Start Manager

This chapter describes the system initialization and system startup process performed by the Macintosh computer. It describes the Start Manager, which lets you specify a few global settings that affect the startup process, and it describes initialization-dependent code, such as system extensions, that the system runs while starting up the computer.

You should read this chapter if you are developing a device driver or other code that is installed at some point during the system initialization and startup process, or if you want to use the Start Manager routines.

This chapter begins with a description of the initialization and startup process performed on Macintosh computers. It then

- describes the boot blocks and defines the fields in the boot block header
- defines global variables that provide timing information
- discusses the Start Manager routines you can use to identify and set default devices and to get and set the timeout interval for the startup drive
- describes how to write a system extension

System Initialization and Startup

When a Macintosh computer is first turned on, but before it can load and run an application, it must go through system initialization and system startup. At ***system initialization***, the system initialization code located in ROM is executed: memory is tested and initialized, slot cards are initialized, ROM drivers are installed, device drivers are located, and more. The next section, "System Initialization," describes the various steps included in system initialization. At ***system startup***, the system code that is located on the startup disk is executed: various software modules are initialized and system extensions are run. The section "System Startup" on page 9-4 describes various steps included in system startup.

▲ **WARNING**
The system initialization and system startup process is not the same for all Macintosh models. In addition, the system initialization sequence and system startup sequence listed in this chapter are both subject to change; therefor use the information in these sections only for informational purposes. ▲

You should read this section if you provide a system extension that installs software, such as a device driver or other code, during system initialization or system startup.

System Initialization

Initialization on a Macintosh computer begins as soon as the power is first supplied to it. Built-in hardware circuits initialize the main processor and other ICs and temporarily alter the memory mapping to make an image of the ROM appear at the location where RAM normally starts (address 0), while making RAM appear at a location higher in memory. This mapping scheme allows the startup routines in the initialization code to obtain critical low-memory vectors. After the initialization code begins executing and obtains the low-memory vectors, it resets the memory mapping back to normal. For further details on this process, see the *Guide to Macintosh Family Hardware.*

The following list summarizes the events that typically take place when the initialization code in ROM is executed.

IMPORTANT

The system initialization sequence is subject to change; the information in this section is provided for informational purposes only. ▲

1. Hardware is initialized. The initialization code performs a set of diagnostic tests to verify functionality of some vital hardware components. If the diagnostics succeed, the initialization code initializes these hardware components. If diagnostics fail, the initialization code issues diagnostic tones to indicate the type of hardware failure. The initialization code determines how much RAM is available and tests it, then validates the parameter RAM (PRAM). Parameter RAM contains a user's preferences for settings of various control panel settings and port configurations. The initialization code determines the global timing variables, `TimeDBRA`, `TimeSCCDB`, and `TimeSCSIDB`. (See "Global Timing Variables" on page 9-9 for more information) and initializes the Resource Manager, Notification Manager, Time Manager, and Deferred Task Manager.
2. On machines with expansion slots, the initialization code initializes the Slot Manager. The Slot Manager then initializes any installed cards by executing the primary initialization code in each card's declaration ROM. Video expansion cards, including built-in video, initialize themselves by determining the type of connected monitor, and then set the display to 1 bit per pixel, and display a gray screen (alternating black and white dots).
3. The initialization code initializes the Vertical Retrace Manager and Gestalt Manager. ROM drivers for all built-in functionality are installed in the unit table and initialized. The initialization code initializes the Apple Desktop Bus (ADB) Manager that then initializes each ADB device. The initialization code initializes the Sound Manager and SCSI Manager.
4. The initialization code loads drivers from all on-line SCSI devices.
5. The initialization code chooses the boot device, and calls the boot blocks to begin initialization of the System Software.

Having initialized the computer's slots, drivers, and hardware, as well as some of the Operating System managers, the initialization code dispatches to the startup code, which immediately begins the startup procedure described in the next section, "System Startup."

System Startup

System startup begins as soon as the initialization code in ROM transfers control to the system startup code. The system startup code is responsible for initializing AppleTalk, the debugger, and system extensions. System extensions are covered in detail in the section "Writing a System Extension" beginning on page 9-10.

This section covers the startup sequence for Macintosh computers running System 7 or later; it then describes the boot blocks and defines the boot block header.

The following list summarizes the events that take place when the system startup code is executed.

IMPORTANT

The system startup sequence is subject to change; the information in this section is provided for informational purposes only. ▲

1. The system startup code looks for an appropriate startup device. It first checks the internal 3.5-inch floppy drive. If a disk is found, it attempts to read it and looks for a System file. If it doesn't find a disk or System file, it checks the default startup device specified by the user in the Startup Disk control panel. If no default device is specified or if the device specified is not connected, it checks for other devices connected to the SCSI port, beginning with the internal drive and proceeding successively from drive 6 through drive 1. If it doesn't find a startup device, it displays the question-mark disk icon until a disk is inserted. If the startup device itself fails, the startup code displays the sad Macintosh icon until the computer is turned off.
2. After selecting a startup device, the system startup code reads system startup information from the startup device. The system startup information is located in the boot blocks, the logical blocks 0 and 1 on the startup disk. The boot blocks contain important information such as the name of the System file and the Finder. The boot blocks are described in detail in the next section.
3. The system startup code displays the happy Macintosh icon.
4. The system startup code reads the System file and uses that information to initialize the System Error Handler and the Font Manager.
5. The system startup code verifies that the necessary hardware is available to boot the system software and displays on the startup screen an alert box with the message "Welcome to Macintosh."
6. The system startup code performs miscellaneous tasks: it verifies that enough RAM is available to boot the system software, it loads and turns on Virtual Memory if it is enabled in the Memory control panel, it loads the debugger, if present. (The system startup information contains the name of the debugger —usually MacsBug), it sets up the disk cache for the file system, and it loads and executes CPU-specific software patches. At this point, the system begins to trace mouse movement.
7. For any NuBus cards installed, the system startup code executes the secondary init code on the card's declaration ROM.
8. The system startup code loads and initializes all script systems, including components for all keyboard input methods. It also executes the initialization resources in the System file.
9. The system startup code loads and executes system extensions. (System extensions can be located in the Extensions folder, in the Control Panels folder, and in the System Folder).

