804	Sets the background pattern to a solid pattern using a certain color.
SetSolidPenPat	QD	$3704	Sets the pen pattern to a solid pattern using the specified color.
SetSoundMIRQV	ST	$1208	Sets the entry point into the sound interrupt handler.
SetSoundVolume	ST	$0D08	Changes the DOC registers' volume setting, or the system volume.
SetSpaceExtra	QD	$9E04	Sets the spExtra field in the GrafPort to the specified value.
SetStdProcs	QD	$8D04	Sets up a record of pointers for customizing QuickDraw II operations.

Call	Tool	Call Number	Function
SetSwitch	EM	$1306	Informs the Event Manager of a pending switch event.
SetSysBar	MUM	$120F	Sets a new system bar.
SetSysField	QD	$4804	Sets the SysField in the GrafPort to a specified value.
SetSysFont	QD	$B204	Sets a specified font as the system font.
SetSysWindow	WM	$4B0E	Marks a specified window as a system window.
SetTextFace	QD	$9A04	Sets the text face to the specified value.
SetTextMode	QD	$9C04	Sets the text mode to the specified value.
SetTextSize	QD	$D204	Call is not implemented at the time of this writing.
SetTSPtr	TL	$0A01	Call used to modify tool sets by installing patches.
SetUserField	QD	$4604	Sets the UserField in the GrafPort to a specified value.
SetUserSoundIRQV	ST	$1308	Sets the entry point for an application-defined interrupt handler.
SetVector	MTS	$1003	Sets the vector address for the specified vector reference number.
SetVisHandle	QD	$C804	Sets the VisRgn handle field in the GrafPort to a specified value.
SetVisRgn	QD	$B404	Copies a specified region into the VisRgn.
SetWAP	TL	$0D01	Sets the pointer to the work area for a specified tool set.
SetWFrame	WM	$2D0E	Sets the bit vector that describes a specified window's frame type.
SetWindowIcons	WM	$4E0E	Sets the icon font for the Window Manager.
SetWRefCon	WM	$280E	Sets a window-record value reserved for an application's use.
SetWTitle	WM	$0D0E	Updates the title of a specified window.
SetZoomRect	WM	$380E	Sets the rectangle used to calculate a window's zoomed size.
SFAllCaps	SF	$0D17	Allows an application to display file names in all uppercase.

Call	Tool	Call Number	Function
SFBootInit	SF	$0117	Initializes the Standard File Operations Tool Set at boot time.
SFGetFile	SF	$0917	Displays the Standard File Operations Tool Set's standard dialog; returns data about selected files.
SFPGetFile	SF	$0B17	Displays a custom dialog and returns information on selected files.
SFPPutFile	SF	$0A17	Displays a custom dialog and returns data on files to be saved.
SFPutFile	SF	$0A17	Displays a dialog and returns data about the file to be saved.
SFReset	SF	$0517	Resets the Standard File Operations Tool Set.
SFShutdown	SF	$0317	Shuts down the Standard File Operations Tool Set.
SFStartUp	SF	$0217	Starts up the Standard File Operations Tool Set.
SFStatus	SF	$0617	Tells if the Standard File Operations Tool Set is active.
SFVersion	SF	$0417	Returns the version number of the Standard File Operations Tool Set.
ShowControl	CM	$0F10	Makes a specified control visible.
ShowCursor	QD	$9104	Shows the cursor incrementing its level to 0, if necessary.
ShowDItem	DLM	$2315	Makes visible a specified item from a specified dialog.
ShowHide	WM	$230E	Shows or hides a window, depending upon a specified parameter.
ShowPen	QD	$2804	Increments the pen level.
ShowWindow	WM	$130E	Makes a specified window visible and draws it if it was invisible.
SizeWindow	WM	$1C0E	Sizes a window's port rectangle to a specified width and height.
SolidPattern	QD	$3904	Sets a specified pattern to a solid pattern using a specified color.
SortList	LM	$0A1C	Alphabetizes a list by rearranging the array of member records.
SoundBootInit	ST	$0108	Called by the Tool Locator at initialization.

Call	Tool	Call Number	Function
SoundReset	ST	$0508	Stops all generators that may be generating sound.
SoundShutDown	ST	$0308	Shuts down the Sound Manager.
SoundStartUp	ST	$0208	Initializes a work area to be used by the sound tools.
SoundToolStatus	ST	$0608	Returns status indicating whether the Sound Tool Set is active.
SoundVersion	ST	$0408	Returns the version of the Sound Tool Set.
SRQPoll	ADB	$1409	Adds a device to the SRQ list if the device exists.
SRQRemove	ADB	$1509	Removes a device from the SRQ list.
StartDrawing	WM	$4D0E	Makes a specified window the current port and sets its origin.
StartInfoDrawing	WM	$500E	Used for drawing outside a window's information bar procedure.
StatusID	MTS	$2203	Inquires whether or not a specified user ID is active.
StatusTextDev	TT	$170C	Executes a status call to a specified text device.
StillDown	EM	$0E06	Tests if the specified mouse button is still down.
StopAlert	DLM	$1815	Performs the same functions as the Alert routine.
StringBounds	QD	$AD04	Sets a specified rectangle to be the bounds of a specified string.
StringWidth	QD	$A904	Returns the width in pixels of a specified string.
SubPt	QD	$8104	Subtracts one point from another; leaves result in destination point.
SynchADBReceive	ADB	$0E09	Receive data from an ADB device.
SysBeep	MTS	$2C03	Calls the Apple II monitor entry point BEEL1.
SysFailMgr	MTS	$1503	Displays a system failure message and ends a program.
SystemClick	DM	$1705	Called when application detects a mouse down in a system window.
SystemEdit	DM	$1805	Passes standard menu edits to system windows.

