Developer Note

Macintosh Quadra 840AV and Macintosh Centris 660AV Computers

Contents

Part 2   Real-Time Data Processing   57

Chapter 3   Introduction to Real-Time Data Processing   59

Chapter 4   Real Time Manager   123

Chapter 5   DSP Operating System   203

Chapter 7   Introduction to Speech Recognition   317

Chapter 8   Speech Rules   335

Figures, Tables, and Listings

Chapter 3

Introduction to Real-Time Data Processing   59

Chapter 4

Real Time Manager   123

Chapter 5

DSP Operating System   203

About This Developer Note

This developer note introduces the Macintosh Quadra 840AV and the Macintosh Centris 660AV, Apple's newest extensions to the Macintosh family of personal computers. It is written primarily for experienced Macintosh hardware and software engineers who want to create products that are compatible with these computers.

This note assumes that you are already familiar with both the functionality and programming requirements of Macintosh computers. If you are unfamiliar with Macintosh computers or would like more technical information, you may want to obtain copies of the related technical manuals listed in "Supplementary Documents," later in this preface.

Contents of This Note

This developer note is divided into four main parts, containing a total of 13 chapters.

Part 1, "Hardware," describes the Macintosh Quadra 840AV and Macintosh Centris 660AV computers from a hardware viewpoint. It contains two chapters:

■ Chapter 1, "The Macintosh Quadra 840AV and Macintosh Centris 660AV Computers," gives you an overview of the configurations and features of these products.

■ Chapter 2, "Hardware Details," describes the circuit boards for the computers, including their physical layout, functional units, signal timing and other electronic characteristics, input and output connectors, and interfaces with other equipment.

Part 2, "Real-Time Data Processing," describes the software technology of the digital signal processing (DSP) facilities in the Macintosh Quadra 840AV and Macintosh Centris 660AV. It contains three chapters:

■ Chapter 3, "Introduction to Real-Time Data Processing," summarizes the software architecture of their real-time data processing facility. This facility consists of an AT&T DSP3210 chip that performs data-processing operations for applications that contain DSP code.

■ Chapter 4, "Real Time Manager," describes a new part of the Macintosh system software that supplies all the services an application requires to use the digital signal processor, including loading and running DSP code and performing DSP memory management.

■ Chapter 5, "DSP Operating System," covers the DSP operating system, contained in the DSP chip. It provides the services every DSP program needs to work with the Macintosh Operating System.

Part 3, "Speech Synthesis and Recognition," explains the capabilities of the Macintosh Quadra 840AV and Macintosh Centris 660AV system software for generating and understanding human speech. It contains three chapters:

■ Chapter 6, "Speech Manager," describes a new Macintosh system software manager that provides a standardized way for applications to generate synthesized speech. The Speech Manager also lets an application control one or more speech synthesizers, which generate spoken sound in specific languages, intonations, and speaking styles.

■ Chapter 7, "Introduction to Speech Recognition," contains a basic tutorial for the Speech Setup control panel. This control panel provides commands for controlling the speech recognition function.

■ Chapter 8, "Speech Rules," explains the speech rules that are built into the Macintosh Quadra 840AV and Macintosh Centris 660AV system software.

Part 4, "System Software Modifications," describe miscellaneous changes to the Macintosh Quadra 840AV and Macintosh Centris 660AV system software, including a new manager for the internal and external SCSI (Small Computer System Interface) ports. It contains five chapters:

■ Chapter 9, "SCSI Manager 4.3," describes the new SCSI Manager.

■ Chapter 10, "DMA Serial Driver," details the new hardware-independent serial driver that uses direct memory access (DMA).

■ Chapter 11, "Video Driver," describes changes to the video driver.

■ Chapter 12, "New Age Floppy Disk Driver," lists changes to the floppy disk driver and tells you how they affect floppy disk compatibility with other Macintosh computers.

■ Chapter 13, "Virtual Memory Manager," details how the Virtual Memory Manager no longer disables interrupts when performing certain tasks.

Four appendixes follow the main parts of this note. They contain information that can help you with specific development tasks:

■ Appendix A, "DSP d Commands for MacsBug," describes three new d commands added to Macsbug that help in debugging DSP code.

■ Appendix B, "BugLite User's Guide," describes a DSP module installer with a graphical user interface. It helps programmers create and install tasks to be executed by the DSP.

■ Appendix C, "Snoopy User's Guide," tells you how to use a browser and debugger for the DSP. It helps programmers debug real-time tasks that run on the DSP.

■ Appendix D, "Mechanical Details" contains foldout drawings of the physical mounting facilities that are provided for internal SCSI devices and accessory cards in the Macintosh Quadra 840AV and Macintosh Centris 660AV.

At the end of this developer note are a glossary and an index. Terms listed in the glossary are printed in **boldface** where they are first defined in the text.

Hardware Overview

The Macintosh Quadra 840AV and Macintosh Centris 660AV have the most features of any models in the Macintosh family of desktop computers. The Macintosh Quadra 840AV is also the fastest Macintosh computer. The two models have nearly identical electronic circuitry. Their differences are that the Macintosh Quadra 840AV is housed in a minitower enclosure with more room for internal disk drives and accessory cards, while the Macintosh Centris 660AV is housed in a low-profile enclosure designed to be placed under the user's monitor. The Macintosh Centris 660AV also offers somewhat lower speed and performance than the Macintosh Quadra 840AV and sells for a lower price.

Principal new hardware features of these computers include

■ digital signal processing, using an AT&T DSP3210 chip

■ video input and output facilities in NTSC, PAL, and SECAM formats

■ high-quality sound processing

■ direct memory access for peripheral devices

■ integrated telephone I/O for ISDN, fax, and other signal forms

Chapter 1, "The Macintosh Quadra 840AV and Macintosh Centris 660AV Computers," describes these and other hardware features; Chapter 2, "Hardware Details," provides deeper technical information.

Software Overview

The Macintosh Quadra 840AV and Macintosh Centris 660AV are supplied with essentially identical versions of the Macintosh System 7.1 software, in ROM and on the internal hard disk. For technical information about standard System 7.1 software, see *Inside Macintosh*, listed in "Supplementary Documents," later in this preface. However, the system software in the Macintosh Quadra 840AV and Macintosh Centris 660AV also contains significant changes and additions to System 7.1. This section summarizes those changes and additions, which are described in greater detail in Chapters 3 through 13.

Digital Signal Processing

The Macintosh Quadra 840AV and Macintosh Centris 660AV use a digital signal processor (DSP) chip separate from the main microprocessor to perform real-time data processing, such as playing sound files. In addition, the DSP chip can perform processing-intensive operations that do not require real-time execution, such as file compression and three-dimensional drawing. Chapter 3, "Introduction to Real-Time Data Processing," explains this capability in more detail.

To take advantage of the DSP capability, you must write and compile DSP code and include it with your application. A new addition to the Macintosh system software, the Real Time Manager, supplies the services your application needs to handle DSP code. It contains the calls needed to access the DSP, load and run DSP code, and transfer data to and from the DSP. The Real Time Manager also handles system memory management for the DSP and automatically locks and unlocks memory that is accessed by both DSP and Macintosh software. Chapter 4, "Real Time Manager," describes these functions in detail.

The Real Time Manager coordinates usage of the DSP chip through a new concept called guaranteed processing bandwidth (GPB). This type of control guarantees that any application that is granted access to the DSP will always process the needed data at the time required.

The DSP operating system, contained in the DSP chip, provides the services that DSP code needs to drive the chip and work with the Macintosh Operating System. The DSP operating system's application programming interface (API) defines how you can create a resource that can be loaded and run on the DSP chip. It automatically handles on-chip memory management to minimize recaching of code and data used by more than one DSP code module. For real-time applications, actual GPB requirements are determined and saved automatically in the DSP Preferences file. For timeshare applications, the program context is automatically saved when execution switches to real-time code.

The DSP operating system provides a run-time environment for the DSP code modules that minimizes programmer difficulties while providing robust support for a variety of tasks. Caching and saving of data and variables can be handled by the DSP operating system or can be explicitly controlled by the programmer. You can exercise complete control over running DSP code or, by setting a counter, can cause code execution to be determined dynamically at run time. For further information, see Chapter 5, "DSP Operating System."

Text-to-Speech Conversion

A new Macintosh manager, the Speech Manager, provides a standardized API for applications to generate synthesized speech. A single call provides simple text-to-speech operation. Other API calls provide more detailed speech

features. Word pronunciation can also be defined by means of embedded commands within the text string being spoken. Chapter 6, "Speech Manager," provides full details of these new capabilities.

Speech Recognition

A new Macintosh system feature, the Speech Recognition Control Panel, provides start, stop, and parameter setup commands for controlling speech recognition behavior. In the Macintosh Quadra 840AV and Macintosh Centris 660AV, built-in speech recognition is provided for many Macintosh system operations. The standard File and Edit menu commands are fully supported, as are the Finder operations. For example, opening the Control Panels folder is as easy as saying *open control panels.*

User-defined speech macros let you customize the speech recognition software to recognize application-specific speech input for performing common tasks. A 60,000-word dictionary allows selection from a wide variety of words to define spoken phrases that trigger user-defined operations by means of speech macros.

Chapter 7, "Introduction to Speech Recognition," describes the current user controls for speech command in the Macintosh Quadra 840AV and Macintosh Centris 660AV. Chapter 8, "Speech Rules," provides information about further programmable controls for speech recognition.

New SCSI Manager

SCSI Manager 4.3 incorporates a new multilevel architecture that affects all modes of Small Computer System Interface (SCSI) operation. It uses a parameter-block-based programming interface for executing SCSI input-output (I/O) requests, which contains all the information required to complete each I/O operation. The new architecture provides a hardware-independent interface to the SCSI Manager. The SCSI driver layer passes the hardware support it provides to the SCSI Manager for complete hardware support. The SCSI Manager follows the phases driven by the target and eliminates the need to track the SCSI bus phases.

SCSI Manager 4.3 supports SCSI connect/disconnect, parity transmission and parity error detection, all SCSI-2 mandatory messages, SCSI Fast or Wide, and autosense. The SCSI DMA supports asynchronous protocols using both multiple bus and multiple logic units on each target.

Besides supporting these new features, SCSI Manager 4.3 also supports the existing SCSI device drivers with little or no modification. However, you should evaluate your existing drivers for compatibility and incorporate the new features where possible. See Chapter 9, "SCSI Manager 4.3," for further details.

Other System Software Changes

The DMA serial driver was completely rewritten internally. However, there are no API changes. The major operational changes affect interrupt handling and DMA versus non-DMA transmissions, elimination of the `PollProc` mechanism, and use of the new DMA chip. For technical details, see Chapter 10, "DMA Serial Driver."

The video driver has been modified as a result of new video capabilities. Changes and additions to the video driver are described in Chapter 11, "Video Driver."

The floppy disk driver has been modified to work with the New Age floppy disk drive controller. This has resulted in minor changes to the floppy disk drive control API, as described in Chapter 12, "New Age Floppy Disk Driver."

The Virtual Memory Manager has been changed so that it no longer disables interrupts when performing certain tasks. These tasks are listed in Chapter 13, "Virtual Memory Manager."

Supplementary Documents

The following documents provide information that complements or extends the information in this developer note:

Apple Computer:

Inside Macintosh is a collection of books, organized by topic, that describe the system software of Macintosh computers. Together, these books provide the essential reference for programmers, software designers, and engineers. Current volumes include the following titles:
Inside Macintosh: Overview
Inside Macintosh: Toolbox Essentials
Inside Macintosh: More Macintosh Toolbox
Inside Macintosh: Files
Inside Macintosh: Processes
Inside Macintosh: Memory
Inside Macintosh: Operating System Utilities
Inside Macintosh: Imaging
Inside Macintosh: Text
Inside Macintosh: Interapplication Communication
Inside Macintosh: Devices
Inside Macintosh: QuickTime
Inside Macintosh: QuickTime Components
Inside Macintosh: Networking

Technical Introduction to the Macintosh Family, second edition, surveys the complete Macintosh family of computers from the developer's point of view.

Macintosh Human Interface Guidelines provides authoritative information on the theory behind the Macintosh "look and feel" and Apple's standard ways of using individual interface components.

Designing Cards and Drivers for the Macintosh Family, third edition, explains the hardware and software requirements for drivers and NuBus '90 accessory cards compatible with Macintosh computers, including the Macintosh Quadra 840AV and Macintosh Centris 660AV.

Technical Note 144 (*Macintosh Color Monitor Connections*) and Technical Note 326 (*M.HW.SenseLines*) provide technical details of the interfaces to various Apple and third-party monitors.

The *NuBus Block Transfers* technical note provides information about block data transfers to and from accessory cards.

Macintosh Classic II, Macintosh PowerBook Family, Macintosh Quadra Family, Macintosh Centris 610, Macintosh Centris 650, and Macintosh Quadra 800 Developer Notes include hardware details for these computers.

The Apple publications listed above are available from APDA, Apple's source for development tools and publications. APDA offers convenient worldwide access to over three hundred Apple and third-party development tools, resources, and information for anyone interested in developing applications on Apple platforms. For a free copy of the *APDA Tools Catalog,* call 1-800-282-2732 (United States), 1-800-637-0029 (Canada), or 716-871-6555 (International).

The following documents are available from the organizations listed:

AT&T:
WEDSP3210 Digital Signal Processor Information Manual

Comité Consultatif International Radio (CCIR):
Recommended Standard 601-2.

IT&T Semiconductors:
ASCO 2300 Audio-Stereo Codec Specification

Motorola:
MC68040 32-Bit Microprocessor User's Manual
MC68040 32-Bit Microprocessor Programmer's Reference Manual
MC68040 32-Bit Microprocessor Designer's Handbook

Phillips:
7169 Video Data Path Chip data sheet
7191B Digital Multistandard Decoder data sheet

Standard Abbreviations

Acronyms and abbreviations that are specific to Macintosh technology are spelled out in the text where they first occur and are listed in the glossary. Other contractions commonly used in the electronics industry are not spelled out. They include the following:

A	amperes	mm	millimeters
cm	centimeters	ms	milliseconds
dB	decibels	mV	millivolts
GB	gigabytes	NC	no connection
KB	kilobytes	ns	nanoseconds
Kbit	kilobits	pF	picofarads
kHz	kilohertz	rms	root mean square
kΩ	kilohms	V	volts
mA	milliamperes	μF	microfarads
MB	megabytes	μs	microseconds
Mbit	megabits	Ω	ohms
MHz	megahertz		

Hardware

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* describes these computers from a hardware viewpoint. It contains two chapters:

■ Chapter 1, "The Macintosh Quadra 840AV and Macintosh Centris 660AV Computers," gives you an overview of the configurations and features of these products.

■ Chapter 2, "Hardware Details," describes the Macintosh Quadra 840AV and Macintosh Centris 660AV circuit boards, including their physical layout, functional units, signal timing and other electronic characteristics, input and output connectors, and interfaces with other equipment.

Other parts of this developer note cover the following topics:

■ Part 2, "Real-Time Data Processing," covers the software technology of the Macintosh Quadra 840AV and Macintosh Centris 660AV DSP facilities.

■ Part 3, "Speech Synthesis and Recognition," explains the capabilities of the Macintosh Quadra 840AV and Macintosh Centris 660AV system software for generating and understanding human speech.

■ Part 4, "System Software Modifications," covers miscellaneous changes to the Macintosh Quadra 840AV and Macintosh Centris 660AV system software, including a new manager for the internal and external SCSI ports.

The Macintosh Quadra 840AV and Macintosh Centris 660AV Computers

The Macintosh Quadra 840AV and Macintosh Centris 660AV represent new advances in the Macintosh family of high-performance desktop computers. Both models contain an input/output (I/O) subsystem that runs independently of the main processor and an independent digital signal processor (DSP) subsystem that supports real-time processing of data. These facilities, combined with a 32-bit MC68040 main processor running at either 40 MHz (the Macintosh Quadra 840AV) or 25 MHz (the Macintosh Centris 660AV), let these computers perform sophisticated manipulation of sound, graphics, video, and analog modem signals.

Both models bring advanced application performance into the mainstream of desktop computing. Potential application features include real-time speech recognition and synthesis, compression and decompression of sound, complex graphics and video processing, widely compatible telephone networking, and three-dimensional graphics rendering. With Apple's mini-videocam (sold separately), both models can also support videophone and desktop photography software. Many of these application features are unattainable on other personal computing platforms.

This chapter describes the computers in general terms and lists some of their features, differences, and compatibility issues.

Models and Accessories

The Macintosh Quadra 840AV and the Macintosh Centris 660AV have many common features. They differ mainly in physical form, speed, and expansion facilities:

- The Macintosh Quadra 840AV is housed in the same minitower configuration as the Macintosh Quadra 800. It runs at 40 MHz and has three NuBus™ slots for expansion cards. The base configuration for this model includes 8 MB of random-access memory (RAM) and a 230 MB internal hard disk.

- The Macintosh Centris 660AV is a less expensive desktop version of the Macintosh Quadra 840AV in the same low-profile enclosure as the Macintosh Centris 610. It runs at 25 MHz and has only one NuBus expansion slot. The base configuration for this model includes 4 MB of RAM and an 80 MB internal hard disk.

The user can expand the RAM capacity of either model by installing expansion cards. The Macintosh Quadra 840AV has an ultimate RAM capacity of 128 MB; the Macintosh Centris 660AV, 68 MB.

Both models include an Apple SuperDrive floppy disk drive, capable of accepting 1.4 MB floppy disks. The Macintosh Quadra 840AV accepts up to three removable SCSI devices inside its case; the Macintosh Centris 660AV can accept two. Besides the base configuration disk drives, internal data storage options include

- a 500 MB internal hard disk

- a 1000 MB internal hard disk (in the Macintosh Quadra 840AV only)

- an internal compact disc read-only memory (CD-ROM) drive

The user can add a variety of external disk drives to either model, using the external SCSI port. For details of the possible SCSI configurations, see "SCSI Connection," in Chapter 2. For information about new SCSI software, see Chapter 9, "SCSI Manager 4.3."

Apple offers the following separate accessories for both models:

- an inexpensive mini-videocam that captures monochrome video action in an image 360 pixels wide by 288 pixels high

- a high-quality microphone specifically designed for speech recording and recognition

- an analog telecom adapter

- an Ethernet cable adapter

Apple will sell the Macintosh Quadra 840AV and Macintosh Centris 660AV in all domestic and international markets for Apple computers.

Summary of Features

Besides standard features common to the Macintosh family of computers, the Macintosh Quadra 840AV and the Macintosh Centris 660AV have these new or improved capabilities:

- *Direct memory access.* A **Peripheral Subsystem Controller (PSC)** provides **direct memory access (DMA)** between the main processor buses and peripheral devices. DMA permits very fast transfers of large amounts of data without burdening the main processor.