10. The system startup code launches the Process Manager, which takes over at this point and launches the Finder. The Finder then displays the desktop and the menu bar. The desktop shows all mounted volumes; it also shows any windows that were open the last time the computer was shut down. The Memory Manager sets up a large, unsegmented application heap, which is divided into partitions as applications start up.

At this point, the system has successfully booted.

The next section, "Boot Blocks," describes the format of the boot block header. This header contains information that the startup code uses to start up the system.

Boot Blocks

The first two logical blocks on every Macintosh volume are ***boot blocks.*** These blocks contain ***system startup information:*** instructions and information necessary to start up (or "boot") a Macintosh computer. This information consists of certain configurable system parameters (such as the capacity of the event queue, the number of open files allowed, and so forth) and is contained in a boot block header. The system startup information also includes actual machine-language instructions that could be used to load and execute the System file. Usually these instructions follow immediately after the boot block header. Generally, however, the boot code stored on disk is ignored in favor of boot code stored in a resource in the System file.

The boot block header has a structure that can be described by the `BootBlkHdr` data type.

▲ **WARNING**
The format of the boot block header is subject to change. If your application relies on the information presented here, it should check the boot block header version number and react gracefully if that number is greater than that documented here. ▲

Note that there are two boot block header formats. The current format includes two fields at the end that are not contained in the older format. These fields allow the Operating System to size the system heap relative to the amount of available physical RAM. A boot block header that conforms to the older format sizes the system heap absolutely, using values specified in the header itself. You can determine whether a boot block header uses the current or the older format by inspecting a bit in the high-order byte of the `bbVersion` field, as explained in its field description.

```
TYPE BootBlkHdr =          {boot block header}
RECORD
   bbID:          Integer; {boot blocks signature}
   bbEntry:       LongInt; {entry point to boot blocks}
   bbVersion:     Integer; {boot blocks version number}
   bbPageFlags:   Integer; {used internally}
   bbSysName:     Str15;   {System filename}
   bbShellName:   Str15;   {Finder filename}
   bbDbg1Name:    Str15;   {first debugger filename}
   bbDbg2Name:    Str15;   {second debugger filename}
   bbScreenName:  Str15;   {name of startup screen}
   bbHelloName:   Str15;   {name of startup program}
   bbScrapName:   Str15;   {name of system scrap file}
   bbCntFCBs:     Integer; {number of FCBs to allocate}
   bbCntEvts:     Integer; {number of event queue elements}
   bb128KSHeap:   LongInt; {system heap size on 128K Mac}
   bb256KSHeap:   LongInt; {system heap size on 256K Mac}
   bbSysHeapSize: LongInt; {system heap size on all machines}
   filler:        Integer; {reserved}
   bbSysHeapExtra:LongInt; {additional system heap space}
   bbSysHeapFract:LongInt; {fraction of RAM for system heap}
END;
```

Field descriptions

`bbID` A signature word. For Macintosh volumes, this field always contains the value $4C4B.

`bbEntry` The entry point to the boot code stored in the boot blocks. This field contains machine-language instructions that translate to `BRA.S *+$90` (or `BRA.S *+$88`, if the older block header format is used), which jumps to the main boot code following the boot block header. This field is ignored, however, if bit 6 is clear in the high-order byte of the `bbVersion` field or if the low-order byte in that field contains $D.

`bbVersion` A flag byte and boot block version number. The high-order byte of this field is a flag byte whose bits have the following meanings:

Bit	Meaning
0-4	Reserved; must be 0
5	Set if relative system heap sizing is to be used
6	Set if the boot code in boot blocks is to be executed
7	Set if new boot block header format is used

If bit 7 is clear, then bits 5 and 6 are ignored and the version number is found in the low-order byte of this field. If that byte contains a value that is less than $15, the Operating System ignores any values in the `bb128KSHeap` and `bbSysHeapSize` fields and configures the system heap to the default value contained in the `bbSysHeapSize` field. If that byte contains a value that is greater than or equal to $15, the Operating System sets the system heap to the value in `bbSysHeapSize`. In addition, the Operating System executes the boot code in the `bbEntry` field only if the low-order byte contains $D.

If bit 7 is set, the Operating System inspects bit 6 to determine whether to execute the boot code contained in the `bbEntry` field and inspects bit 5 to determine whether to use relative sizing of the

	system heap. If bit 5 is clear, the Operating System sets the system heap to the value in `bbSysHeapSize`. If bit 5 is set, the system heap is extended by the value in `bbSysHeapExtra` plus the fraction of available RAM specified in `bbSysHeapFract`.
`bbPageFlags`	Used internally.
`bbSysName`	The name of the System file.
`bbShellName`	The name of the shell file. Usually, the system shell is the Finder.
`bbDbg1Name`	The name of the first debugger installed during the boot process. Typically this is Macsbug.
`bbDbg2Name`	The name of the second debugger installed during the boot process. Typically, this is Disassembler.
`bbScreenName`	The name of the file containing the information (welcome message) initially displayed on the startup screen. Usually, this is StartUpScreen.
`bbHelloName`	The name of the startup program. Usually, this is the Finder.
`bbScrapName`	The name of the system scrap file. Usually, this is the Clipboard.
`bbCntFCBs`	The number of file control blocks (FCBs) to put in the FCB buffer. In System 7 and later, this field specifies only the initial number of FCBs in the FCB buffer because the Operating System can usually resize the FCB buffer if necessary. See the chapter "File Manager" in *Inside Macintosh: Files* for details on the file control block (FCB) buffer.
`bbCntEvts`	The number of event queue elements to allocate. This number determines the maximum number of events that can be stored by the Event Manager at any one time. Usually this field contains the value 20.
`bb128KSHeap`	The size of the system heap on a Macintosh computer having 128 KB of RAM.
`bb256KSHeap`	Reserved.
`bbSysHeapSize`	The size of the system heap on a Macintosh computer having 512 KB or more of RAM. This field might be ignored, as explained in the description of the `bbVersion` field.
`filler`	Reserved.
`bbSysHeapExtra`	The minimum amount of additional system heap space required. If bit 5 of the high-order word of the `bbVersion` field is set, this value is added to the `bbSysHeapSize`.
`bbSysHeapFract`	The fraction of RAM available to be used for the system heap. If bit 5 of the high-order word of the `bbVersion` field is set, this fraction of available RAM is added to the `bbSysHeapSize`.