Call	Tool	Call Number	Function
SystemEvent	DM	$1A05	Entry point for the Event Manager into the Desk Manager.
SystemTask	DM	$1905	Called periodically to support desk accessory actions.
TaskMaster	WM	$1D0E	Calls GetNextEvent and then handles certain other events itself.
TestControl	CM	$1410	Tests which part of a control contains a specified point.
TextBootInit	TT	$010C	Called at boot time; sets up certain default device parameters.
TextBounds	QD	$AF04	Sets a specified rectangle to be the bounds of the specified text.
TextReadBlock	TT	$230C	Reads a block of input characters and writes it to a buffer.
TextReset	TT	$050C	Resets device parameters to the defaults.
TextShutDown	TT	$030C	A standard call that is unnecessary and performs no function.
TextStartUp	TT	$020C	A startup call that is unnecessary and performs no function.
TextStatus	TT	$060C	Returns $FFF, indicating the Text Tool Set is active.
TextVersion	TT	$040C	Returns the version of the Text Tool Set.
TextWidth	QD	$AB04	Returns the width of the specified text.
TextWriteBlock	TT	$1E0C	Writes a block of text to the output text device.
TickCount	EM	$1006	Returns the number of ticks since the system last started up.
TLBootInit	TL	$0101	Initializes the Tool Locator and all other ROM-based tool sets.
TLMountVolume	TL	$1101	Displays a simulated dialog asking the user to mount a volume.
TLReset	TL	$0501	Initializes the Tool Locator and other ROM-based tool sets.
TLShutDown	TL	$0301	Shuts down the Tool Locator when an application shuts down.
TLStartUp	TL	$0201	Starts up the Tool Locator when an application starts up.
TLStatus	TL	$0601	Returns true, indicating the Tool Locator is active.

Call	Tool	Call Number	Function
TLTextMount-Volume	TL	$1201	Displays a 40-column text window asking the user to mount a volume.
TLVersion	TL	$0401	Returns the version of the Tool Locator.
TotalMem	MM	$1D02	Returns the size of all memory, including the main 256K.
TrackControl	CM	$1510	Follows mouse movements until the mouse button is released.
TrackGoAway	WM	$180E	Removes a window from the screen when the go-away box is clicked.
TrackZoom	WM	$260E	Zooms a window when the mouse is clicked in the zoom box.
UDivide	IM	$0B0D	Divides two 16-bit inputs, producing a quotient and a remainder.
UnionRect	QD	$4E04	Places the union of two rectangles in a third rectangle.
UnionRgn	QD	$7204	Places the union of two regions in a third region.
UnloadOneTool	TL	$1001	Unloads a specified tool from memory.
UnloadScrap	SK	$0916	Writes the desk scrap to the scrap file, and releases its memory.
UnPackBytes	MTS	$2703	Unpacks data from the packed format used by PackBytes.
UpdateDialog	DLM	$2515	Redraws the part of a dialog that is in an update region.
ValidRect	WM	$3C0E	Removes a given rectangle from the current window's update region.
ValidRgn	WM	$3D0E	Removes a specified region from the current window's update region.
WaitMouseUp	EM	$0F06	Tests if the mouse button is still down.
WindBootInit	WM	$010E	Initializes the Window Manager at boot time.
WindDragRect	WM	$530E	Pulls around an outline of a rectangle, following mouse movements.
WindNewRes	WM	$250E	Called after the screen resolution is changed.
WindReset	WM	$050E	Resets the Window Manager.