- *Widely compatible video input.* Both models can process a YUV 4:2:2 video input from internal accessory cards. These computers can also process composite or S-video inputs in NTSC, PAL, and SECAM formats from external sources, using standard television-type connectors. Both models store video information in RGB form in a video frame buffer separate from main memory.

- *Flexible video output.* Both models supply video signals to the full range of Apple monitors as well as many third-party monitors. These computers also provide NTSC and PAL composite and S-video outputs for other kinds of video equipment. The Macintosh Quadra 840AV supports color depths up to 24 bits for graphics and 16 bits for video; the Macintosh Centris 660AV supports up to 16 bits for both.

- *Digital signal processing.* A built-in digital signal processor provides fast processing of data in real time. The principles of digital signal processing are discussed in Chapter 3, "Introduction to Real-Time Data Processing."

- *Improved NuBus interface.* Both models use the **Macintosh Universal NuBus Interface (MUNI)** for accessory cards. This interface supports block transfers and data bursts to and from the main processor bus. MUNI capability is optional in the Macintosh Centris 660AV.

- *New floppy disk support.* A new controller for the built-in Apple SuperDrive disk drive is based upon Industry Standard 765, supporting both Apple's **Group Code Recording (GCR)** format and DOS-compatible **Modified Frequency Modulation (MFM)** format.

■ *Built-in Ethernet support.* Both models contain built-in circuitry for Ethernet I/O.

■ *Enhanced serial ports.* Two serial ports both support RS-232, RS-422, and AppleTalk I/O and offer improved system performance with LocalTalk networks.

■ *Integrated telephone I/O.* Both models provide Apple's high-performance serial I/O capability (called **GeoPort**) on one serial port, permitting connection to analog, ISDN, facsimile (fax), and data telephone lines. These computers support full-duplex telephone I/O at transmission rates up to 9600 bits per second.

■ *Enhanced sound I/O.* Using the DSP, both models provide 16-bit digital stereo sound I/O at sample rates up to 48 kHz, including the standard rate of 44.1 kHz.

■ *Large-capacity ROM.* Identical ROM chips totaling 2 MB are provided with both models. These chips contain some of the system software that is on the hard disk in other Apple computers.

■ *Expansion slots.* The Macintosh Quadra 840AV contains three NuBus slots for long or short Macintosh expansion cards; the Macintosh Centris 660AV accepts one short card using an optional angle adapter. All slots carry a 32-bit data and address bus; in addition, a **digital audio/video (DAV) expansion connector** in line with one slot gives an accessory card direct access to YUV video data and digital sound.

■ *Mass media support.* Both models support up to a total of seven SCSI devices. Within this limit, the user can connect up to six external devices to either model's SCSI port in addition to internal hard disk drives and CD-ROM drives.

■ *Advanced processor features.* The MC68040 main processor in both models performs 32-bit paged memory management and has internal 4 KB data and instruction caches. The MC68040 processor also performs high-speed, high-accuracy coprocessing of floating-point numeric data.

■ *Software on/off power control.* The Macintosh Quadra 840AV provides power control service to its expansion slots, so plug-in cards can turn the computer on and off.

■ *Replaceable real-time clock battery.* The real-time clock and parameter RAM in both models are powered by a long-life plug-in battery.

■ *Automatic SCSI termination.* Built-in circuitry provides automatic termination of the internal and external SCSI cables if no SCSI devices are connected.

Differences Between Models

Besides the distinctions of speed, physical form, and base memory configuration cited in "Models and Accessories," earlier in this chapter, the Macintosh Quadra 840AV and the Macintosh Centris 660AV contain the following detailed hardware differences:

■ The maximum video window size at a color depth of 16 bits is 640 by 480 pixels for the Macintosh Quadra 840AV and 512 by 384 pixels for the Macintosh Centris 660AV.

■ The Macintosh Quadra 840AV computer's VRAM is expandable from 1 MB to 2 MB; the Macintosh Centris 660AV computer's is not.

- The Macintosh Quadra 840AV supports 21-inch RGB monitors; the Macintosh Centris 660AV does not.

- The Macintosh Quadra 840AV can accept up to three long (4 by 13 inches) or short (4 by 7 inches) NuBus accessory cards; the Macintosh Centris 660AV accepts only one short card, using an optional angle adapter card.

- The Macintosh Quadra 840AV lets a plug-in card turn the computer on and off. The Macintosh Centris 660AV does not.

- The Macintosh Centris 660AV can accept an accessory card that connects directly to the main processor bus; the Macintosh Quadra 840AV cannot.

- The Macintosh Quadra 840AV requires 60 ns DRAM chips; the Macintosh Centris 660AV can use 70 ns DRAM chips.

System Software

The Macintosh Quadra 840AV and the Macintosh Centris 660AV contain the same ROM and support the same set of system calls, instructions, and data structures (the same **application programming interface**, or **API**). Hence, an application that runs on one model will run on the other. However, the system software API includes access to several new managers in addition to the familiar Macintosh Toolbox. Applications that are written to use these managers can offer valuable new features and capabilities.

For complete information about the system software features that are new with the Macintosh Quadra 840AV and the Macintosh Centris 660AV, see Chapters 3 through 13.

Compatibility Issues

Products that are designed for other computers in the Macintosh family and are compatible with Macintosh system software release 7.0 or 7.1 should work with the Macintosh Quadra 840AV and the Macintosh Centris 660AV, provided they follow Apple's current design guidelines (such as being 32-bit clean). Of course, such products normally will not take advantage of the unique capabilities of these new computers.

Developers who want to design new products that make use of the features of the Macintosh Quadra 840AV and the Macintosh Centris 660AV, while remaining compatible with other Apple computers, should keep these compatibility pointers in mind:

- Use only system API calls to access hardware; never try to modify or program the serial, SCSI, ADB, sound, or video subsystems directly.

- Never change the processor status register directly.

- Do not disable interrupts for longer than 0.5 ms.

- Do not assume any fixed addresses for global variables or ROM routines. You can use the `GetTrapAddress` and `SetTrapAddress` functions to get and set these addresses.

Machine Identification

By using the Gestalt Manager and the `SysEnvirons` function, an application can determine which features exist on the user's system. Table 1-1 lists the relevant machine identification values.

Table 1-1   Gestalt values for the Macintosh Quadra 840AV and Macintosh Centris 660AV

Identifier	Value	Description
SysEnvirons	78	Macintosh Quadra 840AV processor (40 MHz)
SysEnvirons	60	Macintosh Centris 660AV processor (25 MHz)
gestaltNuBusSlotCount	'nubs'	Count of logical NuBus slots
gestaltSlotAttr	'slot'	Slot attributes
gestaltSlotMgrExists	0	True if Slot Manager exists
gestaltNuBusPresent	1	NuBus slots are present
gestaltSESlotPresent	2	SE PDS slot present
gestaltSE30SlotPresent	3	SE/30 slot present
gestaltPortableSlotPresent	4	Portable's slot present
gestaltFirstSlotNumber	'slt1'	Returns first physical slot
gestaltIconUtilities	'icon'	Icon utilities attributes
gestaltIconUtilitiesPresent	0	Icon utilities present
gestaltRealtimeMgrAttr	'rtmr'	Real Time Manager attributes
gestaltRealtimeMgrPresent	0	True if Real Time Manager present
gestaltSoundHardware	'snhw'	Get the sound hardware
gestaltASC	'asc '	Component type of sound chip
gestaltDSP	'dsp '	Component type of DSP
gestaltClassicSound	'clas'	Macintosh Classic® sound
gestaltVIA1Addr	'via1'	VIA-1 base address
gestaltVIA2Addr	'via2'	VIA-2 base address
gestaltVMAttr	'vm '	Virtual memory attributes
gestaltVMPresent	0	True if virtual memory present
gestaltVMNotInstalled	0	True if virtual memory not installed

Hardware Details

The Macintosh Quadra 840AV and Macintosh Centris 660AV computers are shipped with built-in floppy disk drives and removable internal hard disk drives. For each model, the addition of an external monitor, a keyboard, and a mouse forms a complete personal computing system.

This chapter provides details of the Macintosh Quadra 840AV and Macintosh Centris 660AV physical equipment.

Physical Forms

The Macintosh Quadra 840AV and Macintosh Centris 660AV computers are generally described in "Models and Accessories," in Chapter 1. The external dimensions of their enclosures are shown in Table 2-1.

Table 2-1   External dimensions

Dimension	Macintosh Quadra 840AV	Macintosh Centris 660AV
Width	7.8 in. (19.7 cm)	16.3 in. (41.4 cm)
Depth	16.0 in. (40.6 cm)	14.8 in. (37.6 cm)
Height	14.3 in. (36.2 cm)	3.2 in. (8.1 cm)

Both models contain essentially the same circuit board and system components, with variations as noted in this chapter. Drawings of certain Macintosh Quadra 840AV and Macintosh Centris 660AV parts that support the mounting of peripheral devices are given in Appendix D, "Mechanical Details."

Parts Layout
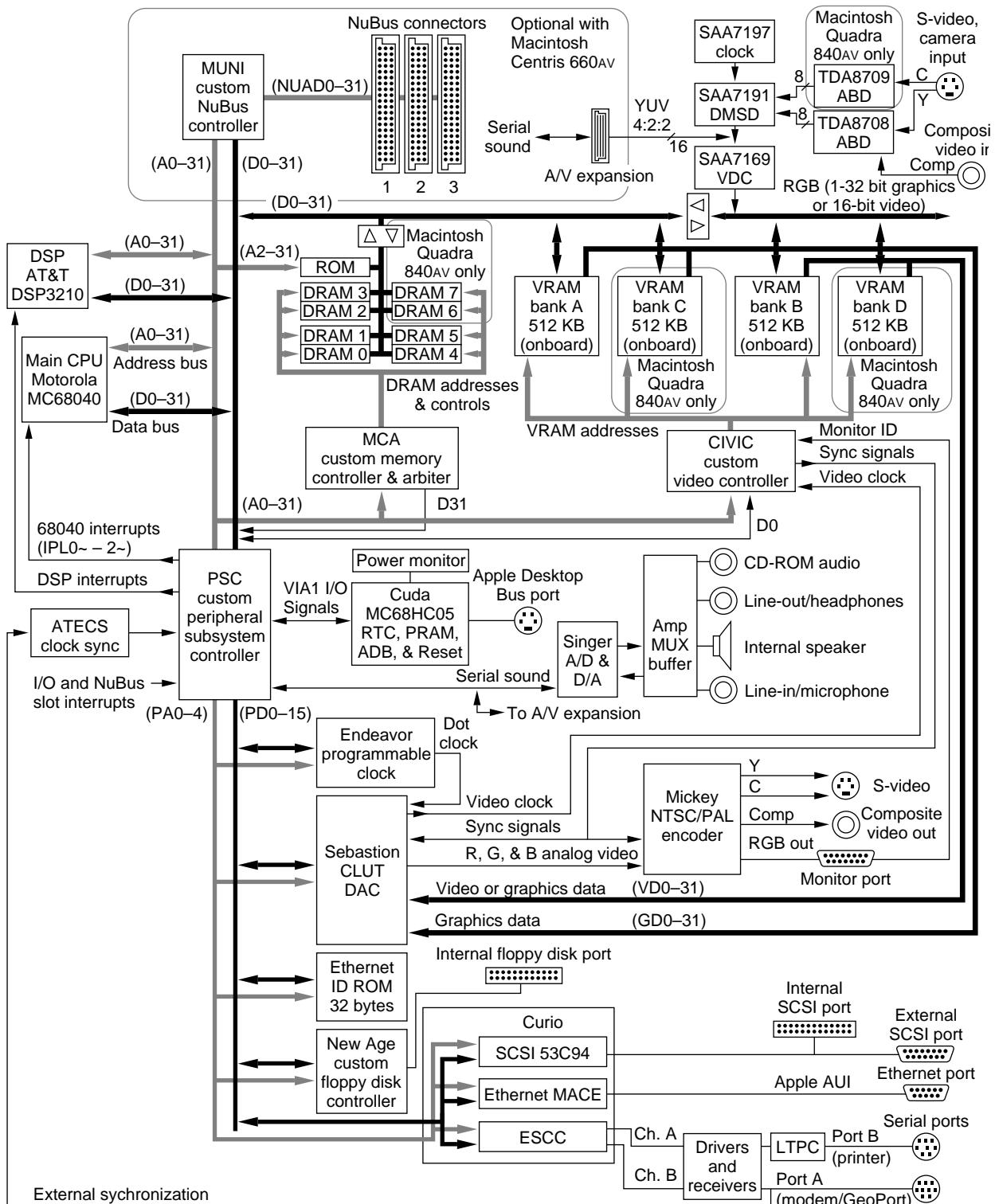
All the Macintosh Quadra 840AV and Macintosh Centris 660AV circuitry, except for the power supply, is contained on a single multilayer circuit board. The chips listed in "Functional Units," later in this chapter, can be identified by markings on the board.

System Architecture

The overall data flow relations among the hardware units of the Macintosh Quadra 840AV and Macintosh Centris 660AV computers are summarized by the block diagram in Figure 2-1. Parts omitted in the Macintosh Centris 660AV are enclosed in dotted lines.

The units shown in Figure 2-1 are described in more detail in the next section.

Figure 2-1 Functional diagram

Functional Units

Features of the major functional hardware units shown in Figure 2-1 are summarized in the next sections.

Main Processor

The main processor of the Macintosh Quadra 840AV and Macintosh Centris 660AV is a Motorola MC68040 microprocessor. This unit has

■ an integer processing unit whose instruction set in user mode is compatible with the MC68030 processor

■ a floating-point processing unit compatible with the MC68881 and MC68882 processors

■ independent demand-paged memory management units for instructions and data

■ independent 4 KB instruction and data caches

■ multiple execution pipelines and full internal Harvard processing architecture

Neither model uses the optional bus-snooping feature of the MC68040 design.

Read-Only Memory

Both models contain 2 MB of ROM. For details of ROM access timing, see "ROM and RAM Management," later in this chapter.

Random-Access Memory

RAM consists of **dynamic random-access memory (DRAM)** on 72-pin **Single Inline Memory Modules (SIMMs).** Each SIMM contains one or two banks of DRAM with up to 16 MB of capacity per bank. In the Macintosh Centris 660AV, 4 MB of RAM capacity is soldered to the board and there are slots for two expansion SIMMs. In the Macintosh Quadra 840AV, all RAM is on SIMMs, with space for four expansion SIMMs. Thus maximum possible RAM capacity (using 16 MB RAM chips) is 128 MB for the Macintosh Quadra 840AV and 68 MB for the Macintosh Centris 660AV.

The DRAM in the Macintosh Quadra 840AV must have an access time of not more than 60 ns. The DRAM in the Macintosh Centris 660AV may have an access time up to 70 ns. DRAM compatible with either model must accept the column address strobe (CAS) before the row address strobe (RAS) for refresh and may not have more than eight chips per bank. Both models ignore the parity connection on DRAM SIMMs.

RAM addressing and RAM access timing are discussed in "ROM and RAM Management," later in this chapter. Certain changes to virtual memory management are described in Chapter 13, "Virtual Memory Manager."

Memory Controller and Arbiter

The **Memory Controller and Arbiter (MCA)** is a **complementary metal-oxide semiconductor (CMOS)** chip in a 160-pin package. It

■ supports the bus interface between RAM and ROM memory and the main processor, DSP, MUNI, and PSC

■ controls access to eight banks of DRAM with up to 16 MB capacity per bank

■ controls access to ROM

■ performs arbitration for control of the CPU bus between the main processor, PSC, MUNI, and DSP

■ furnishes bus timeout signals for the main processor, I/O, MUNI, and DSP buses

For details of MCA operation, see "ROM and RAM Management," later in this chapter.

Digital Signal Processor

The Macintosh Quadra 840AV and Macintosh Centris 660AV computers include an AT&T DSP3210 32-bit floating-point **digital signal processor (DSP).** The DSP gives these computers the ability to perform fast, complex real-time data processing tasks, such as speech recognition, audio compression, analog modem signal processing, and so on. In the Macintosh Quadra 840AV, the DSP runs at 66.6667 MHz; in the Macintosh Centris 660AV, it runs at 55.5000 MHz. The DSP chip contains its own 8 KB of internal RAM, which minimizes its use of system memory. A summary of DSP operation and programming is given in Chapter 3, "Introduction to Real-Time Data Processing," with further details in Chapters 4 and 5.

Peripheral Subsystem Controller

The **Peripheral Subsystem Controller (PSC)** is a CMOS chip in a 208-pin package. The PSC

■ provides nine dedicated DMA channels

■ decodes the I/O memory mapping

■ handles all internal system interrupts

■ handles system interrupts from the Versatile Interface Adapter (VIA) inputs

■ contains 16-byte buffers for sound data to and from the Singer sound encoder and decoder **(codec)**

For details of PSC operation, see "PSC Functions," later in this chapter.

Macintosh Universal NuBus Interface

The **Macintosh Universal NuBus Interface (MUNI)** is a CMOS chip in a 208-pin package. For a general discussion of NuBus, including its software details, see *Designing Cards and Drivers for the Macintosh Family,* third edition. The MUNI version of NuBus

■ supports the full range of NuBus master/slave transactions with single or block moves, including dumps and runs in which the main processor is master and NuBus is slave

■ supports faster data transfer rates to and from the CPU bus

■ supports NuBus '90 data transfers between cards at a clock rate of 20 MHz

■ provides first-in, first-out (FIFO) buffering of data between the CPU bus and accessory cards

In the Macintosh Quadra 840AV, the MUNI is mounted on the main circuit board; in the Macintosh Centris 660AV, it is on an optional NuBus adapter card. Details of the MUNI operation are given in "NuBus Interface," later in this chapter.

Cyclone Integrated Video Interfaces Controller

The **Cyclone Integrated Video Interfaces Controller (CIVIC),** used in both computers, is a CMOS chip in a 144-pin package. The CIVIC

■ manages either 1 MB or 2 MB of video RAM (VRAM)

■ controls data transfers between VRAM and the Video Data Path Chip and between VRAM and the Sebastian video color palette chip (described next)

■ provides 32-bit or 64-bit data paths between VRAM and the main processor or a slot card; supports data bursts from the main processor in all transfer modes

■ performs convolution of graphics data for line-interlaced displays

■ provides NTSC and PAL timing signals (see Table 2-2)

■ generates vertical blanking and video-in interrupt signals

For details of CIVIC operation, see "Video and Graphics I/O," later in this chapter.

Sebastian

The **Sebastian** is a video color palette and video digital-to-analog converter (DAC) in a 100-pin CMOS chip. The Sebastian

■ accepts up to 64 bits of digital input, either as one 64-bit port or as one or two 32-bit ports

■ lets one 32-bit port handle digital video while the other processes graphics (including QuickTime), using the same or different color lookup tables

■ supports mixing video with still graphics, even with different color depths

- supports both Truecolor and pseudocolor with alpha color lookup

- supports a transparency effect when blending video with still graphics under the control of alpha bits

- uses a convolution filter to minimize flicker in line-interlaced displays

- supports displays with dot clocks up to 100 MHz

For details of Sebastian operation, see "Video and Graphics I/O," later in this chapter.