Global Timing Variables

During system initialization, the initialization code initializes the following global variables with timing information.

Variable	Contents
`TimeDBRA`	The number of times the DBRA (decrement branch always instruction) can be executed per millisecond.
`TimeSCCDB`	The number of times the SCC can be accessed per millisecond.
`TimeSCSIDB`	The number of times the SCSI can be accessed per millisecond.

Note
The `TimeDBRA` value is calculated in ROM and is affected by the processing method of the CPU. Accordingly, for routines running in RAM, it is not necessarily a good measure of how fast the computer is. ◆

About the Start Manager

The Start Manager lets you set the Macintosh computer’s default startup and video devices. The Start Manager also lets you get or set the timing interval for the startup drive.

The Start Manager provides routines that let you specify a default startup device, a default video device, a default operating system, and a default timeout interval for the startup drive. Because all Start Manager routines run under the Macintosh Operating System, you cannot execute them early enough in the initialization process to transfer control to another operating system. Start Manager routines constitute just a small part of the process required to boot another operating system on a Macintosh computer. Most programmers should have no reason to use these routines.

The next section gives an overview of how to use the Start Manager routines.

Using the Start Manager

The Start Manager provides a set of simple routines that get and set information in a word in parameter RAM. This information indicates the default status of some peripheral devices connected to the Macintosh computer. Three of these routines get information about the default startup device, default video device, and the default operating system. Another three routines enable you to set this information. The remaining two routines get and set the timeout interval for the startup drive.

The `GetDefaultStartup` procedure returns information about the default startup device, and the `SetDefaultStartup` procedure lets you specify a slot or SCSI device as the default startup device. The ***default startup device*** is the drive on which the startup code first attempts to start up the Operating System. The Startup Disk control panel calls the `GetDefaultStartup` and `SetDefaultStartup` procedures when the user changes the startup disk. Another pair of routines, the `GetVideoDefault` and `SetVideoDefault` procedures, get information about and set the ***default video device***— essentially, the monitor on which the Macintosh computer displays the message "Welcome to Macintosh" and other startup information. The Monitors control panel calls the `GetVideoDefault` and `SetVideoDefault` procedures when the user changes the startup screen. Any changes made to settings in the Monitors control panel take affect at the next system startup.

A third pair of routines, the `GetOSDefault` and `SetOSDefault` procedures, enable you to get information about and set the ***default operating system*** —the operating system that the processor attempts to initialize and start up. At present, the only default operating systems allowed is the Macintosh Operating System.

The last two routines, the `GetTimeout` and `SetTimeout` procedures, get or set the timeout interval for the startup drive. The ***timeout interval*** is the interval of time the system waits for the startup drive to respond while the computer is booting. A disk driver might need to change the timeout interval, for example if the drive takes a long time to reach operating speed.

Writing a System Extension

This section discusses

- the profile of a system extension
- the user interface for a system extension
- how to create additional resources for a system extension
- how to compile a system extension

Before you begin to write a system extension, consider whether the feature that you have in mind is best governed by a system extension. A system extension does not enjoy the full status of an application. The user cannot launch a system extension. During system startup, each system extension is simply loaded and executed in a temporary heap that the system deallocates after the extension is called.

Profile of a System Extension

A ***system extension*** is a file (of file type '`INIT`') containing a code resource of type '`INIT`' and additional other resources. A system extension typically contains code that provides a system-level service, such as a printer driver or a patch to a system software routine, and it contains code that loads this system-level service into the system at system startup time.

Listing 9-1 illustrates code for a simple system extension called `MySampleINIT`. When launched at system startup, `MySampleINIT` loads the `MyShutDownBeep` code resource into the system heap, installs a pointer to the shutdown code in the shutdown queue, and displays an icon indicating whether the installation succeeded or failed. The `MyShutDownBeep` procedure is executed just before the Macintosh computer shuts down or restarts. For more information about the shutdown process and the Shutdown Manager, see the chapter "Shutdown Manager" in *Inside Macintosh: Processes*.

The code for `MySampleINIT` places the `MyShutDownBeep` procedure in the system heap, making this procedure available after system startup. The `MyShutDownBeep` procedure calls `SysBeep` just before the Macintosh computer shuts down or restarts.

Listing 9-1 The `MySampleINIT` system extension

```
UNIT MySampleINIT    {write a Pascal system extension as a UNIT}

INTERFACE
USES
   Types, Events, Errors, Resources, Memory, Shutdown;
CONST
   kIconIDSuccess = 128;   {icon of this system extension}
   kIconIDFailure = 129;   {icon of this system extension }
                           { with an "X" on it}
   kMyShutDownResourceType = 'SHUT'
   kMyShutDownResourceID = 128;
   moveX = -1;

IMPLEMENTATION
PROCEDURE MyShowINIT(theIcon, moveX: Integer); EXTERNAL;
PROCEDURE MyShutDownBeep; FORWARD;

PROCEDURE MyINIT;
VAR
   theIcon:             Char;
   myShutDownCodeHndl:  Handle;
   myShutDownCodePtr:   ProcPtr;
BEGIN
   theIcon := kIconIDSuccess;
   {retrieve a handle to MyShutDownBeep procedure}
   myShutDownCodeHndl := GetResource(kMyShutDownResourceType,
                                     kMyShutDownResourceID);
   IF ((myShutDownCodeHndl = NIL) OR
      (ResError <> noErr) ) THEN
         theIcon := kIconIDFailed;
```

```
IF (theIcon = kIconIDSuccess) THEN
      BEGIN
         {the MyShutDownBeep code resource is present, detach it}
         { from the resource file and check for an error}
         DetachResource(myShutDownCodeHndl);
         IF (ResError <> noErr) THEN
            theIcon = kIconIDFailed;
         ELSE
            ReleaseResource(myShutDownCodeHndl);
      END;
      IF (theIcon = kIconIDSuccess) THEN
         BEGIN
            MoveHHi(myShutDownCodeHndl);
            HLock(myShutDownCodeHndl);
         END;
      MyShowINIT(theIcon, moveX);{place the icon at boot time}
      {install MyShutDownBeep procedure into shutdown queue}
      myShutDownCodePtr := myShutDownCodeHndl^);
      ShutDwnInstall(myShutDownCodePtr, sdOnUnmount);
END;

PROCEDURE MyShutDownBeep;
BEGIN
   SysBeep(40);
END;

END. {of UNIT}
```

Notice that the code for the `MySampleINIT` extension is defined as a Pascal `UNIT` rather than a `PROGRAM`. This distinction is important because Pascal programs are applications that require an application heap, an initialized A5 register, the Segment Loader, and the services of other Operating System and Toolbox managers. By comparison, a Pascal unit is merely a collection of routines. It does not enjoy the full status of an application. You cannot launch a system extension. It is simply loaded and executed in a temporary heap that the system deallocates soon after the system finishes booting the computer.