Call	Tool	Call Number	Function
WindShutDown	WM	$030E	Shuts down the Window Manager.
WindStartUp	WM	$020E	Initializes the Window Manager.
WindStatus	WM	$060E	Returns whether or not the Window Manager is active.
WindVersion	WM	$040E	Returns the version number of the Window Manager.
Write Next	ST		Writes 1 byte of data to the next DOC register or RAM address.
Write RAM	ST		Writes a 1-byte value to any specified Ensoniq RAM location.
Write Register	ST		Writes a 1-byte parameter to any register in the DOC chip.
WriteBParam	MTS	$0B03	Writes data to a specified parameter in battery RAM.
WriteBRam	MTS	$0903	Writes 252 bytes, plus 4 checksum bytes, to the battery RAM.
WriteChar	TT	$180C	Writes a character to the output text device.
WriteCString	TT	$200C	Writes a C-style string to the output text device.
WriteLine	TT	$1A0C	Writes a string, plus a carriage return, to the output text device.
WriteRamBlkock	ST	$09080	Writes a specified number of bytes from system RAM into DOC RAM.
WriteString	TT	$1C0C	Writes a string to the output text device.
WriteTimeHex	MTS	$0E03	Sets the current time using hexadecimal format.
X2Fix	IM	$200B	Converts extended to fixed.
X2Frac	IM	$210B	Converts extended to fraction.
XorRgn	QD	$7404	Extend-ORs two regions and places the result in a third region.
ZeroScrap	SK	$0B16	Clears the contents of the scrap.
ZoomWindow	WM	$270E	Zooms a window to its maximum size when the zoom box is clicked.

Bibliography

The Apple IIe User's Guide.
New York: Macmillan, 1983.

Apple Human Interface Guidelines.
Menlo Park, CA: Addison-Wesley, 1987.

Apple Numerics Manual.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Firmware Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Hardware Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS ProDOS 16 Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Programmer's Workshop Assembler Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Programmer's Workshop C Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Programmer's Workshop Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Apple IIGS Toolbox Reference.
Menlo Park, CA: Addison-Wesley, 1987.

Programmer's Introduction to the Apple IIGS.
Menlo Park, CA: Addison-Wesley, 1986.

Technical Introduction to the Apple IIGS.
Menlo Park, CA: Addison-Wesley, 1986.

Andrews, Mark

Apple Roots: Assembly Language Programming.
Berkeley: Osborne McGraw-Hill, 1986.

Eyes, David

Programming the 65816.
New York: Brady (Prentiss-Hall), 1986.

Findley, Robert

6502 Software Gourmet Guide and Cookbook.
Rochelle Park, NJ: Hayden Book Co., Inc., 1979.

Fischer, Michael

Apple IIGS Technical Reference.
Berkeley: Osborne McGraw-Hill, 1986, 1987.

Goodman, Danny

The Apple IIGS Toolbox Revealed.
New York: Bantam Books, 1986.

Hunter, Bruce H.

Understanding C.
Berkeley: Sybex, 1984.

Kerninghan, Brian W.

The C Programming Language.
Englewood Cliffs, NJ: Prentiss-Hall, 1978.

Leventhal, Lance A.

6502 Assembly Language Programming.
Berkeley: Osborne McGraw-Hill, 1979.

6502 Assembly Language Subroutines.
Berkeley: Osborne McGraw-Hill, 1982.

Maurer, W. Douglas

Apple Assembly Language.
Rockville, MD: Computer Science Press, Inc., 1984.

Mottola, Robert

Assembly Language Programming for the Apple II.
Berkeley: Osborne McGraw-Hill, 1982.

Wagner, Roger

Assembly Lines: The Book.
Santee, CA: Roger Wagner Publishing, Inc., 1984.

Waite, Mitchell

C Primer Plus.
Indianapolis, IN: Howard W. Sams & Co., Inc., 1985.

Zaks, Rodnay

Programming the Apple II in Assembly Language.
Berkeley: Sybex, 1983.

Programming the 6502.
Berkeley: Sybex, 1983.

Index

HOWARD W. SAMS & COMPANY

Programming the Apple IIGS™
in C and Assembly Language

Learning how to program the Apple IIGS is easy with this book by best-selling author Mark Andrews.

The first of its kind to include both assembly language and C, this book enables professional programmers and hobbyists alike to take advantage of the power, speed, graphics, and sound capabilities of the IIGS.

Packed with useful, entertaining type-and-run programs, *Programming the Apple IIGS in C and Assembly Language* equips you with all you need to program the Apple IIGS in C and integrate assembly language to supercharge your programs.

In this plain-English guide, you'll discover

- How to program the Apple IIGS's 65C816 chip in assembly language
- How to use the Apple IIGS Programmer's Workshop program development system
- How to create mouse-driven programs with such eye-catching graphics features as pull-down menus, multiple screen windows, icons, and dialog boxes
- How to write sound tracks for your programs using the IIGS's 15-voice, 32-oscillator sound and music synthesizer

To use this book, you need an Apple IIGS with at least two 3.5-inch disk drives, a monochrome or color monitor, and a memory expansion card with at least 512K of additional RAM.

Mark Andrews, an experienced program designer and technical writer, currently works as an assembly language programmer at Apple Computer, Inc. He has written eight books about computers and computer programming, including *Commodore 128® Assembly Language Programming* and *Commodore 64®/128 Assembly Language Programming* for Howard W. Sams & Company. He has also written hundreds of technically oriented magazine and newspaper articles and has designed and developed many commercial microcomputer programs.

$18.95/22599

ISBN: 0-672-22599-9

HOWARD W. SAMS & COMPANY
A Division of Macmillan, Inc.
4300 West 62nd Street
Indianapolis, Indiana 46268 USA