Video Data Path Chip

The **Video Data Path Chip (VDC)** is a Phillips CMOS chip in a 100-pin package. The VDC

- performs input video window scaling with horizontal and vertical filtering

- accepts YUV 4:2:2 color-encoded input from the Digital Multistandard Decoder or the digital audio/video (DAV) expansion slot card bus

- produces 16-bit 1:5:5:5 RGB, 8-bit grayscale, or YUV 4:2:2 output

For details of VDC operation, see "Video and Graphics I/O," later in this chapter, and the Phillips 7169 data sheet.

Mickey

The **Mickey** is a composite video encoder in a 28-pin advanced bipolar CMOS chip. The Mickey

- accepts analog video signals from the Sebastian video color palette chip

- encodes to NTSC or PAL digital format

- produces S-video, composite, and RGB video outputs

For details of Mickey operation, see "Video and Graphics I/O," later in this chapter.

New Age

The **New Age** is a floppy disk controller in a 64-pin CMOS chip. The New Age

- controls an Apple SuperDrive for all its recording densities

- uses a command set compatible with MFM and Apple GCR formats

- generates an interrupt on disk insertion

- performs 16-byte first-in, first-out data buffering

- supports full asynchronous operation for DMA

The New Age supports the standard Macintosh floppy disk software interface, as described in *Inside Macintosh.*

Curio

The **Curio** is a multipurpose I/O chip that contains a Media Access Controller for Ethernet (MACE), a SCSI controller, and a Serial Communications Controller (SCC). New serial driver code in the Macintosh Quadra 840AV and Macintosh Centris 660AV system software is discussed in Chapter 10, "DMA Serial Driver."

The SCC section of the Curio includes 8-byte FIFO buffers for both transmit and receive data streams.

Curio functions are discussed in "External Device Interfaces" and "PSC Functions," later in this chapter.

Apple Telecom External Clock Synchronizer

The **Apple Telecom External Clock Synchronizer (ATECS)** is a control chip that can synchronize the DSP and sound subsystems to an external clock signal received through the Apple GeoPort serial port connector. In the absence of an external clock, it generates crystal-controlled 49.152 MHz timing signals for 48 KHz operation or 45.1584 MHz timing signals for 44.1 KHz operation. For a description of Apple GeoPort, see "Serial Ports," later in this chapter.

Cuda

The **Cuda** is a microcontroller chip. It

■ turns system power on and off

■ manages system resets from various commands

■ maintains parameter RAM

■ manages the Apple Desktop Bus (ADB)

■ manages the real-time clock

■ lets an external signal through the Apple GeoPort serial port control system power

Cuda functions are discussed in more detail in "External Device Interfaces," later in this chapter. For a description of Apple GeoPort, see "Serial Ports," later in this chapter.

Singer

The **Singer** is an I/O chip that constitutes a 16-bit digital sound codec. It conforms to the IT&T *ASCO 2300 Audio-Stereo Codec Specification*. For details of its operation, see "Sound I/O," later in this chapter.

Endeavor

The **Endeavor** is a programmable video clock chip used in the Macintosh Quadra 840AV; the equivalent in the Macintosh Centris 660AV is called **Clifton Plus.** For details of the operation of these chips, see "Video and Graphics I/O," later in this chapter.

Digital Multistandard Decoder

The **Digital Multistandard Decoder (DMSD)** is a Phillips chip that decodes the color information in NTSC, PAL, and SECAM video formats using a clock synchronized to their line frequency. For details of DMSD operation, see "Video and Graphics I/O," later in this chapter, and the Phillips 7191B data sheet.

System Clocks

Operation of the Macintosh Quadra 840AV and Macintosh Centris 660AV is driven by several different clocks running at different frequencies. These clocks are listed in Table 2-2, where their frequencies are given in MHz.

Table 2-2   Clock frequencies

Location	Signal name	Frequency (MHz)	Source	Usage
CPU bus	PClk	80.0000/50.0000[*]	Divider	Main processor, MCA, CIVIC[†]
	BClk	40.0000/25.0000[*]	Divider	Main processor, MCA, CIVIC, PSC, MUNI
	PClk/4	20.0000/12.5000[*]	Oscillator	Divider
I/O bus	C22_5792M	22.5792	PSC	Singer (44.1 KHz)
	C24_576M	24.5760	PSC	Singer (48 KHz)
	C16M	15.6672	PSC	New Age, Curio
	C32M	31.3344	Oscillator	PSC
	C25M	25.0000	Oscillator	SCSI
	C15_0528M	15.0528	Crystal	ATECS (44.1 KHz)
	C16_384M	16.3840	Crystal	ATECS (48 KHz)
	C45_1584M	45.1584	ATECS	PSC (44.1 KHz)
	C49_1520M	49.1520	ATECS	PSC (48 KHz)
	C20M	20.0000	Crystal	MACE Ethernet
	CudaClk	0.032768	Crystal	Cuda

Table 2-2   Clock frequencies (continued)

Location	Signal name	Frequency (MHz)	Source	Usage
DSP	CKI	66.6667 / 55.5000[*]	Oscillator	DSP, MCA
NuBus	C40M	40.0000	Oscillator	MUNI
	C20M	20.0000	MUNI	Accessory card slots
	CN10M	10.0000	MUNI	Accessory card slots
Video	Video in	26.8000	Crystal	DMSD
Video	Dot clock	Various[‡]	Endeavor	Sebastian
	NTSC	14.31818	Oscillator	Sebastian, Mickey
	PAL	17.734475	Oscillator	Sebastian, Mickey

[*] The two values are the frequencies in the Macintosh Quadra 840AV and Macintosh Centris 660AV, respectively.
[†] In the Centris 660AV only.
[‡] Up to 100 MHz maximum, depending on the monitor; see Table 2-14.

Signal Buses

The Macintosh Quadra 840AV and Macintosh Centris 660AV contain several internal signal buses. The principal ones are shown in Figure 2-1. They are

■ the CPU bus, containing 32 address lines and 32 data lines

■ NuBus, containing 32 combined data and address lines

■ the digital audio/video (DAV) expansion connector bus, containing 16 video data lines and five audio data lines

■ the video bus, containing two sets of 32 lines each (or a combined set of 64 lines) for video or graphics

■ the I/O bus, containing five address lines and 16 data lines

For detailed information about NuBus, see *Designing Cards and Drivers for the Macintosh Family,* third edition. In the Macintosh Centris 660AV, NuBus exists only if an adapter card has been installed in the PDS connector. The PDS connector is described in "Processor-Direct Cards for the Macintosh Centris 660AV," later in this chapter.

Bus Arbitration

Four chips are able to take control of the CPU bus: the main processor, the MUNI, the PSC, and the DSP (which is asynchronous with respect to the main processor). The MCA performs arbitration between these chips for control of the CPU bus.

Priority among accessory cards for control of NuBus depends on which slots they occupy. Lower-numbered slots have higher priority.

Control of the I/O bus is managed by the PSC chip, as described in "Bus Arbitration Performed by the PSC," later in this chapter.

Bus Timeouts

The MCA chip performs two timeout functions for the CPU bus:

■ If the CPU, PSC, or MUNI does not respond to a cycle start signal within a critical time, the MCA terminates that chip's bus control and issues a bus error signal. The critical time is 32 µs for NuBus address space ($6000 0000 to $FFFF FFFF) and 16 µs for the rest of the address space ($0000 0000 to $5FFF FFFF).

■ If the DSP does not issue a cycle start signal within 16 DSP clock cycles after it is granted bus control, the MCA terminates the DSP's control and issues a bus error signal.

The MUNI chip generates a bus error if any Nubus transaction takes longer than 25.6 µs.

ROM and RAM Management

During the system startup process, the system software programs the MCA for the configuration of ROM and RAM actually present in the user's hardware.

The CPU bus can read from and write to RAM, as well as read from ROM, in either single-address accesses or four-address bursts.

DRAM Configurations

Possible DRAM configurations are shown in Table 2-1.

Table 2-1   DRAM configurations

Bank size (MB)	Organization	Row address bits	Column address bits
1	256 Kbit x 4	9	9
1	1 Mbit x 4	10	10
2	512 Kbit x 8	10	9
4	4 Mbit x 4	11	11
8	2 Mbit x 8	11	10
4	4 Mbit x 4	12	10
8	2 Mbit x 8	12	9
16	1 Mb x 16	11	9

Startup Memory Addressing

At the beginning of the system startup process, the MCA maps the ROM code to the lower end of the RAM address space, starting at address $0000 0000. When the main processor is reset, it sets the program counter to the 32-bit address found at $0000 0004, which transfers execution to the actual system software entry point in the space $4000 0000 to $4FFF FFFF. After this first access, ROM is no longer mapped to the bottom of the RAM space.

Access Timing

The Macintosh Quadra 840AV and Macintosh Centris 660AV use 60-ns DRAM chips. Access to RAM in both models may occur under either of two conditions:

■ Multiple access occurs when the CPU bus requests a new access immediately after the end of the previous access.

■ Single access occurs when there is an interval of at least one main processor clock cycle after the previous access.

The number of clock cycles required for various chips to access RAM under these two conditions in various read and write modes is shown in Table 2-3. The clock rate for determining cycle times is BClk for the main processor, the MUNI, and the PSC; it is CKI for the DSP. These clock rates are listed in Table 2-2.

Table 2-3   DRAM access times

		Main processor		MUNI and PSC		
Access type	**Condition**	Macintosh Quadra 840AV	Macintosh Centris 660AV	Macintosh Quadra 840AV	Macintosh Centris 660AV	**DSP**
Single read	Single	6	4	5	4	9
	Multiple	7	5	6	5	11
Single write	Single	6	3	4	3	8
	Multiple	7	5	6	5	11
Burst read	Single	6-3-3-3	4-2-2-2	5-3-3-3	4-2-2-2	9-4-4-4
	Multiple	7-3-3-3	4-2-2-2	6-3-3-3	4-2-2-2	11-4-4-4
Burst write	Single	6-2-2-2	3-2-2-2	4-3-3-3	3-2-2-2	8-7-7-7
	Multiple	7-2-2-2	4-2-2-2	6-3-3-3	4-2-2-2	11-7-7-7

The DRAM refresh cycle takes eight main processor clock cycles, but does not affect RAM access timing. DRAM refresh does not occur when the DSP controls the CPU bus.

Both computers use 120-ns ROM chips. CPU bus accesses to ROM take five bus clock cycles in the Macintosh Centris 660AV and seven bus clock cycles in the Macintosh Quadra 840AV.

For additional information about burst write actions on the processor bus, see "Processor Bus Burst Write Timing," later in this chapter.

External Device Interfaces

This section discusses the interfaces between the Macintosh Quadra 840AV and Macintosh Centris 660AV computers and external devices through
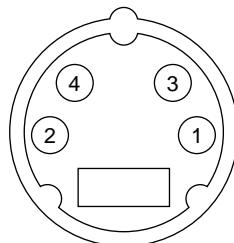
- the ADB, which supports keyboards, mouse devices, trackballs, and so on
- an Ethernet port for wide area network access
- two serial ports for printers, modems, AppleTalk, and other serial I/O
- a SCSI connection for devices such as hard disk drives
- an interface to the Apple SuperDrive floppy disk drive

For video interface information, see "Video and Graphics I/O," later in this chapter. For sound interface information, see "Sound I/O," later in this chapter.

Apple Desktop Bus

The **Apple Desktop Bus (ADB)** is an asynchronous serial communication bus used to connect relatively slow user-input devices to Macintosh computers. Its software characteristics are described in *Inside Macintosh.* One ADB connector is located on the back panel. It is a mini-DIN 4-pin socket, as shown in Figure 2-2.

Figure 2-2   ADB socket

The ADB pin assignments are shown in Table 2-4.

Table 2-4   ADB pin assignments

Pin	Description
1	Data; grounded by an open collector or pulled to +5 V through 470 Ω
2	Power on, fed by +5 V through 100 kΩ; connect to pin 4 to turn on the system
3	+5 V at 500 mA maximum drain; protected by a 1.25-A circuit breaker
4	Ground return

Ethernet Port

Both the Macintosh Quadra 840AV and Macintosh Centris 660AV contain built-in support for Ethernet. The user can plug a drop-box cable available from Apple or from third-party vendors into a standard Ethernet connector on the back panel.

The Ethernet port pin assignments are shown in Table 2-5.

Table 2-5   Ethernet port pin assignments
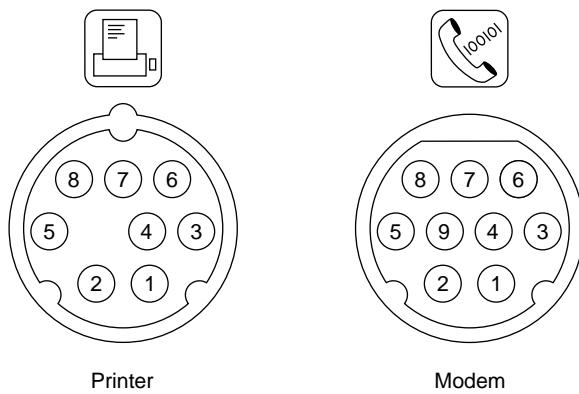
Pin	Description	Pin	Description
1	+5 V	8	+5 V
2	DI+	9	DO+
3	DI–	10	DO–
4	Ground	11	Ground
5	CI+	12	NC
6	CI–	13	NC
7	+5 V	14	+5 V

Serial Ports

The back panel on both the Macintosh Quadra 840AV and Macintosh Centris 660AV contains two serial data I/O ports. An 8-pin mini-DIN socket is marked as a printer port, and a 9-pin mini-DIN socket is marked as a modem port. Both sockets accept 8-pin plugs, but only the modem port accepts 9-pin plugs.

The physical patterns for the serial port sockets are shown in Figure 2-3.

Figure 2-3   Serial port connectors

Printer   Modem

Either port can be independently programmed for asynchronous or synchronous communication formats up to 9600 baud, including AppleTalk. The printer port supports LocalTalk hardware protocols by means of a **LocalTalk Patch Chip (LTPC).** In addition, the port marked "Modem" supports the full range of Apple GeoPort protocols, helping the computer communicate with a variety of ISDN and other telephone transmission means by using external pods.

For information about serial driver software in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Chapter 10, "DMA Serial Driver."

Table 2-6 gives the pin assignments for the two serial ports. The Both ports column shows how both connectors support AppleTalk and common serial communications; the GeoPort columns show how the 9-pin connector can also support Apple GeoPort protocols.

Table 2-6   Serial port pin assignments

Pin	Both ports	GeoPort	Special GeoPort functions
1	HSK_o	$SCLK_{out}$	Reset pod or get pod attention
2	HSK_i	$Sync_{in}/SCLK_{in}$	Serial clock from pod (up to 920 Kbits/sec)
3	TxD–	TxD–	
4	Gnd/shield	Gnd/shield	
5	RxD–	RxD–	
6	TxD+	TxD+	
7	GP_i	Wakeup/TxHS	Wake up CPU or do DMA handshake
8	RxD+	RxD+	
9		+5 V	Power to pod

SCSI Connection

The SCSI interface in the Macintosh Quadra 840AV and Macintosh Centris 660AV exists in two forms: an internal 50-pin ribbon connector for internal devices and an external DB-25 connector for external devices. Internal device mounting is shown in Appendix D, "Mechanical Details."

The Macintosh Centris 660AV can support one 3.5-inch internal hard disk and an optional 5.25-inch CD-ROM drive. The Macintosh Quadra 840AV can support an additional internal disk drive. Both models accept up to a total of seven SCSI devices, internally and externally.

The SCSI interface in the Macintosh Quadra 840AV and Macintosh Centris 660AV is electrically and mechanically compatible with SCSI interfaces on other computers of the Macintosh family. For information about SCSI Manager software in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Chapter 9, "SCSI Manager 4.3."