When `MySampleINIT` calls the application-defined procedure `MyShowInit`, `MyShowInit` displays an icon on the bottom left of the startup screen, and it does not erase the screen. If you want an icon displayed at system startup time, you must supply this application-defined procedure.

IMPORTANT

If you provide a procedure that displays an icon of your system extension, do no erase the screen. ▲

For information about compiling system extensions, see the section "Building a System Extension" beginning on page 9-17.

Note
System extensions are not well equipped to declare global variables and deal with the A5 world. Stand-alone code modules that do these things are not system extensions and thus are beyond the scope of this discussion. See the chapter "Writing Stand-Alone Code" in *Building and Managing Programs in MPW* for information on this topic. ◆

Because a system extension possesses no A5 world of its own, it cannot easily define global variables: the system allocates no space for them, and the A5 register contains no meaningful value. Extension code that defines global variables usually compiles and links successfully without a warning from the linker; however, the extension's global variables typically overwrite globals defined by the current application.

▲ **WARNING**
Code containing references to global variables defined in the MPW libraries, such as QuickDraw globals, generate fatal link errors. ▲

As a general rule, a system extension can call Operating System managers at any time, but it can call only a few of the Toolbox managers before the startup process completes. It can call the routines from the File Manager, Memory Manager, Resource Manager, and the Notification Manager before the system extension is completely launched, but it must refrain from calling the `InitFonts`, `InitWindows`, `InitDialogs`, `InitMenus` and `TEInit` procedures, as well as other QuickDraw, Window Manager, Dialog Manager, and Font Manager routines. (Note that the code installed by a system extension can utilize the full set of Operating System and Toolbox routines.)

A system extension must do without the services of the Segment Loader, which divides application code into segments that the processor can handle. The size of a system extension's code resource should not exceed 32 KB.

You should consider installing your system extension in the system heap if you want its resources to be available after the computer finishes booting. For example, some system extensions leave routines in the system heap that can be called through patches to those routines. The `MySampleINIT` system extension shown in Listing 1-1 on page 9-11 loads the `MyShutDownBeep` procedure in the system heap.

The procedure your system extension uses to install code in the system heap varies according to what you want to accomplish. Basically, you have to request a block of memory in the system heap and store the code or resources you want to preserve in the block. To allocate memory in the system heap in System 7 and later, you merely need to call the appropriate Memory Manager routines, and the system heap expands dynamically to meet your requests. In earlier versions of system software, you must use a system heap space resource of type '`sysz`' to indicate how much the Operating System should increase the size of the system zone.

See the chapter "Memory Manager" in *Inside Macintosh: Memory* for details on how to allocate memory in the system heap.

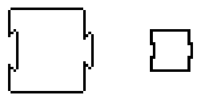
Defining the User Interface for a System Extension

The user interface for a system extension consists of

- the system extension icon
- other elements your system extension needs to communicate with the user

You should provide an icon for the file that contains your system extension. An extension icon looks like a puzzle piece. Figure 9-1 illustrates the default icon for a system extension that appears in the Finder if you don't supply a custom icon for your system extension. You can customize an extension icon by adding a graphic to the default icon. You can display the system extension icon in a horizontal or vertical orientation with the protruding part facing any direction. If you do add graphics, keep them simple so that the icon still looks good when scaled to the small, 16-by-16 pixel icon size.

Figure 9-1 The default system extension icon



The code in your system extension should also display the icon for your system extension when it is first executed at system startup time. You typically display this icon near the bottom-left corner of the startup screen. If the code installed by your extension requires resources or hardware that is not available at system startup, your extension can instead display a crossed-out version of the system extensions icon in the bottom-left corner of the screen.

You should design a system extension so that a user can install it by dragging the icon on top of the System Folder. The Finder then asks the user whether to place the system extension in the Extensions folder. Do not install system extensions in the System file.

When designing a system extension, avoid displaying dialog or alert boxes that interrupt system booting. Whenever possible, use the Notification Manager to notify users of important messages. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for a description on how to send a notification request. You should also avoid calling routines like `InitWindows` that wipe the entire screen clean, obliterating any startup icons that other system extensions and drivers might have displayed.

Your system extension may only create files in the Preferences folder during execution. It is important that your system extension does not create files in the Extensions folder, the Control Panels folder, or the System Folder during execution. The system reads the files in each of these folders sequentially. Creating an additional file in one of these folders shifts the location of the other files, causing the system to either skip a system extension or execute one twice.

If your system extension requires a user interface, you can also create a control panel. If you use a system extension with your control panel, include it in the control panel file along with the required resources and any other optional resources you use. In System 7, system extensions can be installed in the Control Panels folder or in the Extensions folder (both of which are stored in the System Folder) or directly in the System Folder. However, if it contains a system extension, your control panel file must reside in the Controls Panels folder within the System Folder. At startup time, the system software opens files of type 'cdev' that reside in the Control Panels folder and executes any system extensions that it finds there. If the system extension portion of a control panel is not loaded at startup, the control panel won't function properly. For additional information about control panels, see the chapter Control Panels in *Inside Macintosh: More Macintosh Toolbox*.

Creating a System Extension's Resources

A file comprising a system extension contains a resource of type 'INIT' and additional resources. A resource of type 'INIT' contains the code that loads the system-level service into the system at system startup time, and it often contains the code that provides the system-level service. You can use additional resources to describe the icons for the system extension, specify a version number and copyright information for the information window displayed by the Get Info command, increase the size of the system heap, and more.