Power Budgets

The maximum continuous power budgets available for each SCSI device attached to a Macintosh Quadra 840AV or Macintosh Centris 660AV computer are shown in Table 2-7.
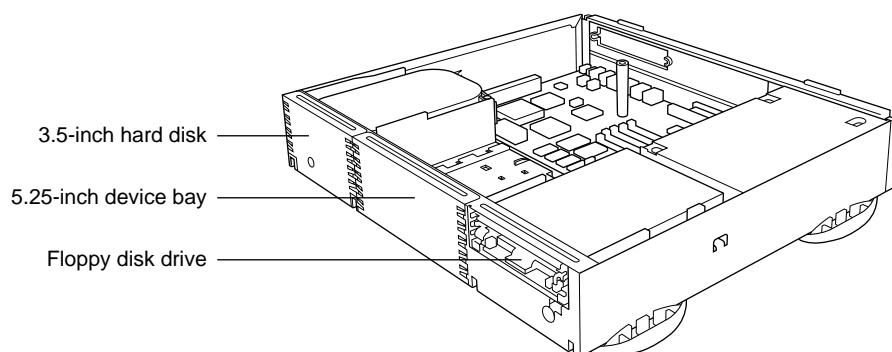
Table 2-7   SCSI power budgets

Computer	+5 V	+12 V
Macintosh Quadra 840AV	1.5 A	4.3 A
Macintosh Centris 660AV	1.5 A	1.5 A

Internal SCSI Locations

The locations of the Macintosh Centris 660AV computer's internal SCSI devices are shown in Figure 2-4.
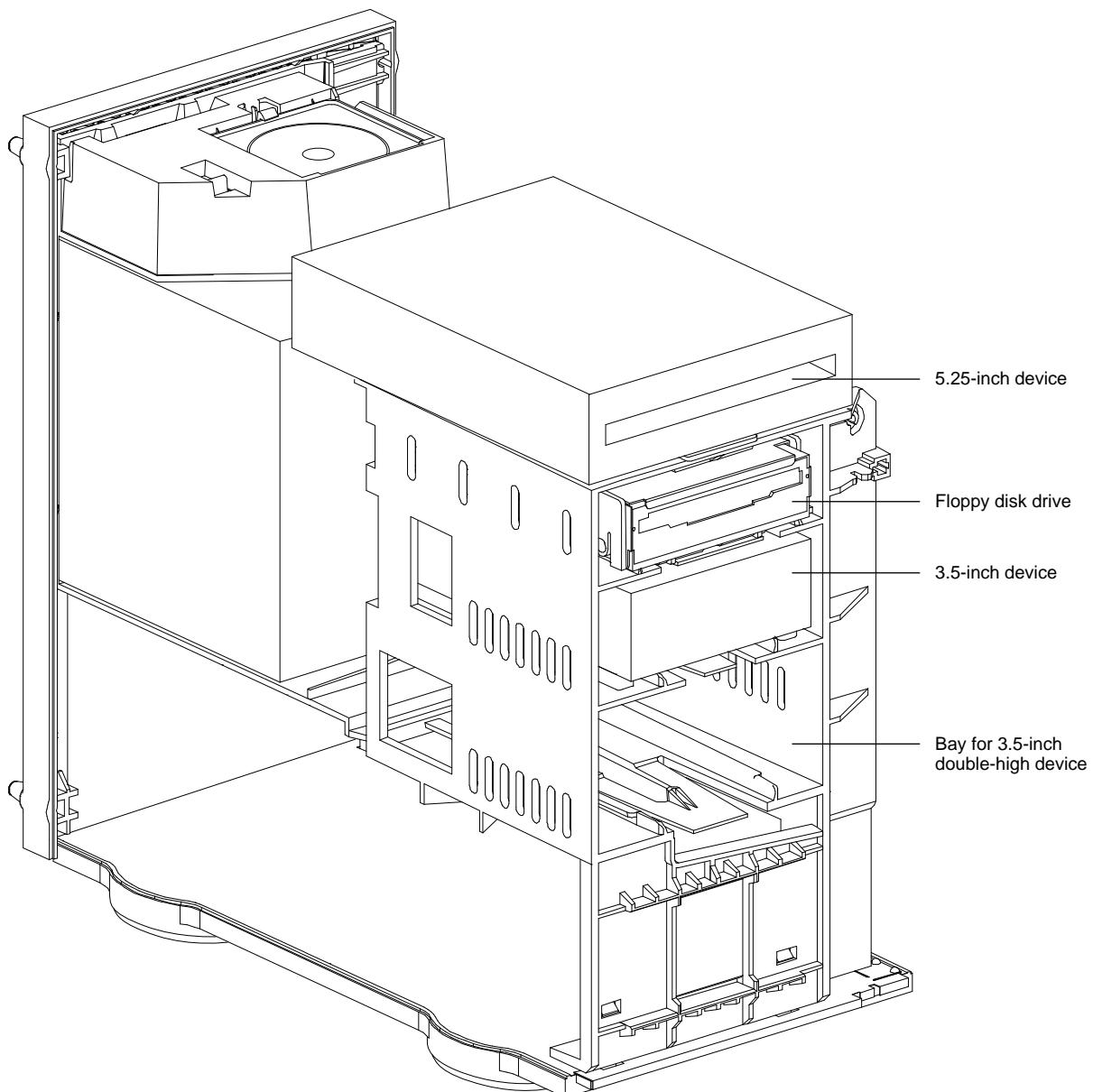
Figure 2-4   Macintosh Centris 660AV internal SCSI device space

3.5-inch hard disk

5.25-inch device bay

Floppy disk drive

The locations of the Macintosh Quadra 840AV computer's internal SCSI devices are shown in Figure 2-5.

Figure 2-5   Macintosh Quadra 840AV internal SCSI device space

Pin Assignments

The standard internal and external SCSI pin assignments are shown in Table 2-8.

Table 2-8   SCSI pin assignments

Pin number		Description	Pin number		Description
Internal	External		Internal	External	
1	7	Ground	2	8	/DATA0[*]
3	9	Ground	4	21	/DATA1
5	14	Ground	6	22	/DATA2
7	16	Ground	8	10	/DATA3
9	18	Ground	10	23	/DATA4
11	24	Ground	12	11	/DATA5
13		Ground	14	12	/DATA6
15		Ground	16	13	/DATA7
17		Ground	18	20	/DATAP
19		Ground	20		Ground
21		Ground	22		Ground
23		NC	24		NC
25		NC	26	25	TERMPWR
27		NC	28		NC
29		Ground	30		Ground
31		Ground	32	17	/ATN
33		Ground	34		Ground
35		Ground	36	6	/BUSY
37		Ground	38	5	/ACK
39		Ground	40	4	/RST
41		Ground	42	2	/MSG
43		Ground	44	19	/SEL
45		Ground	46	15	/C/D
47		Ground	48	1	/REQ
49		Ground	50	3	/I/O

[*]   A slash before a signal name indicates that it is in the low state when active.
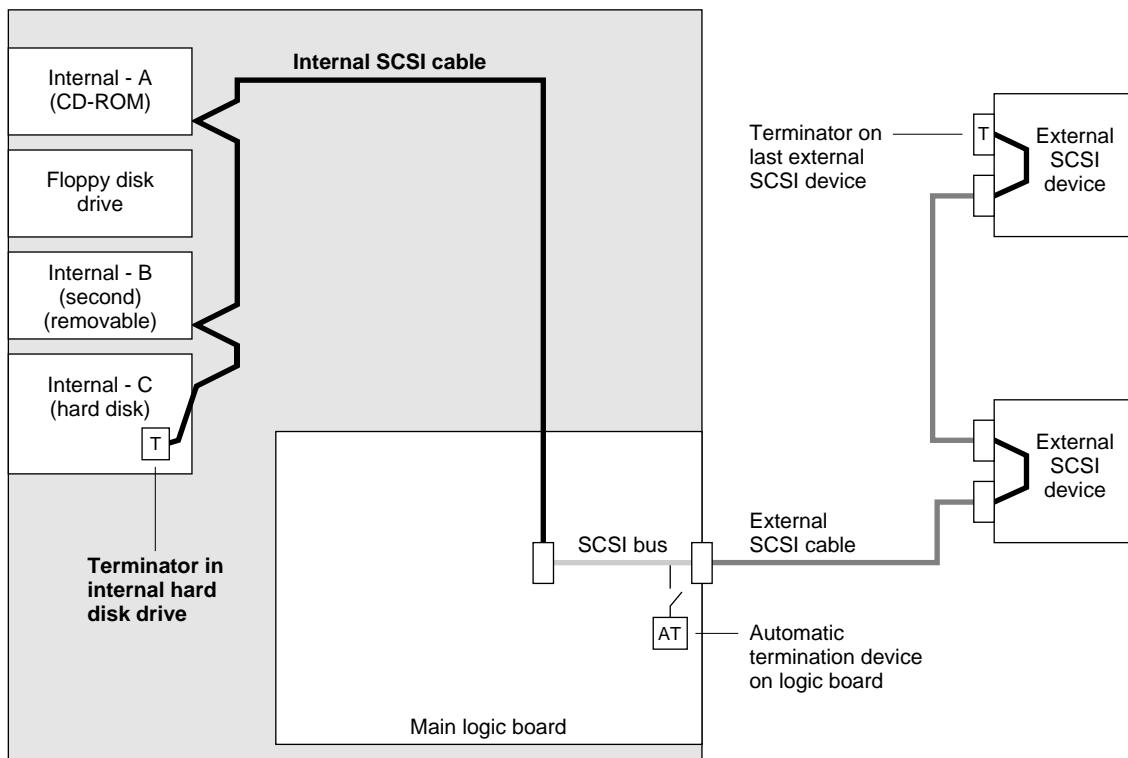
Automatic SCSI Termination

Because the internal portion of the SCSI bus must be long enough to connect multiple devices, the bus requires termination at both ends. As on other Macintosh models, the external end of the bus is normally terminated at the last external device. On the Macintosh Quadra 840AV and Macintosh Centris 660AV, the internal end of the bus—the end at the last internal hard disk drive—is terminated in the drive itself.

Figure 2-6 shows the arrangement of the SCSI cables in a typical Macintosh Quadra 840AV configuration. The bus is continuous across the internal SCSI cable, the SCSI bus traces on the logic board, and the external SCSI cable (if any). The boxes with the letter T represent terminators. The Macintosh Centris 660AV has a similar arrangement, but contains a maximum of two internal SCSI devices.

Figure 2-6   SCSI bus terminators in a typical Macintosh Quadra 840AV configuration

Both computers include a new feature that automatically provides the proper termination when no external device is connected, that is, when the SCSI bus ends at the external connector. When no external device is connected, special circuitry terminates the bus on the logic board near the external connector. When one or more external SCSI devices are connected, the circuitry detects the external termination during system reset and disconnects the termination on the logic board. In Figure 2-6, the box marked AT on the logic board indicates the automatic termination device.

Signals on the SCSI bus are usually connected to open-collector devices that can pull the line low but that depend on external power to pull it high. The bus includes a line called TERMPWR that provides pull-up power. The standard Macintosh terminator block used at the last external SCSI device terminates each line with a 220 Ω pull-up resistor and a 330 Ω resistor to ground.

Installing Internal SCSI Devices

In the Macintosh Quadra 840AV and Macintosh Centris 660AV, the device at the end of the internal SCSI cable includes terminators; all other internal devices do not. When installing internal SCSI devices, the installer must make sure that the device at the end of the cable has terminators and must remove terminators from any other internal SCSI devices.

SCSI termination can be present on only the last internal device—the one at the end of the internal SCSI cable. If none of the internal devices is terminated, the computer will malfunction; if more than one internal device includes terminators, the computer will malfunction and the logic board may be damaged.

Physical provisions for internal SCSI device mounting are given in Appendix D, "Mechanical Details."

As in all SCSI installations, all devices on the SCSI bus must have different ID numbers.

Floppy Disk Drive Connection

Both models contain one internal Apple SuperDrive floppy disk drive. Table 2-9 gives the pin assignments for the 20-pin floppy disk drive connector.

Table 2-9   Floppy disk drive connector pin assignments

Pin	Signal	Description	Pin	Signal	Description
1	GND	Ground	11	+5V	+5 V
2	PH0	Phase 0 state control	12	SEL	Head select
3	GND	Ground	13	+12V	+12 V
4	PH1	Phase 1 state control	14	/ENBL[*]	Drive enable
5	GND	Ground	15	+12V	+12 V
6	PH2	Phase 2 state control	16	RD	Read data
7	GND	Ground	17	+12V	+12 V
8	PH3	Phase 3 register write strobe	18	WR	Write data
9	NC	NC	19	+12V	+12 V
10	/WRREQ	Write data request	20	NC	NC

[*] A slash before a signal name indicates that it is in the low state when active.

PSC Functions

The PSC contains nine programmable DMA channels that transfer data between random-access memory and various I/O interfaces. The PSC also performs address decoding for many I/O memory allocations.

DMA Channels Controlled by the PSC

The PSC provides nine DMA channels that can access RAM or ROM but cannot access the I/O or NuBus address spaces. The characteristics and uses of these DMA channels are shown in Table 2-10. The "Width" column lists the number of bits of data that the PSC transfers over the I/O bus during each transaction. The "Buffer" column lists the capacity of the PSC's internal FIFO buffer for each DMA channel.

Table 2-10   PSC DMA channels

Name	Width (bits)	Buffer (bytes)	I/O function served
SCSI	16	16	SCSI port
ENetRd	16	16	Ethernet read
ENetWr	16	16	Ethernet write
FDC	8	4	Floppy disk controller
SCCA	8	4	SCC Channel A (SCC Rx or Tx, GeoPort)
SCCB	8	4	SCC Channel B (SCC Rx or Tx)
SCCATx	8	4	SCC Channel A (Scc Tx, GeoPort)
SndIn	1	16	Singer sound input
SndOut	1	16	Singer sound output

Each DMA channel is controlled by two sets of programming registers. Using two register sets helps software optimize data transfers to and from physical memory and increases the limit of system interrupt latency when the data input is continuous (such as from Ethernet). In a virtual memory environment, software must guarantee that memory pages are contiguous when DMA transfers controlled by the PSC cross a page boundary.

Bus Arbitration Performed by the PSC

The PSC controls access to the I/O bus by the main processor and by the DMA channels described in the previous section. At the first level of arbitration, the PSC grants its DMA channels two accesses for every one access granted to the main processor. When DMA

channels contend with one another, the PSC grants them access to the I/O bus in the order of priority shown in Table 2-11. This table also shows the order of priority in which the PSC controls access by its DMA channels to the CPU bus.

Table 2-11   Priority of DMA channel access

Priority	To the I/O bus	To the CPU bus
Highest	FDC	SndOut
	SCCA	SndIn
	SCCA Tx	FDC
	SCCB	SCCA
	ENetRd	SCCA Tx
	ENetWr	SCCB
	SCSI	ENetRd
		ENetWr
Lowest		SCSI

Video and Graphics I/O

The Macintosh Quadra 840AV and Macintosh Centris 660AV contain a sophisticated video and graphics I/O system that handles video input and output signals and supports a wide variety of Apple and third-party monitors.

The video I/O system also lets the user connect a television set as a monitor, using either NTSC or PAL format. For further information, see "Video Television Output," in Chapter 11.
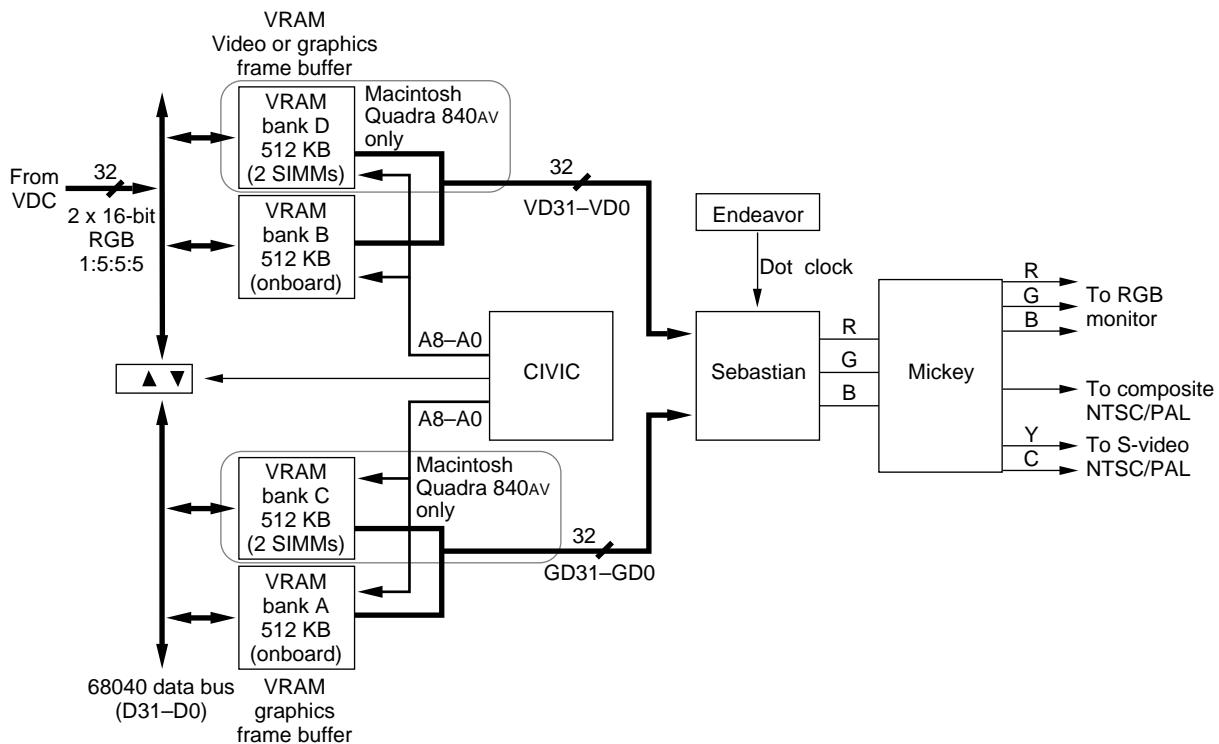
Some monitors go into power-saving mode when the sync signals are disabled. New routines in the video driver support this feature, as described in "New Control and Status Routines," in Chapter 11.

The output part of the video system is shown in Figure 2-7.

As shown in Figure 2-7, the video and graphics I/O system is built around two banks of VRAM. Each bank holds 512 KB and is expandable in the Macintosh Quadra 840AV to 1 MB. Thus, total VRAM capacity in the Macintosh Quadra 840AV may be either 1 MB or 2 MB; in the Macintosh Centris 660AV it is limited to 1 MB. The VRAM is controlled by the CIVIC chip. By programming the CIVIC, an application can configure it in either of the following two ways:

■ as a single frame buffer that uses all the VRAM capacity

■ as two frame buffers, one for video and one for graphics

Figure 2-7   Video and graphics output system block diagram

If the VRAM is configured as a single video frame buffer, it can all be used for graphics and the video input can be disabled. In this case, the CIVIC controls data access to VRAM from the following sources:

■ the main processor

■ the PSC, using I/O direct memory access

■ the MUNI chip

If the VRAM is configured as two frame buffers, it can store video as well as graphics. In Figure 2-7, the VRAM banks shown at the top of the figure can store video frames and the lower banks can store only graphics. In this configuration, the CIVIC can provide access to all VRAM from the sources just listed and it can also store video data from the VDC in the video VRAM. The video input subsystem that provides data to the VDC is described in the next section.

Video images and graphics images stored in VRAM may have different color depths. The two images exit VRAM through its serial access memory port and pass to the Sebastian color palette chip. Sebastian provides independent color lookup tables for video and graphics images and mixes them into a single digital RGB data stream. The Sebastian then converts the result into analog RGB video, using internal DAC circuits.

Analog RGB data passes to the Mickey encoder chip. Mickey either sends RGB directly to the monitor connector or encodes it into NTSC or PAL video signals in composite or
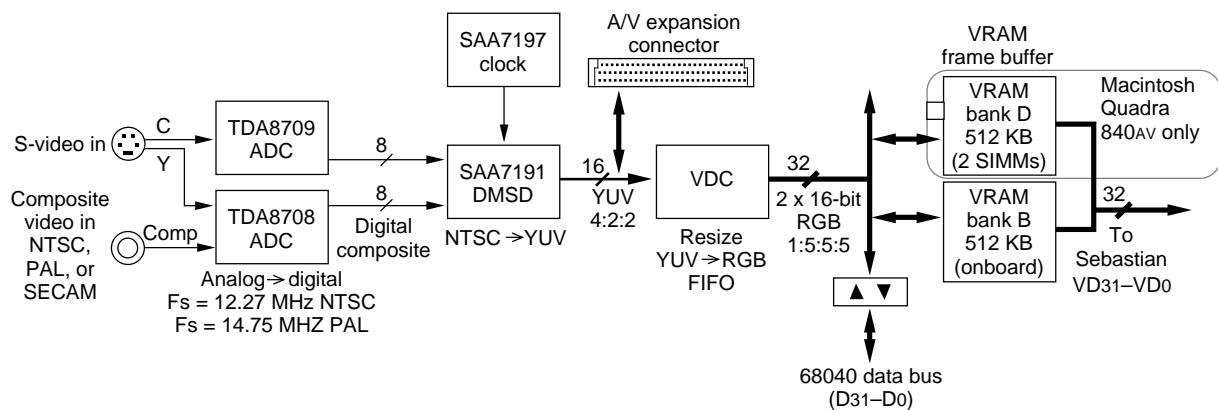
S-video format and sends it to other connectors located on the back panel. Monitors available for the Macintosh Quadra 840AV and Macintosh Centris 660AV are discussed in "Video Monitor Interface," later in this chapter.

External Video Input

Figure 2-8 shows details of the processing of video input from an external source such as a videocam or videocassette deck.

Figure 2-8   Video input subsystem
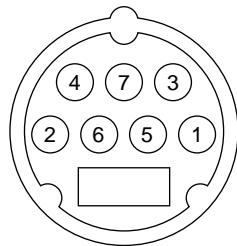
The input signal, which may be analog composite or S-video in NTSC, PAL, or SECAM format, enters through an external 7-pin mini-DIN socket. A cable adapter is provided to receive composite video from external devices that have RCA connectors. TDA8708 and TDA8709 video ADC chips digitize the composite video waveform, and the DMSD chip decodes the result into YUV format. This common digital video format, also known as YCrCb, is described in CCIR Recommended Standard 601-2.

Digital video in YUV format then passes to the digital audio/video (DAV) expansion connector, where it may be picked up by a NuBus expansion card, and to the VDC. A slot card that uses the DAV connector may disable the DMSD and feed its own YUV video to the VDC—for example, a slot card containing a video decompression engine. The DAV connector is described in "Digital Audio/Video Expansion Connector," later in this chapter.

The VDC scales down the video image and converts its format to either 8-bit grayscale, 15-bit RGB, or 16-bit YUV. It stores the result in the VRAM buffer under the control of the CIVIC chip. The video input connector is shown in Figure 2-9.

The pin assignments for the video input connector are shown in Table 2-12.

Figure 2-9   Video input connector

Table 2-12   Video input connector pin assignments

Pin	Description
1	AGND (signal)
2	AGND (power)
3	Video Y
4	Video C
5	I^2C clock (Phillips serial bus)
6	+12 V at 20 mA maximum drain
7	I^2C data (Phillips serial bus)

The data rate for full-screen NTSC video (640 by 480 pixels at 30 frames per second) is 18.43 MB per second. The data rate for full-screen PAL video (768 by 576 pixels at 25 frames per second) is 22.12 MB per second. This means that it is practical to record a video image up to one-quarter screen in size on an output device such as a hard disk drive in real time, without data compression.

Video RAM Usage

Each computer is delivered with two banks of VRAM soldered in, each bank providing 0.5 MB of storage. One of the two banks can supply a graphics screen image for monitors of small size or low color depths, letting the other bank supply live video to be mixed with the graphic image. Two banks together can support graphics alone on monitors that are larger or use more bits per pixel. The maximum video window size on the Macintosh Quadra 840AV is 640 by 480 pixels; the maximum video window size on the Macintosh Centris 660AV is 512 by 384 pixels.

The Macintosh Quadra 840AV also contains provision for VRAM expansion. When the user installs two more banks of VRAM in the SIMM expansion sockets, the resulting VRAM capacity can support mixed video and graphics in full 24-bit color on small and medium-sized monitors and in 16-bit or 8-bit color on larger monitors.

The color depths available when the Macintosh Quadra 840AV drives Apple monitors with and without VRAM expansion are listed in Table 2-13. Expanded VRAM is available only in the Macintosh Quadra 840AV; color depths and monitor configurations that are also supported by the Macintosh Centris 660AV (using standard VRAM) are printed in **boldface.**

Table 2-13   VRAM sizes and monitor color depths

Monitor type	Screen size Hor. x vert.	Standard VRAM (1 MB)		Expanded VRAM (2 MB)	
		Graphics	Graphics/video	Graphics	Graphics/video
12-inch RGB*	512 x 384	**32**	**8 / 16**	32	8 / 16
	560 x 384	**32**	**8 / 16**	32	8 / 16
13-inch RGB or 12-inch mono*	512 x 384	**32**	**8 / 16**	32	8 / 16
	640 x 400	**32**	**8 / 16**	32	8 / 16
	640 x 480	**16**	**8 / 16**	32	8 / 16
Full-page mono*	640 x 870	**8**	**4 / 16**	8	8 / 16
Full-page RGB	640 x 870	**8**	**4 / 8**	16	8 / 16*
16-inch RGB*	832 x 624	**16**	**8 / 16**	32	8 / 16
19-inch RGB	1024 x 768	**8**	**4 / 8**	16	8 / 8
Two-page mono	1152 x 870	8	4 / 8	8	8 / 8
Two-page RGB	1152 x 870	8	4 / 8	16	8 / 8
VGA*	640 x 480	**16**	**8 / 16**	32	8 / 16
Super VGA 56 Hz*	800 x 600	**16**	**8 / 16**	32	8 / 16
Super VGA 72 Hz*	800 x 600	**16**	**8 / 16**	32	8 / 16
Super VGA 60 Hz	1024 x 768	**8**	**4 / 8**	16	8 / 8
Super VGA 70 Hz	1024 x 768	**8**	**4 / 8**	16	8 / 8
NTSC	640 x 480	**16**	**8 / 16**[*]	32	8 / 16
	512 x 384	**32**	**8 / 16**[*]	32	8 / 16
Convolved NTSC	640 x 480	**8**	n.a.	8	n.a.
	512 x 384	**8**	n.a.	8	n.a.

continued

Table 2-13   VRAM sizes and monitor color depths (continued)

	Screen size	Standard VRAM (1 MB)		Expanded VRAM (2 MB)	
Monitor type	Hor. x vert.	Graphics	Graphics/video	Graphics	Graphics/video
PAL	768 x 576	**16**	**8 / 16***	32	8 / 16
	640 x 480	**16**	**8 / 16***	32	8 / 16
Convolved PAL	768 x 576	**8**	n.a.	8	n.a.
	640 x 480	**8**	n.a.	8	n.a.

* With a color depth of 16 bits in these configurations, the maximum video window size is
limited. If the video window width is 512 pixels or less, the height may be as large as 512
pixels; if the video window width is more than 512 pixels, the height is limited to 340 pixels.

The color depths in Table 2-13 are shown as the number of bits in which the color or
grayscale value of each pixel can be encoded. The refresh rates and pixel clocking rates at
which these monitors run are shown in Table 2-14, later in this chapter.
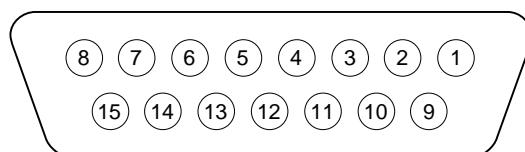
VRAM expansion requires 80-ns chips, using the same configuration of SIMM cards
as VRAM expansion in other Macintosh Quadra computers. Mechanical details, timing,
and pin assignments are given in "VRAM Expansion Cards," at the end of this chapter.

Video Monitor Interface

Either computer can be connected to a wide variety of external monitors by means of a
DB-15 socket located on their back panel. Some popular monitor types are listed in the
left column of Table 2-13. Signal timing diagrams for certain of these monitors are given
in "Video Output Timing," later in this chapter.

Apple Technical Note 326 contains full information about connecting various monitors
to the video monitor interface, including details of connector pin assignments and ID
codes assigned to Apple and some third-party monitors. It also describes hard-wire
connections that allow monitors to assert their ID codes and therefore automatically
configure the system when they are connected. Apple Technical Note 144 contains
additional information about color monitors.

Figure 2-10 shows the physical form of the DB-15 video monitor connector.
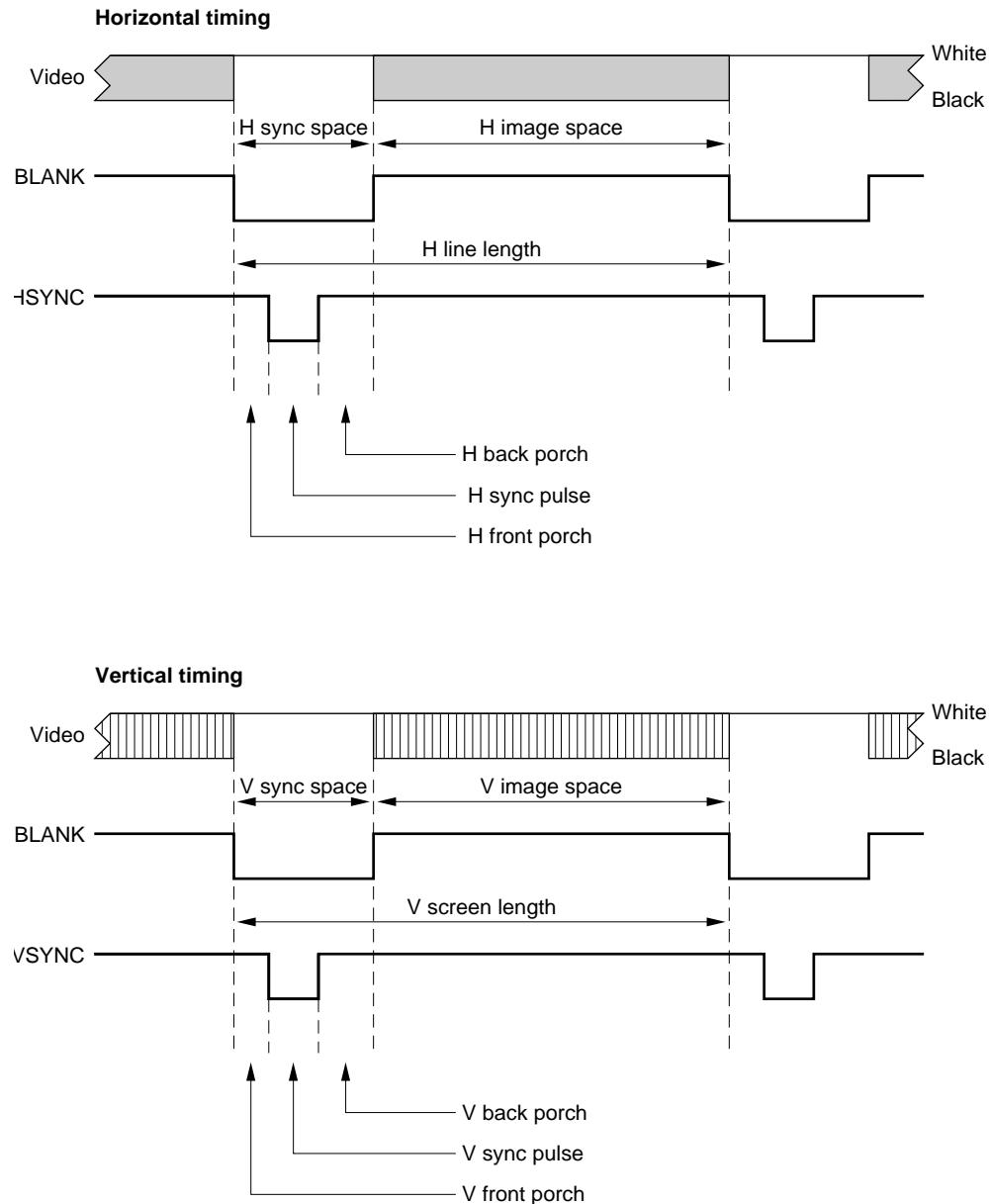
Figure 2-10   Video monitor connector

Video Output Timing

Figure 2-11 shows a general video timing diagram for Apple monitors.

Figure 2-11   Video timing diagram

Horizontal timing

Vertical timing

Table 2-14 gives monitor timing values. Some of these values are shown in Figure 2-11.

Table 2-14   Apple monitor timing values

Parameter	640 x 480	VGA	640 x 870	1024 x 768	1152 x 870
Dot clock, MHz	30.240	25.175	57.273	80.000	99.958
Dot interval, ns	33.069	39.722	17.460	12.500	10.004
Line rate, kHz	35.00	31.469	68.837	60.241	68.652
Line interval, μs	28.571	31.778	14.527	16.600	14.566
Frame rate, Hz	66.67	59.94	74.99	74.93	75.03
Frame interval, ms	15.000	16.683	13.336	13.346	13.328
H sync space, dots	224	160	192	304	304
H image space, dots	640	640	640	1024	1152
H line length, dots	864	800	832	1328	1456
H front porch, dots	64	16	32	32	32
H sync pulse, dots	64	96	80	96	128
H back porch, dots	96	48	80	176	144
V sync space, lines	45	45	48	36	45
V image space, lines	480	480	870	768	870
V screen length, lines	525	525	918	804	915
V front porch, lines	3	10	3	3	3
V sync pulse, lines	3	2	3	3	3
V back porch, lines	39	33	42	30	39

Miniature Videocam

Apple offers an inexpensive miniature videocam for the Macintosh Quadra 840AV and Macintosh Centris 660AV computers that uses a light-sensitive matrix on a chip with a built-in lens. The user can mount the mini-videocam above the monitor for videophone imaging or can move it about on the end of its cable to take pictures of objects or documents. With supplementary lenses, the videocam can make images of detail as fine as the wire bonds on an integrated circuit.

The mini-videocam has these general characteristics:

■ image: 360 by 288 pixels, monochrome with 256 gray levels

■ nominal view angle: 66°

■ sensitivity: adequate for use in a dimly lit office

■ depth of field (without supplementary lenses): 45 cm to infinity

■ output signal: 1 V, 75 Ω composite video with PAL or NTSC timing

■ software controls: exposure, gamma, image capture interval, video timing

■ unit identification in firmware: camera type and features

■ power drain: 50 mA at 12 V

Sound I/O

The system contains external ministereo sockets for sound I/O, connected through amplifiers to the Singer codec. The Singer uses only frame 0, leaving other frames available for other sound processing (for example, through the DAV connector). External plugs carry the left channel on the tip and the right channel on the ring, with the sleeve common. Sound I/O signals are described in Table 2-15.

Table 2-15   Sound I/O signals
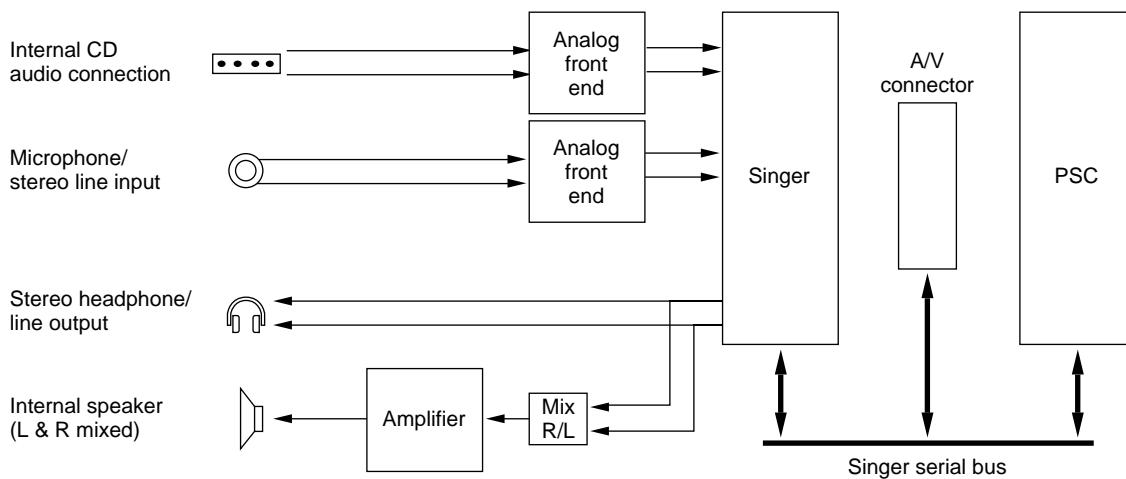
Panel label	Description
Audio In	8 kΩ impedance, 2 V rms maximum, 22.5 dB gain available
Audio Line Out	37 Ω impedance, 0.9 V rms maximum, attenuated –22.5 dB (crosstalk degrades from –80 dB to –32 dB when the audio output is connected to 32 Ω headphones)

Sound I/O bandwidth is 20 Hz to 20 kHz, plus or minus 2 dB. Total harmonic distortion and noise is less than 0.05% over the bandwidth with a 1 V rms sine wave input. The input signal-to-noise ratio (SNR) is 82 dB and the output SNR is 85 dB, with no audible discrete tones.

Both models are supplied with built-in speakers. Apple also offers a compatible high-quality microphone that is specifically designed for speech recognition applications. The components that support sound I/O are shown in Figure 2-12.

For details of speech generation and recognition in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Chapter 6, "Speech Manager," and Chapter 7, "Introduction to Speech Recognition."

Figure 2-12   Sound I/O components

NuBus Interface

The NuBus expansion card interface provides access between RAM or ROM and plug-in accessory cards. It is not designed to let plug-in cards access peripheral devices directly. The Macintosh Quadra 840AV accepts up to three cards; the Macintosh Centris 660AV accepts one. The expansion card implementation is based on the NuBus '90 specification (ANSI/IEEE Std 1196-1990) and has the following new features:

■ Each of the three Macintosh Quadra 840AV slots has a 4-bit geographic address. The addresses are $C, $D, and $E, corresponding to slots 4, 5, and 6 in the Macintosh II family of computers. The Macintosh Centris 660AV slot is address $C.

■ All data transfers on NuBus are synchronized by a 10 MHz clock. An additional 20 MHz clock supports burst transfers in cards that conform to the NuBus '90 specification. This permits faster data transfers than are possible with earlier NuBus designs.

■ NuBus supports a 32-bit addressing space (4 GB), accessible through justified 8-bit, 16-bit, and 32-bit data transfers.

■ MUNI generates a bus error if any transaction takes longer than 25.6 μs.

For full technical details about NuBus, including NuBus '90, see *Designing Cards and Drivers for the Macintosh Family,* third edition. For information about enabling NuBus block moves, see "NuBus Block Moves," in Chapter 11.

The NuBus interface supports several address ranges for data transfer between the 32-bit NuBus address space and the 32-bit physical address space (which may be different from the logical space used by software).

MUNI provides separate FIFO buffers for data on the CPU bus and on NuBus. These buffers can operate concurrently. Buffer capacities are shown in Table 2-16.