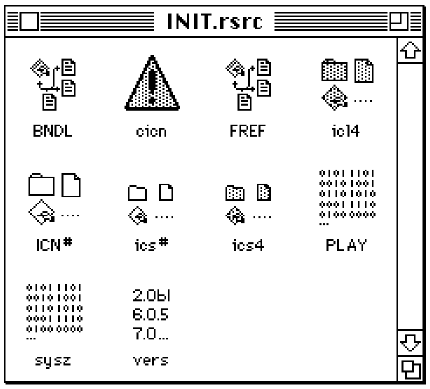
This list describes some of the additional resources you typically use when you create a system extension:

- The version ('vers') resource, which you can use to record version information for your system extension. The version resource allows you to store a version number, a version message, and a region code.
- The bundle ('BNDL') resource, which groups together your system extension's icons.
- Icon family resources ('ICN#', 'ics#', ic18', 'ic14', ics8', and 'ics4') to represent your system extension in the Finder.
- The system heap space ('sysz') resource.

The 'sysz' resource is described in this section. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information about the other resources mentioned in this section.

Figure 9-2 shows a ResEdit window containing additional resources for a system extension. These additional resources can be compiled with an 'INIT' resource into a system extension that goes in the Extensions folder.

Figure 9-2 Typical resources for a system extension



Not all of the resources in Figure 9-2 are required for all system extensions, but they do add useful features to a system extension.

Note
You can use a high-level tool such as the ResEdit application, which is available through APDA, to create your resources. See *ResEdit Reference* for details on using ResEdit. ◆

Creating Icons for a System Extension

You should provide two sets of icons for your system extension:

- an icon family for the file that contains your system extension
- an icon that your system extension displays at system startup time. This icon indicate whether the installation succeeded or failed

You should provide icon family resources for the file that contains your system extension. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a detailed description of the icon family resources.

You can create a color icon resource of type '`cicn`' for your system extension if you want to display a color startup icon at the bottom left of the screen. You can implement this feature by creating your own application-defined `MyShowINIT` procedure, or you can use a similar program called `ShowInit`. You can obtain the `ShowInit` program from various on-line services. (You can also contact APDA for further developer product information). To use `ShowINIT`, you pass the resource ID of your system extension's '`cicn`' resource to the `ShowINIT` procedure, and `ShowINIT` displays the '`cicn`' icon on the bottom-left corner of the screen.

Creating a System Heap Zone Resource for a System Extension

You should read the information in this section only if you plan to install code from your system extension into the system heap and run your system extension on system software prior to System 7.

If you install code in the system heap and run your system extension on system software prior to System 7, you should include a system heap space resource of type 'sysz'. The 'sysz' resource tells the system software the amount of memory the system heap needs to expand by, in order to accommodate space for code installed by your system extension.

Note
It is not necessary to include a 'sysz' resource for system extensions running only on System 7 and later. The system heap in System 7 grows dynamically and expands as long as there is any unused RAM available. ◆

Using a 'sysz' resource, you can request the system software to increase the memory in the system heap by the amount specified in the 'sysz' resource. If the system software is able to allocate the needed memory in the system heap, your system extension will execute. If the system is unable to allocate the extra memory to the system heap, your system extension will not be able to execute.

To create a 'sysz' resource, you can use an editor like the ResEdit application. Specify, in bytes, the amount of memory you want the system heap to increase by. For example, if your system extension takes 8 KB to execute, you should increase the system heap by that amount.

You do not need to allocate memory for the actual system extension code ('INIT' resource), only for the amount of memory for any code installed by your system extension needs to execute.

Building a System Extension

Once you have created a file containing the 'INIT' resource and a file containing all the additional resources, you can build your system extension. To build a system extension, compile and link the 'INIT' resource and the additional resources into an executable file for your system extension.

When you compile the 'INIT' resource and your additional resources, you should keep the following points in mind:

- Make sure that the file type of the system extension is of type 'INIT'.
- Specify a creator if you want the Finder to use icons for your system extension.
- Specify the resource type 'INIT' and a resource ID (usually 128).
- Specify the main entry point for your system extension. When written in Pascal, the main entry point of a module is the first written instruction.
- Specify that the 'INIT' resource be loaded into the system heap if you want its resources to be available after the computer finishes booting.
- Specify the 'INIT' resource (code resource) as locked to prevent the system from moving the resource during execution.
- Make sure that all additional resources are unlocked and purgeable.

Start Manager Reference

This section describes the data structures and routines that are specific to the Start Manager. The “Data Structures” section explains the data structures for the default startup device parameter block, the default video device parameter block, and the default operating system record. The “Routines” section describes routines that get information about and set devices or values that the system uses as defaults when booting a Macintosh computer.

Data Structures

This section describes the data structures that you use to provide information to the Start Manager or the Start Manager uses to return information to your application.

The Default Startup Device Parameter Block

Two procedures, `GetDefaultStartup` and `SetDefaultStartup`, use the default startup device parameter block. You can use these procedures and the default startup device parameter block to get or set the default startup device. As defined by the `DefStartType` data type, a startup device is either a slot or a SCSI device. The `DefStartRec` data type defines the default startup device parameter block.

```
TYPE  DefStartType = (slotDev, scsiDev);

      DefStartRec =
RECORD
   CASE DefStartType OF
      slotDev:
         sdExtDevID: SignedByte; {external device ID}
         sdPartition:SignedByte; {reserved}
         sdSlotNum:  SignedByte; {slot number}
         sdSRsrcID:  SignedByte; {SResourceID}
      scsiDev:
         sdReserved1:SignedByte; {reserved}
         sdReserved2:SignedByte; {reserved}
         sdRefNum:   Integer;    {driver reference number}
END;
DefStartPtr = ^DefStartRec;
```

Field descriptions

`sdExtDevID`	The external device ID specified by a slot's driver. This ID identifies one of perhaps several devices connected through a single slot.
`sdPartition`	Reserved.
`sdSlotNum`	A number that identifies the location of the NuBus slot containing the default startup card. (Currently, these numbers range from $9 through $E on six-slot computers.)
`sdSRsrcID`	The resource ID (`SResourceID`) for the slot.
`sdReserved1`	Reserved.
`sdReserved2`	Reserved.
`sdRefNum`	A negative value in this field indicates the driver reference number for a SCSI device. A positive number indicates a slot device, in which case the fields in the `slotDev` variant.

The Default Video Device Parameter Block

Two procedures, `GetVideoDefault` and `SetVideoDefault`, use the default video device parameter block. You can use these procedures with the default video device parameter block to get or set the default video device. The `DefVideoRec` data type defines the default video device parameter block.

```
TYPE  DefVideoRec =
RECORD
   sdSlot:     SignedByte; {slot number}
   sdsResource:SignedByte; {SResourceID}
END;
DefVideoPtr = ^DefVideoRec;
```

Field descriptions

`sdSlot`	The physical slot number for the default video device. A value of 0 indicates no video device is the default.
`sdSResource`	The slot resource ID (`SResourceID`) for the default video device.

The Default Operating System Parameter Block

Two procedures, `GetDefaultOS` and `SetDefaultOS`, use the default operating system parameter block. You can use these procedures with the default operating system parameter block to get or set the default operating system. The `DefOSRec` data type defines the default operating system parameter block.

```
TYPE  DefOSRec =
RECORD
   sdReserved: SignedByte; {reserved}
   sdOSType:   SignedByte; {operating-system type}
END;
DefOSPtr = ^DefOSRec;
```

Field descriptions

`sdReserved`	Reserved.
`sdOSType`	A value identifying the operating system installed at startup. A 1 indicates the Macintosh Operating System. The numbers 0 through 15 are reserved.

Routines

This section describes the Start Manager routines you can use to identify and change the default startup device, the default video device, default operating system, and the default timeout value for the startup drive.

Many Start Manager routines specify a pointer to a parameter block as a parameter. For these routines, the routine description includes a list of the fields in the parameter block used by the routine. For each routine that uses a parameter block, information about the fields appears in the following format:

Parameter block

→	`input1`	`LongInt`	Input parameter comment.
←	`output1`	`LongInt`	Output parameter comment.

The arrow on the far left indicates whether the field is an input or output parameter. You must supply values for all input parameters. The routine returns values in the output parameters. The next column shows the field name as defined in the MPW interface files, followed by the data type of that field. This matches the MPW interface name of the data type as shown in the parameter block. The fourth column contains a comment about or a brief definition of the field.

Identifying and Setting the Default Startup Device

You can use the routines in this section to get information that identifies the default startup device or to supply information that sets a default startup device. These routines provide applications with the same capability that the Startup Disk control panel supplies for Macintosh users.

GetDefaultStartup

You can use the `GetDefaultStartup` procedure to return information about the default startup device.

```
PROCEDURE  GetDefaultStartup (paramBlock: DefStartPtr);
```

`paramBlock` A pointer to a default startup device parameter block.

Parameter block

←	`sdExtDevID`	`SignedByte`	External device ID.
←	`sdPartition`	`SignedByte`	Reserved.
←	`sdSlotNum`	`SignedByte`	Physical slot number.
←	`sdSRsrcID`	`SignedByte`	Slot resource ID (SResourceID).
←	`sdReserved1`	`SignedByte`	Reserved.
←	`sdReserved2`	`SignedByte`	Reserved.
←	`sdRefNum`	`Integer`	Driver reference number.

DESCRIPTION

The `GetDefaultStartup` procedure returns information about the default startup device from parameter RAM. The default startup device parameter block of data type `DefStartType` defines two kinds of startup devices: either a slot or a SCSI device. The `GetDefaultStartup` procedure returns in the `sdRefNum` field a value indicating the startup device type. A negative value indicates a SCSI device. A positive value indicates a slot device. If the value is negative, the `sdRefNum` field contains the driver reference number needed to identify that device. If the value is positive, the `slotDev` variant of the default startup device parameter block contains information about the slot device.

You cannot read the system's default startup device parameter block directly. Instead, create another parameter block to which the `GetDefaultStartup` procedure can write and pass `GetDefaultStartup` a pointer to that parameter block.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0	Address of the default startup device parameter block

Registers on exit

A0	Address of the default startup device parameter block

SEE ALSO

For more information about the default startup device parameter block see "The Default Startup Device Parameter Block" beginning on page 9-18.
To specify the default startup device, see the description of the `SetDefaultStartup` procedure described next.

SetDefaultStartup

You can use the `SetDefaultStartup` procedure to write information to parameter RAM that specifies the default startup device.

```
PROCEDURE SetDefaultStartup (paramBlock: DefStartPtr);
```

`paramBlock` A pointer to a default startup device parameter block.

Parameter block for a slot device

→	`sdExtDevID`	`SignedByte`	External device ID.
→	`sdPartition`	`SignedByte`	Reserved.
→	`sdSlotNum`	`SignedByte`	Physical slot number.
→	`sdSRsrcID`	`SignedByte`	Slot resource ID (`SResourceID`).

Parameter block for a SCSI device

→	`sdReserved1`	`SignedByte`	Reserved.
→	`sdReserved2`	`SignedByte`	Reserved.
→	`sdRefNum`	`Integer`	Driver reference number.

DESCRIPTION

The `SetDefaultStartup` procedure writes information to parameter RAM that specifies the default startup device. The default startup parameter block of data type `DefStartType` defines two kinds of startup devices: either a slot or a SCSI device. To specify a slot device as the default, pass the external device ID, the slot number, and the slot resource ID. The external device ID, supplied by the slot's driver, identifies a particular device connected through that slot. It's possible that the card in this slot could have several devices connected to it.

To specify a SCSI device as the default, pass its driver reference number (always negative) in the `sdRefNum` field. To specify no device as the default, pass a value of 0 in this field.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0	Address of the default startup device parameter block

Registers on exit

A0	Address of the default startup device parameter block

SEE ALSO

For more information about the default startup device parameter block see "The Default Startup Device Parameter Block" beginning on page 9-18.

To retrieve information about the default startup device, see the description of the `GetDefaultStartup` procedure described on page 9-20.

Identifying and Setting the Default Video Device

You can use the routines in this section to get information about the default video device or to supply information that sets or changes a default video device. These routines provide applications with the same capability that the Monitors control panel supplies for Macintosh users. The default video device is equivalent to the monitor that displays the startup message "Welcome to Macintosh" as well as other startup indications.