Table 2-16   MUNI buffer capacities
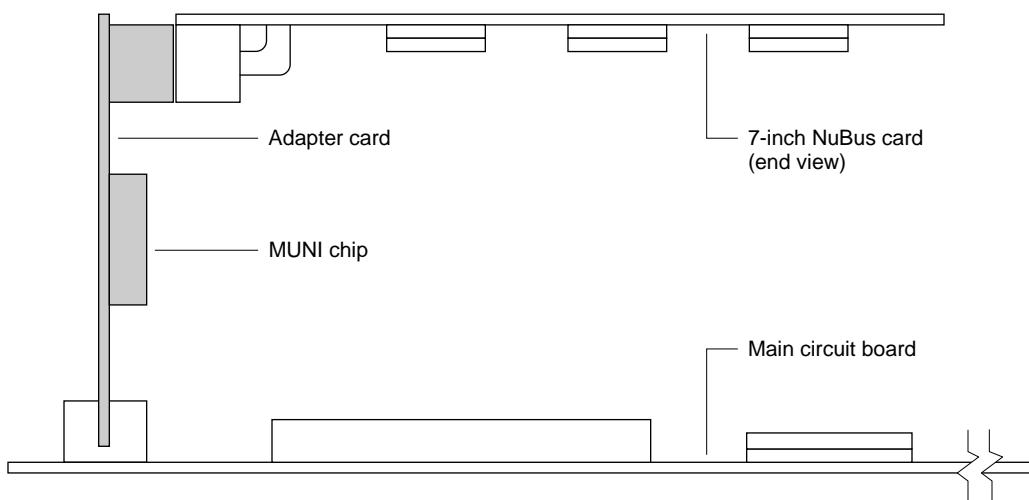
Buffer	Read capacity	Write capacity
CPU bus	4 longwords (1 burst)	16 longwords (4 bursts)
NuBus	16 longwords (1 block 16)	32 longwords (2 block 16s)

Slot Connections

Macintosh Quadra 840AV NuBus slots accept both long (4 by 13 inches) and short (4 by 7 inches) accessory cards of the same physical configuration as those used with the Macintosh II and Macintosh Quadra families. The Macintosh Centris 660AV accepts only short accessory cards, which may be the same as short Macintosh Quadra 840AV cards. For mechanical details of long and short accessory cards, see *Designing Cards and Drivers for the Macintosh Family,* third edition.

The single Macintosh Centris 660AV NuBus slot requires an adapter card that places its NuBus accessory card parallel to the main circuit board, as shown in Figure 2-13. The adapter card carries the MUNI chip, so this chip is present in the system only when the adapter card is installed. For card mounting information, see Appendix D, "Mechanical Details."

Figure 2-13   Macintosh Centris 660AV accessory card mounting

The pin assignments for the 96-pin Euro-DIN NuBus accessory card connectors in the NuBus interface are shown in Table 2-17.

Table 2-17   NuBus pin assignments

Pin	Name	Pin	Name	Pin	Name
a1	–12V	b1	–12V	c1	/RESET[*]
a2	SB0	b2	GND	c2	GND
a3	/SPV	b3	GND	c3	+5V
a4	/SP	b4	+5V	c4	+5V
a5	/TM1	b5	+5V	c5	/TM0
a6	/AD1	b6	+5V	c6	/AD0
a7	/AD3	b7	+5V	c7	/AD2
a8	/AD5	b8	/TM02	c8	/AD4
a9	/AD7	b9	/CM0	c9	/AD6
a10	/AD9	b10	/CM1	c10	/AD8
a11	/AD11	b11	/CM2	c11	/AD10
a12	/AD13	b12	GND	c12	/AD12
a13	/AD15	b13	GND	c13	/AD14
a14	/AD17	b14	GND	c14	/AD16
a15	/AD19	b15	GND	c15	/AD18
a16	/AD21	b16	GND	c16	/AD20
a17	/AD23	b17	GND	c17	/AD22
a18	/AD25	b18	GND	c18	/AD24
a19	/AD27	b19	GND	c19	/AD26
a20	/AD29	b20	GND	c20	/AD28
a21	/AD31	b21	GND	c21	/AD30
a22	GND	b22	GND	c22	GND
a23	GND	b23	GND	c23	/PFW
a24	/ARB1	b24	/CLK2X	c24	/ARB0
a25	/ARB3	b25	STDBYPWR[†]	c25	/ARB2
a26	/GA1	b26	/CLK2XEN	c26	/GA0
a27	/GA3	b27	/CBUSY	c27	/GA2
a28	/ACK	b28	+5V	c28	/START

continued

Table 2-17   NuBus pin assignments (continued)

Pin	Name	Pin	Name	Pin	Name
a29	+5V	b29	+5V	c29	+5V
a30	/RQST	b30	GND	c30	+5V
a31	/NMRQx	b31	GND	c31	GND
a32	+12V	b32	+12V	c32	/CLK

* A slash before a signal name indicates that it is in the low state when active.
† Trickle +5 V supply.

The power available and maximum capacitance loading for each expansion card are shown in Table 2-18.

Table 2-18   Power budget for each slot card

Voltage (V)	Maximum current (A)	Maximum capacitance (μF)
+5	2.0	1513
+12	0.175	536
–12	0.15	698
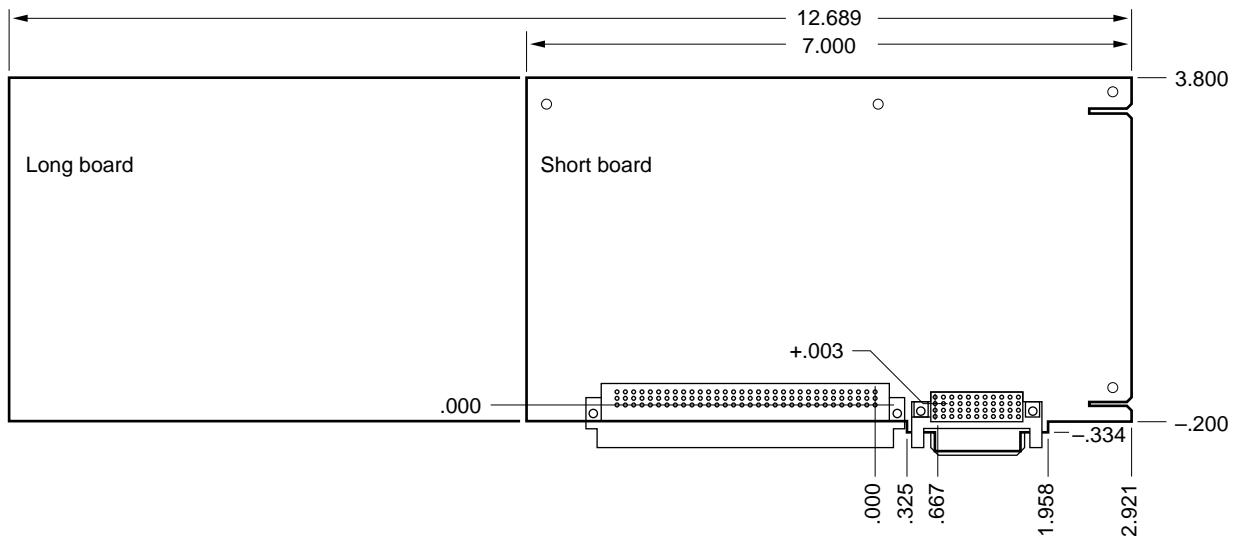
Digital Audio/Video Expansion Connector

In the Macintosh Quadra 840AV, a digital audio/video (DAV) expansion connector is mounted on the main circuit board in line with NuBus slot address $C (the slot nearest the center of the computer), to let an accessory card access sound and video data directly. In the Macintosh Centris 660AV, the DAV connector is mounted on the optional NuBus adapter card (shown in Figure 2-13). Both models can accept a short NuBus accessory card that accesses the DAV connector; the Macintosh Quadra 840AV can also accept a long card.

Figure 2-14 illustrates the lower-right portion of a standard short or long NuBus card that has a connector added to plug into the DAV connector. It shows the mechanical relation between the DAV connector and the normal NuBus connector, with dimensions given in inches.

The DAV connector provides access to the system's 4:2:2 unscaled YUV video input signal and to the digital audio signal input for the Singer codec. One use for this feature is to provide a hardware audio or video compression capability on an accessory card, which could write out compressed data to NuBus. The DAV connector is a 40-pin type, model KEL 8801-40-170L. Table 2-19 gives its pin assignments.

Figure 2-14 DAV connection on a NuBus card

Table 2-19 DAV connector pin assignments

Pin	Signal	Pin	Signal	Pin	Signal
1	Y bit 7	15	Y bit 0	29	UV bit 1
2	LLClk	16	Ground	30	NC (reserved)
3	Y bit 6	17	UV bit 7	31	UV bit 0
4	Ground	18	FEI~	32	Ground
5	Y bit 5	19	UV bit 6	33	SingerSync
6	VS	20	Ground	34	Ground
7	Y bit 4	21	UV bit 5	35	SingerSerOut
8	Ground	22	iicSDA	36	SingerBitClk
9	Y bit 3	23	UV bit 4	37	SingerSerIn
10	HRef	24	Ground	38	Ground
11	Y bit 2	25	UV bit 3	39	Ground
12	Ground	26	iicSCL	40	SingerMClk
13	Y bit 1	27	UV bit 2		
14	vdcCRef	28	Ground		
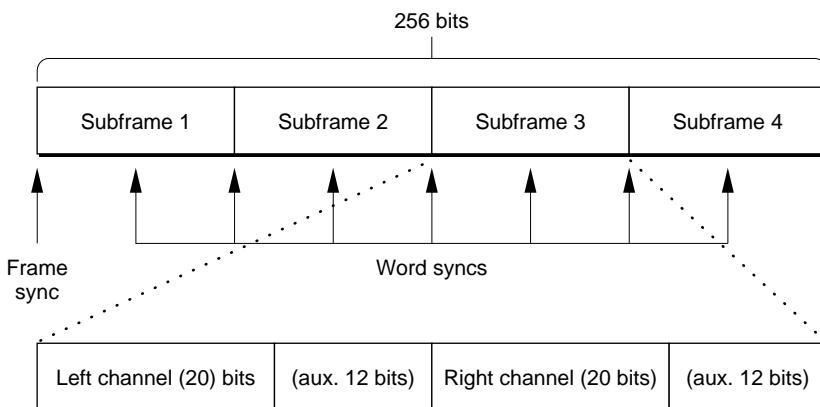
DAV Sound Interface

The Singer sound codec uses time-division multiplexing to transfer multiple audio channels between the DAV connector, the Singer chip, and the PSC for DMA transfers to and from RAM memory. The sound signals that appear at the DAV connector are listed in Table 2-20. These signals have a minimum setup time of 10 ns and a minimum hold time of 8 ns; they can tolerate a maximum load of 20 pF.

Table 2-20   DAV connector sound signals

Signal	Description
singerMClk	24.576 MHz master clock
singerBitClk	Bit clock that clocks serial data on singerSerOut and singerSerIn; 256 times the sample rate; also used to clock singerSync
singerSync	Signal that marks the beginning of a frame and a word
singerSerOut	Sound output from PSC to DAV connector
singerSerIn	Sound input from DAV connector to PSC
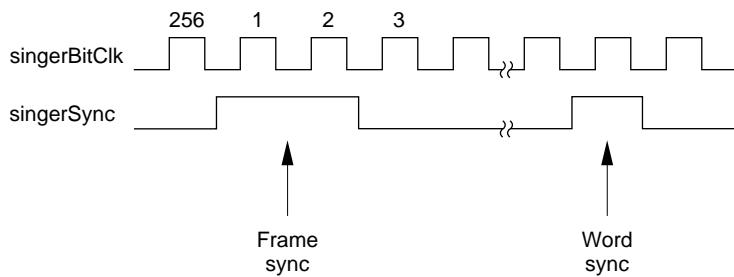
The Singer codec transfers data in 256-bit frames, each of which contains four subframes of 64 bits each. Each subframe carries two 32-bit audio samples, one left and one right. Each sample contains 20 data bits and 12 auxiliary bits. Subframe 1 is reserved for the Macintosh system sound I/O; the other subframes are available for applications and accessory cards to use. The Singer frame structure is shown in Figure 2-15

Figure 2-15   Singer sound frame
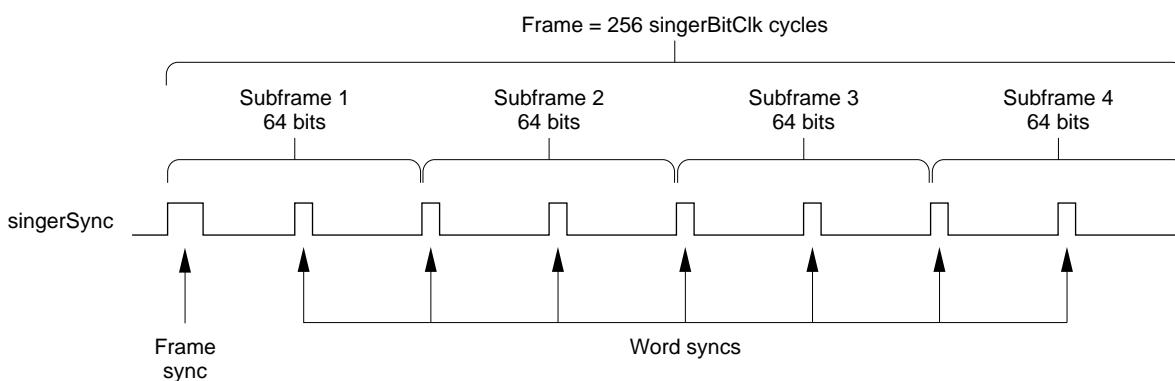
The signals singerSync, singerSerOut, and singerSerIn are clocked by the singerBitClk
signal. The falling edge of the clock is used to clock the signals, and the rising edge is
used to sample them. As shown in Figure 2-16, a frame sync is marked by a pulse two
singerBitClk cycles wide; a word sync is marked by a pulse one singerSync cycle wide.

Figure 2-16   Sound frame and word synchronization

The singerSync synchronization signals for each subframe are shown in Figure 2-17.
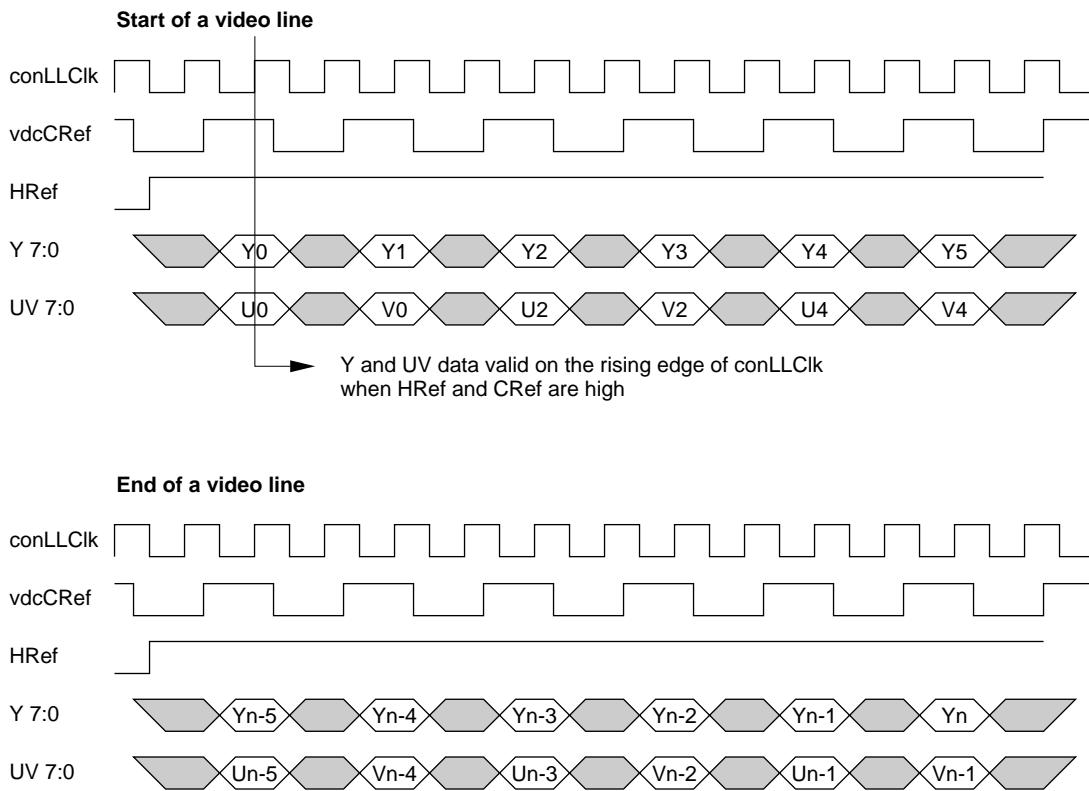
Figure 2-17   Sound subframe synchronization

DAV Video Interface

At the DAV connector, the digital video signal data format conforms to CCIR
Specification 601 and is compatible with most video compression chips. In the DAV
interface, video lines are defined by the HRef signal; it goes high during the image
transmission and low during the blanking interval. The DAV video signal timing
relations are shown in Figure 2-18.

Figure 2-18   DAV video timing

Start of a video line

conLLClk

vdcCRef

HRef

Y 7:0   Y0   Y1   Y2   Y3   Y4   Y5

UV 7:0   U0   V0   U2   V2   U4   V4

Y and UV data valid on the rising edge of conLLClk
when HRef and CRef are high

End of a video line

conLLClk

vdcCRef

HRef

Y 7:0   Yn-5   Yn-4   Yn-3   Yn-2   Yn-1   Yn

UV 7:0   Un-5   Vn-4   Un-3   Vn-2   Un-1   Vn-1

Processor-Direct Cards for the Macintosh Centris 660AV

The Macintosh Centris 660AV (but not the Macintosh Quadra 840AV) can accept an accessory card that plugs directly into the main circuit board instead of into the adapter card shown in Figure 2-13. An accessory card plugged into the main circuit board can gain access to the processor as well as to the DAV bus. The resulting processor-direct slot (PDS) capability is similar to that of the Macintosh Centris 610 computer, described in the *Macintosh Centris 610 Developer Note.*

The Macintosh Centris 610 computer uses an AMP type 650231-5 connector for PDS cards; the Macintosh Centris 660AV uses an AMP type 650231-3 connector. Because the corresponding pins are aligned, it is possible to design PDS cards that work on both models.

The Macintosh Centris 660AV PDS Connector

The pin assignments for the Macintosh Centris 660AV PDS connector are given in
Table 2-21. Pin numbers preceded by an asterisk have signals that are different from
those in the Macintosh Centris 610. Pin numbers preceded by a minus sign are not used
by the Macintosh Centris 660AV.