GetVideoDefault

You can use the `GetVideoDefault` procedure to return information that identifies the default video device.

```
PROCEDURE GetVideoDefault (paramBlock: DefVideoPtr);
```

`paramBlock` A pointer to a default video device parameter block.

Parameter block

←	`sdSlot`	`SignedByte`	Physical slot number.
←	`sdSResource`	`SignedByte`	Slot resource ID (SResourceID).

DESCRIPTION

The `GetVideoDefault` procedure returns information from parameter RAM that identifies the default video device. If the `sdSlot` field returns a 0, indicating no default video device, the Start Manager chooses the first available video device when the computer starts up.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0	Address of the default video device parameter block

Registers on exit

A0	Address of the default video device parameter block

SEE ALSO

For more information about the default startup device parameter block see "The Default Video Device Parameter Block" beginning on page 9-19.

To specify the default video device, see the description of the `SetVideoDefault` procedure described next.

SetVideoDefault

You can use the `SetVideoDefault` procedure to write information to parameter RAM that sets or changes the default video device.

```
PROCEDURE SetVideoDefault (paramBlock: DefVideoPtr);
```

`paramBlock` A pointer to a default video device parameter block.

Parameter block

→	`sdSlot`	`SignedByte`	Physical slot number.
→	`sdSResource`	`SignedByte`	Slot resource ID (SResourceID).

DESCRIPTION

The `SetVideoDefault` procedure writes information to parameter RAM that sets or changes the default video device. If you set the `sdSlot` field to 0, indicating no default video device, the Start Manager chooses the first available video device when the computer starts up.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0	Address of the default video device parameter block

Registers on exit

A0	Address of the default video device parameter block

SEE ALSO

For more information about the default video device parameter block see "The Default Video Device Parameter Block" beginning on page 9-19.

To retrieve information about the default video device, see the description of the `GetVideoDefault` procedure on page 9-23.

Identifying and Setting the Default Operating System

You can use the routines in this section to get information about the default operating system or to supply information that sets or changes a default operating system. These routines read from and write to a byte in parameter RAM.

GetOSDefault

You can use the `GetOSDefault` procedure to identify the operating system that gets booted on the Macintosh computer.

```
Procedure GetOSDefault (paramBlock: DefOSPtr);
```

`paramBlock` A pointer to a default operating system parameter block.

Parameter block

←	`sdReserved`	`byte`	Reserved.
←	`sdOSType`	`byte`	Operating-system type.

DESCRIPTION

The `GetOSDefault` procedure identifies the operating system that gets booted on the Macintosh computer. A value of 1 returned in the `sdOSType` field indicates the Macintosh Operating System. Apple Computer, Inc. reserves the numbers 0 through 15 for its use.

When the Macintosh Operating System boots, certain startup routines call `GetOSDefault` and compare the value it returns with the value in the `ddType` field of the driver's portion of the driver descriptor record. Each driver for the startup device has its own block of fields in this record. The startup routine tries to match the operating-system type returned by `GetOSDefault` with the value in one of the `ddType` fields. If it finds a match, the computer continues to boot; if it doesn't, the startup routine searches other drives attached to the computer. The boot process does not continue until the startup routine finds a `ddType` value that matches the one returned by `GetOSDefault`.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0	Address of the default operating system parameter block

Registers on exit

A0	Address of the default operating system parameter block

SEE ALSO

For more information about the default operating system parameter block, see "The Default Operating System Parameter Block" beginning on page 9-19.
For information about the driver descriptor record, see the chapter "SCSI Manager" in *Inside Macintosh: Devices*.

To specify the default operating system, see the description of the `SetOSDefault` procedure described next.

SetOSDefault

You can use the `SetOSDefault` procedure to set a byte in parameter RAM that indicates the operating system that gets booted on the Macintosh computer.

```
PROCEDURE SetOSDefault (paramBlock: DefOSPtr);
```

`paramBlock`	A pointer to a default operating system parameter block.

Parameter block

→	`sdReserved`	`SignedByte`	Reserved.
→	`sdOSType`	`SignedByte`	Operating-system type.

DESCRIPTION

The `SetOSDefault` procedure sets a byte in parameter RAM that indicates the operating system that gets booted on the Macintosh computer. Setting a value of 1 in the `sdOSType` field indicates the Macintosh Operating System, which is currently the only default operating system allowed. The numbers 0 through 15 are reserved by Apple Computer.

Unless the value in the `sdOSType` field matches the value in one of the `ddType` fields of the driver descriptor record, the computer cannot continue booting. Every drive connected to the computer has a driver descriptor record at the beginning of physical block 0.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

A0	Address of the parameter block for the default operating system record

Registers on exit

A0	Address of the parameter block for the default operating system record

SEE ALSO

For information about the driver descriptor record, see the chapter "SCSI Manager" in *Inside Macintosh: Devices*.

Getting and Setting the Timeout Interval

You can use the routines in this section to get or set the default timeout interval for the startup drive. This timeout indicates how long the system waits for the startup drive to respond while the computer is booting.

GetTimeout

You can use the `GetTimeout` procedure to identify the current timeout interval set for the startup drive.

```
PROCEDURE GetTimeout (VAR count: Integer);
```

`count`	Indicates the number of seconds the system waits for the startup drive to respond during the boot cycle. A value of 0 indicates the default timeout of 20 seconds.

DESCRIPTION

The `GetTimeout` procedure identifies the current timeout interval set for the startup drive. Timeout values increment in 1-second intervals, from 1 to a maximum of 31 seconds. A `count` of 1 equals 1 second.

ASSEMBLY LANGUAGE INFORMATION

The register on exit from the routine is

Registers on exit

A0	Value of `count` field

The `_GetTimeout` macro expands to invoke another trap macro, whose routine selector is passed in the A0 register.

Trap Macro	**Selector**
`_InternalWait`	$0000

SetTimeout

You can use the `SetTimeout` procedure to set the timeout interval for the startup drive.

```
PROCEDURE SetTimeout (count: Integer);
```

`count`	Indicates the number of seconds that you want the system to wait for the startup drive to respond during the boot cycle. A value of 0 indicates the default timeout of 20 seconds. The maximum value is 31 seconds.