Table 2-21   Macintosh Centris 660AV PDS connector pin assignments

Pin number	Signal name	Pin number	Signal name
1	GND	26	+5 V
2	A(1)	27	D(19)
3	A(3)	28	D(17)
4	A(4)	29	GND
5	A(6)	30	D(14)
6	A(7)	31	D(13)
7	A(9)	32	D(11)
8	A(11)	33	D(9)
9	A(13)	34	D(8)
10	A(15)	35	D(6)
11	GND	36	D(4)
12	A(18)	37	+5 V
13	A(19)	38	D(1)
14	A(21)	39	GND
15	A(23)	40	SIZE(1)
16	A(24)	41	RW
17	A(26)	–42	/TIP.CPU[*]
18	A(29)	*43	GND[†]
19	A(31)	44	/TEA
20	D(31)	*45	/NC
21	D(29)	*46	GND
22	D(27)	47	/TRST
23	D(25)	–48	/CI.OUT
24	D(24)	49	GND
25	D(22)	*50	NC

continued

Table 2-21   Macintosh Centris 660AV PDS connector pin assignments (continued)

Pin number	Signal name	Pin number	Signal name
–51	/BR.40SLOT	82	A(17)
52	/BB	83	+5 V
53	/LOCK	84	A(20)
54	/MEM.RESET	85	A(22)
55	/CPURESETOUT	86	GND
56	+5 V	87	A(25)
*57	040INPROGRESS	88	A(27)
58	/NMRQ(6)	89	A(28)
59	GND	90	A(30)
60	/IPL(0)	91	D(30)
61	/IPL(1)	92	D(28)
62	/IPL(2)	93	D(26)
63	–12V	94	GND
64	GND	95	D(23)
*65	NC	96	D(21)
*66	NC	97	D(20)
*67	/NMRQ(5)	98	D(18)
*68	/NMRQ(4)	99	D(16)
*69	/040LOCKE	100	D(15)
70	+5 V	101	+5 V
71	AUX.CPUCLK	102	D(12)
72	A(0)	103	D(10)
73	A(2)	104	GND
74	+5 V	105	D(7)
75	A(5)	106	D(5)
76	GND	107	D(3)
77	A(8)	108	D(2)
78	A(10)	109	D(0)
79	A(12)	110	SIZE(0)
80	A(14)	111	+5 V
81	A(16)	*112	+5 V

continued

Table 2-21   Macintosh Centris 660AV PDS connector pin assignments (continued)

Pin number	Signal name	Pin number	Signal name
113	/TA	127	/SYS.RESET
114	GND	–128	TM(0)
115	/TS	–129	TM(1)
*116	GND	–130	TM(2)
*117	+5 V	131	+5 V
*118	+5 V	*132	NC
–119	/BG.40SLOT	133	+12V
120	/BG.CPU	134	GND
121	+5 V	135	BS.CLK
122	TT(0)	136	/BS.MODE
123	TT(1)	*137	MUNI/RQ
124	GND	*138	NC
–125	TLN(0)	139	reserved
–126	TLN(1)	140	+5 V

[*] A slash before a signal name indicates that it is in the low state when active.
[†] GND on pin 43 identifies the Macintosh Centris 660AV; on the Macintosh Centris 610 pin 43 is not connected.

Most of the signals listed in Table 2-21 are connected directly to the computer's processor. Table 3-22 lists the PDS signals that are connected to the computer's processor but that should not be connected to a processor on a PDS card. Table 3-23 lists the PDS signals that are not directly connected to the computer's processor.

Table 3-22   Restricted microprocessor signals on the PDS connector

Signal name	Direction	Function
/IPL(0–2)	I[*]	Interrupt priority lines from the PSC; not to be used as wire-OR lines; can be monitored by a PDS card
/TIP.CPU	I	From the MC68040 on the main circuit board; not connected to any other part of the computer

[*]I indicates input to the PDS card.

Observe the following additional cautions when designing PDS cards for the Macintosh Centris 660AV:

Table 3-23   Nonmicroprocessor signals on the PDS connector

Signal name	Direction	Function
/SYS.RESET	I/O[*]	Enables PDS to drive system reset signal; used only for testing
AUX.CPU.CLK	I	Buffered version of main processor's bus clock (BClk)
/BG.CPU	O	Bus grant for main processor
/BG.40SLOT	I	Bus grant for PDS card
/BR.40SLOT	O	Bus request for PDS card
/MEM.RESET	I	Fast reset generated for memory controller IC
/MI.SLOT	O	Memory inhibit from PDS card to memory controller IC
/NMRQ(6)	O	NuBus slot $E interrupt; also connected to NuBus slot $E

[*] I indicates input to the PDS card; O indicates output from the PDS card.

■ Most signals on the PDS connector are connected directly to the main processor with no buffers. Therefore, the PDS card must present capacitive loads of not more than 40 pF on the address, data, and clock lines and not more than 20 pF on the control lines.

■ The AUX.CPUCLK line (pin 71) is terminated with a series resistor. To reduce reflections on this line, all loads on the card should be lumped.

■ /DLE (pin 45) is not connected because DLE-type read actions are not supported by some Macintosh Centris 660AV bus masters.

■ The Macintosh Centris 660AV does not support snooping; pins 46 and 116 are grounded.

■ /BR.CPU (pin 50) is not connected because the Macintosh Centris 660AV system bus arbiter always grants bus control to the microprocessor when there are no other high-priority bus requests.

■ The Macintosh Centris 660AV does not support /BG.40SLOT (pin 119) and /BR.40SLOT (pin 51) because it does not support using an accessory card as a bus master in addition to the existing bus masters (the processor, the DSP, the PSC, and the MUNI).

■ /TBI (pin 112) is connected to +5 V because the TBI signal is not allowed by some bus masters.

■ /PDS.SLOT.E.EN (pin 132) is not connected because the MUNI is programmed to decode or ignore individual slots.
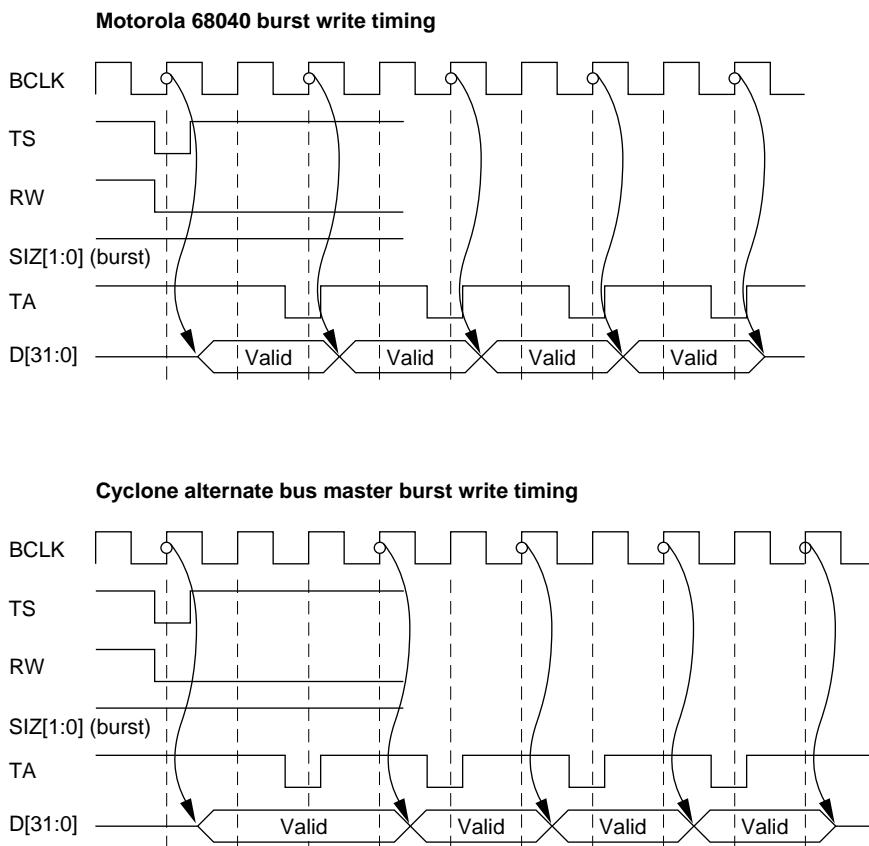
The 040INPROGRESS signal (pin 57) is high when the main processor is the bus master. When this signal is low, a different bus master can use the alternate burst write timing protocol described in the next section.

Processor Bus Burst Write Timing

The Macintosh Centris 660AV computer's processor bus supports two different timing protocols for burst write actions. When pin 57 of the PDS connector is high, the main processor is bus master and burst write actions must use the timing shown in the top half of Figure 2-19. When pin 57 of the PDS connector is low, an alternate bus master may perform burst write actions using the timing shown in the bottom half of Figure 2-19.

Figure 2-19   Burst write timing

Motorola 68040 burst write timing

Cyclone alternate bus master burst write timing

RAM Expansion Cards

The user can expand RAM capacity by inserting 72-pin SIMM cards in RAM expansion slots. Table 3-24 shows the RAM SIMM pin assignments.

Table 3-24   RAM SIMM pin assignments
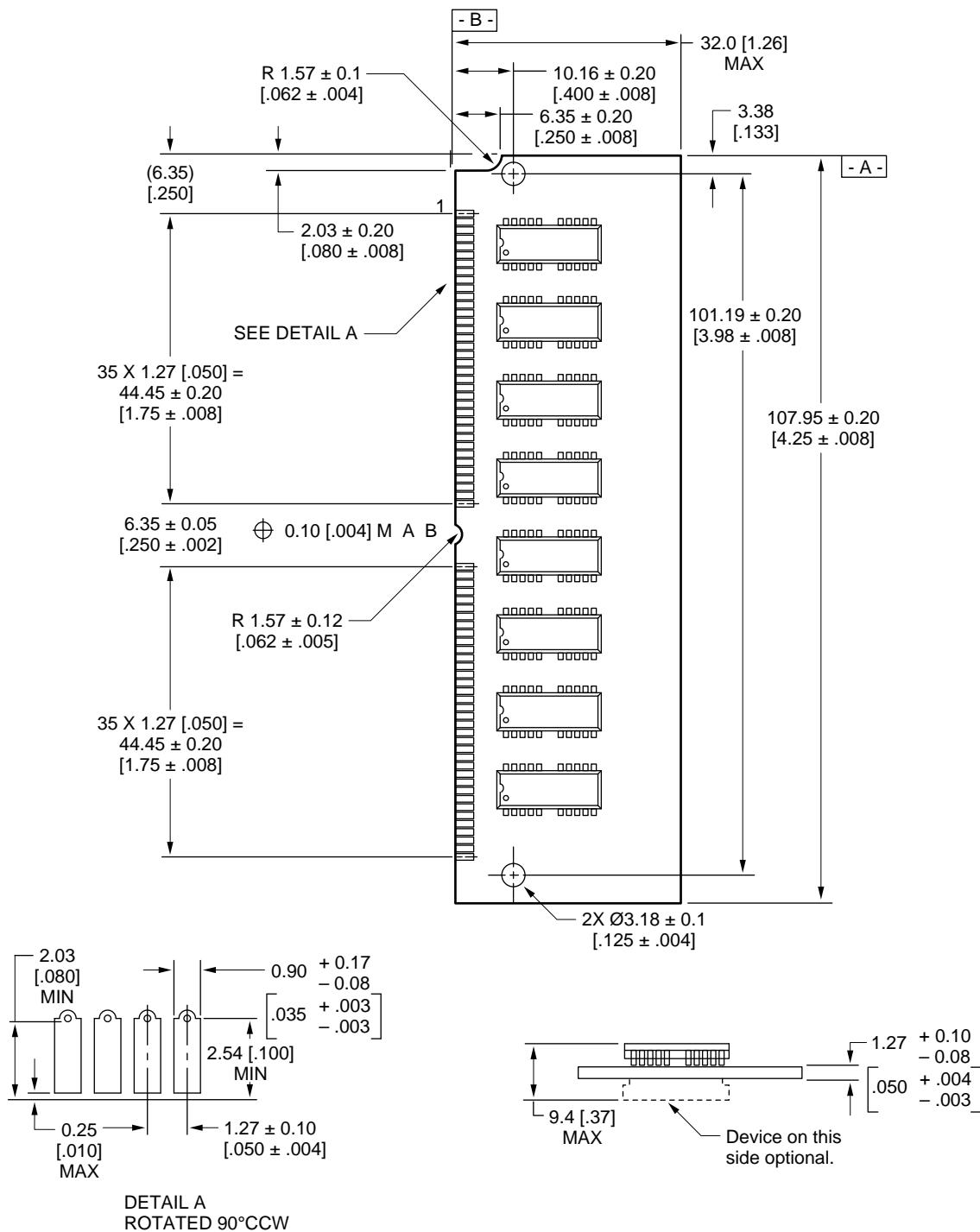
Pin	Name	Pin	Name	Pin	Name
1	GND	25	DQ22	49	DQ8
2	DQ0	26	DQ7	50	DQ24
3	DQ16	27	DQ23	51	DQ9
4	DQ1	28	A7	52	DQ25
5	DQ17	29	NC	53	DQ10
6	DQ2	30	+5V	54	DQ26
7	DQ18	31	A8	55	DQ11
8	DQ3	32	A9	56	DQ27
9	DQ19	33	/RAS3*	57	DQ12
10	+5V	34	/RAS2	58	DQ28
11	NC	35	Reserved	59	+5V
12	A0	36	Reserved	60	DQ29
13	A1	37	Reserved	61	DQ13
14	A2	38	Reserved	62	DQ30
15	A3	39	GND	63	DQ14
16	A4	40	/CAS0	64	DQ31
17	A5	41	/CAS2	65	DQ15
18	A6	42	/CAS3	66	NC
19	NC	43	/CAS1	67	Reserved
20	DQ4	44	/RAS0	68	Reserved
21	DQ20	45	/RAS1	69	Reserved
22	DQ5	46	NC	70	Reserved
23	DQ21	47	WE	71	Reserved
24	DQ6	48	NC	72	GND

* A slash before a signal name indicates that it is in the low state when active.

Figure 2-20 shows the mechanical dimensions of SIMM modules for expanding RAM. Dimensions are given in millimeters, with inch equivalents in brackets.

Figure 2-20   RAM SIMM mechanical dimensions
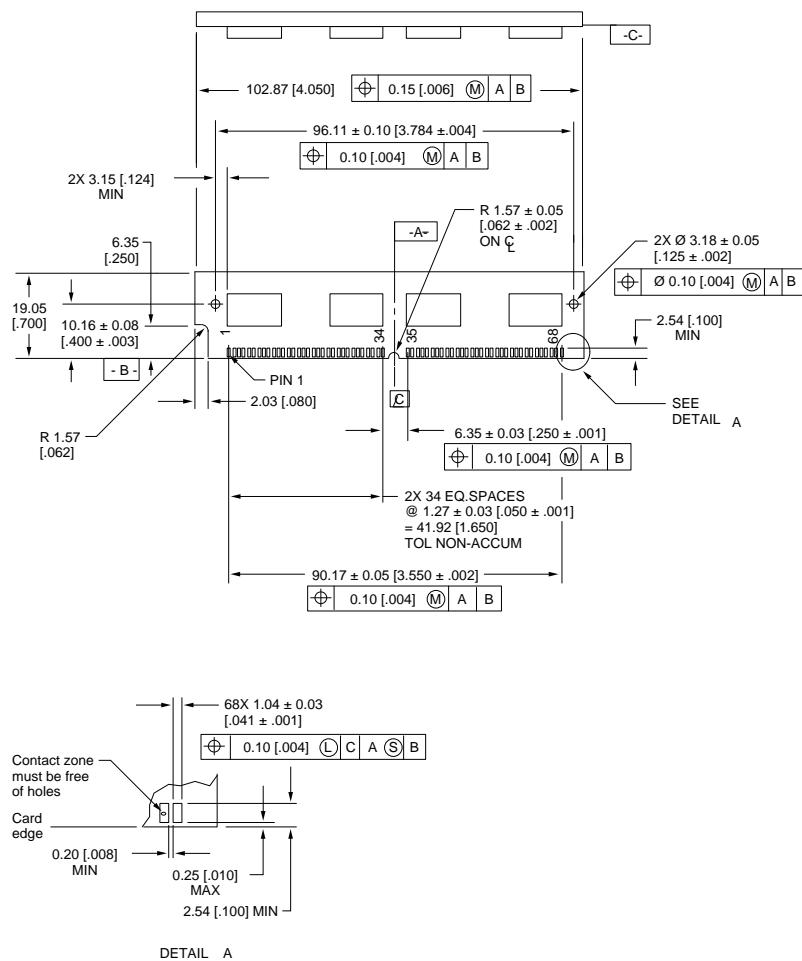
Because of signal loading limits, there may not be more than eight chips per bank of RAM; composite SIMM cards cannot be used.

The Macintosh Quadra 840AV also accepts SIMM cards of a different configuration to expand its VRAM, as shown in the next section.

VRAM Expansion Cards

The Macintosh Quadra 840AV lets the user expand VRAM capacity by inserting 68-pin SIMM cards in its two VRAM expansion slots. Figure 2-21 shows the mechanical dimensions of SIMM modules for expanding VRAM, which are different from the RAM cards discussed in the previous section. Dimensions are given in millimeters, with inch equivalents in brackets.

Figure 2-21   VRAM SIMM mechanical dimensions

VRAM SIMM pin assignments are shown in Table 3-25.

Table 3-25   VRAM SIMM pin assignments

Pin	Name	Pin	Name	Pin	Name
1	+5V	25	DQ5	49	A7
2	DSF	26	SDQ7	50	A8
3	SDQ0	27	SDQ6	51	NC
4	SDQ1	28	NC	52	+5V
5	/DT-OE0[*]	29	+5V	53	GND
6	DQ0	30	DQ7	54	GND
7	DQ1	31	DQ6	55	SDQ12
8	SDQ3	32	/CAS0	56	SDQ13
9	SDQ2	33	A4	57	NC
10	/WE0	34	A5	58	DQ12
11	/RAS	35	GND	59	DQ13
12	/SE0	36	SC	60	SDQ15
13	DQ3	37	SDQ8	61	SDQ14
14	DQ2	38	SDQ9	62	NC
15	A0	39	/DT-OE1	63	NC
16	A1	40	DQ8	64	DQ15
17	A2	41	DQ9	65	DQ14
18	A3	42	SDQ11	66	/CAS1
19	GND	43	SDQ10	67	GND
20	GND	44	/WE1	68	GND
21	SDQ4	45	/SE1		
22	SDQ5	46	DQ11		
23	NC	47	DQ10		
24	DQ4	48	A6		

[*] A slash before a signal name indicates that it is in the low state when active.

VRAM access times in numbers of clock cycles are shown in Table 3-26. The Random columns in Table 3-26 show the times required for random accesses; the Second columns show the times required for immediately succeeding accesses.

Table 3-26   VRAM access times

	Macintosh Quadra 840AV		**Macintosh Centris 660AV**	
Access type	Random	Second	Random	Second
Single write	5	7	3	4
Burst write	5-3-3-3	7-3-3-3	3-2-2-2	4-2-2-2
Single read	7	7	4	4
Burst read	7-3-3-3	7-3-3-3	4-2-2-2	4-2-2-2
VDC write	19	20	19	20

Real-Time Data Processing

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* covers the software technology of the Macintosh Quadra 840AV and Macintosh Centris 660AV digital signal processing facilities. It contains three chapters:

- Chapter 3, "Introduction to Real-Time Data Processing," describes the software architecture of the real-time data processing facility in the Macintosh Quadra 840AV and Macintosh Centris 660AV. This facility consists of an AT&T DSP3210 chip that performs data-processing operations for applications that contain digital signal processor (DSP) code.