DESCRIPTION

The `SetTimeout` procedure sets the timeout interval for the startup drive. Timeout values increment in 1-second intervals, from 1 to a maximum of 31 seconds. Setting the `count` parameter to a value of 1 indicates 1 second.

ASSEMBLY LANGUAGE INFORMATION

The registers on entry for this routine are

Registers on entry

A0	$0001

The `_SetTimeout` macro expands to invoke another trap macro, whose routine selector is passed in the A0 register:

Trap Macro	Selector
`_InternalWait`	$0001

Summary of the Start Manager

Pascal Summary

Data Types

```
TYPE
   DefStartType = (slotDev, scsiDev);

   DefStartRec =
   RECORD
      CASE DefStartType OF
      slotDev:
         sdExtDevID: SignedByte;    {external device ID}
         sdPartition:SignedByte;    {reserved}
         sdSlotNum:  SignedByte;    {slot number}
         sdSRsrcID:  SignedByte;    {SResourceID}
      scsiDev:
         sdReserved1:SignedByte;    {reserved}
         sdReserved2:SignedByte;    {reserved}
         sdRefNum:   Integer        {driver reference number}
   END;

   DefStartPtr = ^DefStartRec;      {pointer to a start definition record}

   DefVideoRec =
   RECORD
      sdSlot:     SignedByte;       {slot number}
      sdsResource:SignedByte;       {SResourceID}
   END;

   DefVideoPtr = ^DefVideoRec;   {pointer to a video definition record}

   DefOSRec    =
   RECORD
      sdReserved: SignedByte;    {reserved--should be 0}
      sdOSType:   SignedByte;    {operating-system type}
   END;

   DefOSPtr    = ^DefOSRec;   {pointer to a default Operating System Record}
```

Routines

Identifying and Setting the Default Startup Device

```
PROCEDURE GetDefaultStartup (paramBlock: DefStartPtr);
PROCEDURE SetDefaultStartup (paramBlock: DefStartPtr);
```

Identifying and Setting the Default Video Device

```
PROCEDURE GetVideoDefault   (paramBlock: DefVideoPtr);
PROCEDURE SetVideoDefault   (paramBlock: DefVideoPtr);
```

Identifying and Setting the Default Operating System

```
PROCEDURE GetOSDefault      (paramBlock: DefOSPtr);
PROCEDURE SetOSDefault      (paramBlock: DefOSPtr);
```

Getting and Setting the Timeout Interval

```
PROCEDURE GetTimeout        (VAR count: Integer);
PROCEDURE SetTimeout        (count: Integer);
```

C Summary

Data Types

```
struct SlotDev {
   char sdExtDevId;     /*external device ID*/
   char sdPartition;    /*reserved*/
   char sdSlotNum;      /*slot number*/
   char sdSRsrcID;      /*SResourceID*/
};

typedef struct SlotDev SlotDev;

struct SCSIDev {
   char sdReserved1;    /*reserved*/
   char sdReserved2;    /*reserved*/
   short sdRefNum;      /*driver reference number*/
};

typedef struct SCSIDev SCSIDev;

union DefStartRec {
   SlotDev slotDev;
   SCSIDev scsiDev;
};

typedef union DefStartRec DefStartRec;
typedef DefStartRec *DefStartPtr;

struct DefVideoRec {
   char sdSlot;      /*slot number*/
   char sdsResource; /*SResourceID*/
};

typedef struct DefVideoRec DefVideoRec;
typedef DefVideoRec *DefVideoPtr;

struct DefOSRec {
   char sdReserved;  /*reserved —should be 0*/
   char sdOSType;    /*operating-system type*/
};

typedef struct DefOSRec DefOSRec;
typedef DefOSRec *DefOSPtr;
```

Routines

Identifying and Setting the Default Startup Device

```
pascal void GetDefaultStartup (DefStartPtr paramBlock);
pascal void SetDefaultStartup (DefStartPtr paramBlock);
```

Identifying and Setting the Default Video Device

```
pascal void GetVideoDefault  (DefVideoPtr paramBlock);
pascal void SetVideoDefault  (DefVideoPtr paramBlock);
```

Identifying and Setting the Default Operating System

```
pascal void GetOSDefault    (DefOSPtr paramBlock);
pascal void SetOSDefault    (DefOSPtr paramBlock);
```

Getting and Setting the Timeout Interval

```
pascal void GetTimeout      (short *count);
pascal void SetTimeout      (short count);
```

Assembly-Language Summary

Data Structures

Default Startup Device Data Structure

0	`sdExtDevID`	byte	external device ID
1	`sdPartition`	byte	reserved
2	`sdSlotNum`	byte	slot number
3	`sdSRsrcID`	byte	slot resource ID

0	`sdReserved1`	byte	reserved
1	`sdReserved2`	byte	reserved
2	`sdRefNum`	word	driver reference number

Default Video Device Data Structure

0	`sdSlot`	byte	slot number
1	`sdSResource`	byte	slot resource ID

Default Operating System Data Structure

0	`sdReserved`	byte	reserved
1	`sdOSType`	byte	operating-system type

Trap Macros

Trap Macros Requiring Register Setup

Trap macro name	Registers on entry	Registers on exit
`_GetDefaultStartup`	A0: address of default video device parameter block	A0: address of default startup device parameter block
`_SetDefaultStartup`	A0: address of default video device parameter block	A0: address of default startup device parameter block
`_GetVideoDefault`	A0: address of default video device parameter block	A0: address of default video device parameter block
`_SetVideoDefault`	A0: address of default video device parameter block	A0: address of default video device parameter block
`_GetDefaultOS`	A0: address of default operating system parameter block	A0: address of default operating system parameter block
`_SetDefaultOS`	A0: address of default operating system parameter block	A0: address of default operating system parameter block
`_GetTimeout`		D0: count (word)
`_SetTimeout`	D0: count (word)	

Trap Macros Requiring Routine Selectors

`_InternalWait`

Selector	Routine
$0000	`GetTimeout`
$0001	`SetTimeout`

Global Variables

`TimeDBRA`	The number of times the DBRA instruction is executed per millisecond.
`TimeSCCDB`	The number of times the SCC is accessed per millisecond.
`TimeSCSIDB`	The number of times the SCSI is accessed per millisecond.