- Chapter 4, "Real Time Manager," describes a new part of the Macintosh system software that supplies all the services an application requires to use the DSP, including loading and running DSP code and performing DSP memory management.

- Chapter 5, "DSP Operating System," covers the DSP operating system, contained in the DSP chip. It provides the services every DSP program needs to work with the Macintosh Operating System.

Introduction to Real-Time Data Processing

This chapter describes the new real-time data processing software architecture for the Macintosh Quadra 840AV and Macintosh Centris 660AV computers, including the functional specifications, features, programming interface, capabilities, and performance. For hardware information about these computers' DSP implementation, see Chapter 2, "Hardware Details."

For the novice in digital signal processing, this chapter begins with an overview of the AT&T DSP3210 digital signal processor and the architecture of real-time data processing. It provides the basics for understanding the rest of the chapter, which provides a more complete discussion of all the concepts and fuller architectural details.

The serious programmer of real-time data processing should read this entire chapter. You must understand several concepts introduced in the section "Real-Time Processing Architecture" to handle real-time programming and data flow properly.

Other parts of this book supplement this chapter. Chapter 4, "Real Time Manager," provides information to the Macintosh programmer and can be skipped by the DSP programmer. Chapter 5, "DSP Operating System," provides information to the DSP programmer and can be skipped by the Macintosh programmer. However, for a complete understanding of the interrelationships and dependencies between the two types of programming anyone doing system debugging or integration should read both chapters. For information about installing and debugging DSP programs in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Appendix A, "DSP d Commands for MacsBug," Appendix B, "BugLite User's Guide," and Appendix C, "Snoopy User's Guide."

Introduction to Digital Signal Processors

Real-time data processing requires a hardware and software architecture for integrating digital signal processing technology into the Macintosh Quadra 840AV and Macintosh Centris 660AV computers. The architecture supports the computer's digital signal processor as a coprocessor that has its own operating system but is capable of accessing the same data memory as the main processor.

Concepts of Digital Signal Processing

Digital signal processing is the manipulation and conversion of digitized data. Digitized data are digital representations of analog signals, which may represent sounds, images, speech, or other analog forms. To correctly process these signals it is necessary to know at what rate they were converted (the sample rate) and the format of the digital bits used to represent the original data. With this information the signal can be manipulated by a conventional program using the digitized data as its input. The result can then be stored on disk or converted back into an analog signal.

All such processing accomplished by a computer is called digital signal processing. The digital signal processor supports the math routines required in a special chip designed specifically for signal processing applications. The multiply/accumulate operation is the basic ingredient of signal processing programs. The digital signal processor is designed to perform this operation very rapidly.

The equivalent of digital signal processing in the analog domain is accomplished using electronic components, such as inductors, capacitors, resistors, and transistors. The advantage of doing the processing in the digital domain is that the functions can be very precise, reliable, elaborate, and software-configurable. It is difficult and costly to achieve these same goals in the analog domain.

Real-Time Processing Capability

The Macintosh Quadra 840AV and Macintosh Centris 660AV computers' real-time capability uses a multi-tasking coprocessor to give high-performance processing of sound, communications, speech, and images (both graphic and video) while utilizing the system's low-cost dynamic random-access memory (DRAM) for primary storage of data and code. The standard hardware is the AT&T DSP3210 and the audio and telephone input/output (I/O) ports. The software is a custom operating system designed to perform isochronous (*real-time*) and asynchronous (*timeshare*) algorithms. The operating system is based on a *team processing* approach where the work of the total system is carefully separated and delegated between the main processor and the digital signal processor.

This approach has the benefit of

- greatly reducing implementation and hardware costs

- simplifying and speeding up interprocessor communications and data sharing or data streaming

- allowing flexible dynamic load sharing between the main processor and the DSP on selected algorithms

- maximizing the potential to meet future needs for higher performance and multiple coprocessors

- increasing the range of possible application functions the DSP can provide
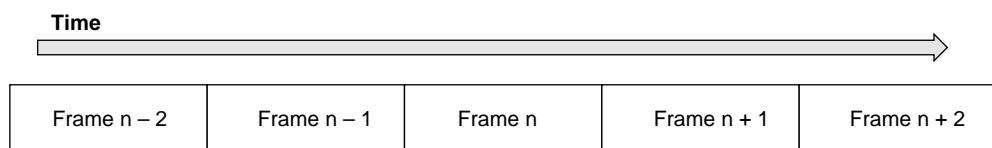
The DSP software architecture supports *dual threaded* processing streams. **Real-time processing** uses interrupt-level isochronous algorithms with guaranteed processing bandwidth to execute real-time functions requiring precisely timed signal generation or inputs such as sound and communications. (Guaranteed processing bandwidth is defined in the next section.) **Timeshare processing** uses asynchronous algorithms that employ the excess DSP bandwidth for functions not requiring time-correlated processing, such as still image decompression or scientific computing.

Additionally, the architecture supports the implementation of NuBus cards to make configurations of multiple DSPs possible.

Real-Time Processing Architecture
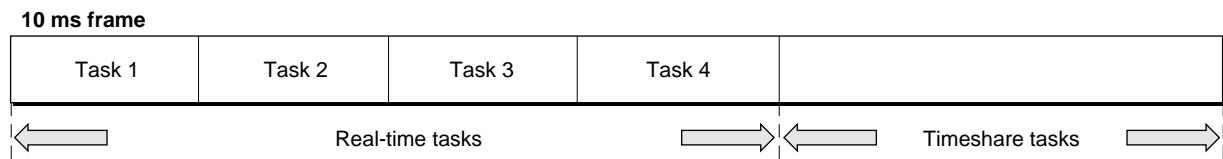
Program execution on the DSP is divided into segments of time called **frames**, typically 10 ms in length, as diagrammed in Figure 3-1. During each frame an attempt is made to run all of the code that is installed on the DSP. **Tasks** are blocks of DSP code that are grouped together by the programmer to perform a specific function.

Figure 3-1   Frames

Time

Frame n – 2	Frame n – 1	Frame n	Frame n + 1	Frame n + 2
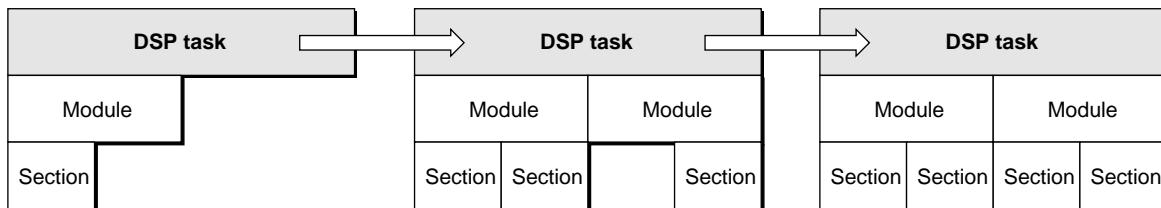
There are two types of tasks: real-time and timeshare. During each frame all of the real-time tasks are executed and then any remaining time in the frame is used for executing timeshare tasks, as diagrammed in Figure 3-2. Real-time tasks are useful for sound, modem, and video processing where there is a fixed amount of data that must be processed during each frame; if more processing time were available it would not be used. However, timeshare tasks use as much processing power as they can get each frame. Image decompression is an example of a timeshare task, since it should decompress the image as fast as possible. This means that when a faster version of the DSP3210 is available timeshare tasks run faster but real-time tasks continue to process the same amount of data.

Figure 3-2   Real-time and timeshare tasks

10 ms frame

Task 1	Task 2	Task 3	Task 4	

Real-time tasks   Timeshare tasks

Each task is assembled out of **modules**, which are the functions that the DSP programmer creates, and each module is composed of **sections**. This relationship is shown in Figure 3-3.

Figure 3-3   Task list

To understand the need for sections, it is necessary to understand how the memory system of the DSP works. To keep hardware costs down, the DSP uses the same DRAM as the main processor. Because the DSP can access memory at a much higher rate than the RAM can provide, and must also compete with the main processor for RAM access, some type of caching on the DSP is needed. The DSP does not have a hardware cache like that in the 68040 main processor. It has a small amount of memory on the DSP chip that is accessed in the same way as main RAM. It is called on-chip memory, in contrast to main memory, which is off-chip. The lack of DSP hardware caching means that caching must be managed by the DSP program and the DSP operating system. This is called **visible caching** as opposed to the transparent operation of most main processor caches.

To accomplish visible caching, the DSP programmer must mark which sections of the code are loaded in on-chip memory before execution and which sections are saved off-chip after execution. Visible caching operates in one of two modes. In **AutoCache** mode, loading and saving are controlled by the DSP operating system; there is only one set of sections on-chip during the execution of a module. In **DemandCache** mode, loading and saving are controlled by the DSP program, so sections can be moved on and off-chip during the execution of the module. Caching modes are discussed in more detail in "Visible Caching" and "Execution Models," later in this chapter.

To make modules slightly more general, a mechanism is provided for a single module to work at different frame rates and sample rates. This is done by making sections individually scalable. The DSP programmer has the option of saying which sections are scalable and the possible sizes of the scalable sections. For example, if a reverberation module works with both 24k Hz and 48k Hz sound at a 10 millisecond frame rate it would have an input and an output section, both of which would be scalable to either 240 or 480 samples per frame. When the Macintosh program loads the module from disk, it specifies the module scale of operation.

To ensure that all of the real-time tasks are executed during each frame, the DSP programmer must specify an upper bound for the execution time of the module. If there is enough processing power on the DSP, the task that contains this module will be installed and executed. As long as every module's estimate is correct, the DSP will execute frames evenly. However, if a module's estimate is not its upper bound, the DSP

could take more time to execute the real-time tasks than is available in a given frame. When this frame overrun occurs the DSP operating system will find the module that specified its incorrect upper bound, remove the task that contains this module from the execution stream, and then resume execution. This procedure is called **guaranteed processing bandwidth (GPB).**

Since a task is made up of modules which typically share data, optimization is provided to keep the data on-chip between modules, instead of saving it off-chip in one module and then loading it back on-chip for the next module. This is accomplished by connecting sections from one module to another, letting the DSP operating system decide if data saving and loading is required. Data that must be shared between tasks, such as the sound going to the speaker, is passed between tasks in **intertask buffers (ITBs).** The only logical difference between ITBs and connected sections is that the sections are in different tasks for ITBs and in the same task for connected sections. Both ITBs and connected sections are managed by the Macintosh programmer, as described in "Data Buffering," later in this chapter.

Software Model

The software model for real-time data processing in the Macintosh Quadra 840AV and Macintosh Centris 660AV computers consists of three distinct pieces:

■ The host toolbox is the **Real Time Manager.** The Real Time Manager runs on the main processor and is written in C for portability.

■ The DSP Driver contains both main processor code and DSP code components. All hardware-dependent functions are included in the drivers. They are written in the 68000 and DSP assembly languages for efficiency.

■ The DSP toolbox is called the **DSP operating system.** The DSP operating system runs on the DSP, and is written in DSP assembly language for efficiency.
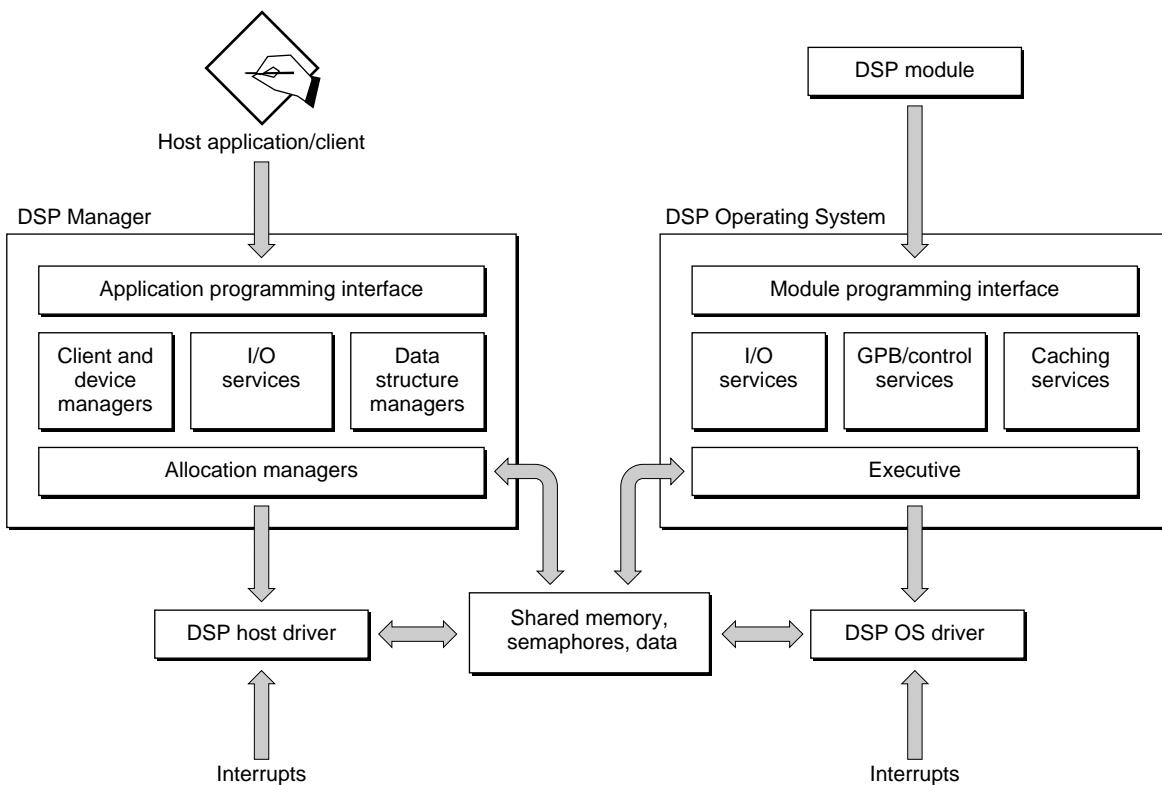
Almost all routines in the Real Time Manager are reentrant and callable from interrupt level. This is necessary, since communications between the DSP and main processor often take the form of interrupt messages.

A major component of the model is a shared block of memory. This memory consists of local memory as well as main memory. The local memory is either in system DRAM or in optional card memory. It is through data structures and semaphores in this shared memory that the main processor and DSP toolboxes communicate. A more complete diagram of the software model is shown in Figure 3-4.

Dual Programming Model

Figure 3-4 shows the dual programming interface for real-time data processing: the application programming interface (API) in the Real Time Manager, and the module programming interface (MPI) in the DSP operating system. These two interfaces are completely separate, and designed to be used by different programmers. It is not necessary for a programmer to be both a Macintosh programmer and a DSP programmer.

Figure 3-4   Real-time data processing organization

It is usually better to have two programmers involved when programming an application that requires DSP modules. This is because the two types of programming are very different, and very specialized. The two programmers communicate with each other by creating a *DSP Module Specification* document. This document provides a vehicle for transferring all the information necessary to ensure a correct interface between the main processor program and each DSP module. For more information about the data this document should contain see "DSP Program Information for the Macintosh Programmer," in Chapter 5.

Real Time Manager

The Real Time Manager uses the standard trap interface to call the Macintosh Toolbox. The set of calls accessible to an application are labeled as the application programming interface layer in Figure 3-4.

Three major functions of the Real Time Manager support I/O services, client and device management, and data structure management. These functions make calls on the Real Time Manager's allocation routines at the lowest level.

The allocation layer is responsible for DSP cache and local memory allocation, for GPB allocation, and for I/O resource allocation.

DSP Operating System

The DSP operating system also has an interface layer. This layer works in a similar fashion to the Real Time Manager: a trap mechanism is used to make calls on the DSP operating system from the DSP module.

The DSP operating system also provides services to the DSP module: I/O services, including FIFO management, GPB and control services, and caching operations on the DSP. The underlying function of the DSP operating system is contained in an executive layer, which is responsible for managing task-sequencing and frame-handling functions.

DSP Driver

The DSP Driver has two distinct components. One works exactly like a standard Macintosh driver, and is written in 68000 code. The other component performs a similar function for the DSP operating system. It contains all DSP code that is hardware-dependent, as well as booting and restart code These two components are stored together as one driver. The DSP driver also controls the I/O drivers for any serial or parallel I/O ports included as part of the DSP system. These resources are accessed using the Real Time Manager services.

Other Software Components

Additional system software that supports real-time data processing includes:

■ A *sound driver* provides the interface between the Macintosh Sound Manager and the Real Time Manager by means of a set of *standard sound* modules, including sound input and output, compression, filtering, sample rate conversion, and mixing.

■ A *telecom driver* provides the interface between the telecommunications Manager/ Communications Toolbox and the Real Time Manager, including a set of standard telecom modules, plus modem, fax, and speech.

■ *Development tools* include a DSP C compiler, assembler, libraries, linker, resource generator, and include-files with macros and definitions.

■ *Debugging and test tools* include a graphical module installer, DSP code debugger, and MacsBug extensions.
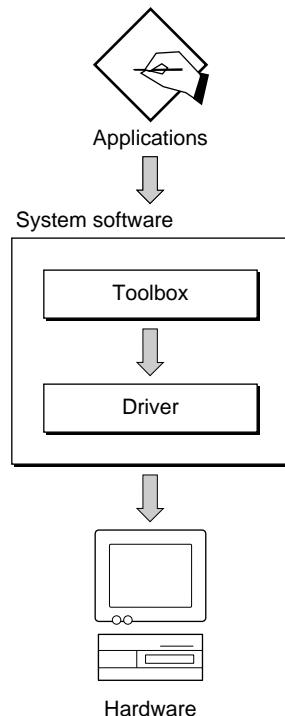
The purpose of the various toolbox drivers is to provide access to the capability of the DSP at the highest possible toolbox level. This allows applications that are not written for the DSP to use it automatically when it is available. Even with this level of toolbox support, it is clear that many applications will work better by directly accessing the DSP using the DSP API. Such applications provide significantly more functionality or speed when a DSP is available. However, an application that uses the DSP API either cannot run on a platform without the DSP, or must provide alternative main processor programming if a DSP is not available.

Software Layers

The basic Macintosh software model has four primary conceptual layers: the application layer on top, the toolbox layer, the driver layer, and finally the hardware layer. The separation of system software into toolbox and driver layers allows the separation of hardware dependencies from the major system functions, and makes revisions in the hardware easier to support. If this model is followed correctly, major changes in the hardware can be made without breaking applications. For this reason, Apple encourages developers to access functions at the highest possible toolbox layer, even if they could be more efficient writing directly to the hardware. This separation allows Apple to improve the hardware base without disrupting the application base. A diagram of the four-layer Macintosh model is shown in Figure 3-5.

Figure 3-5   Four-layer Macintosh model
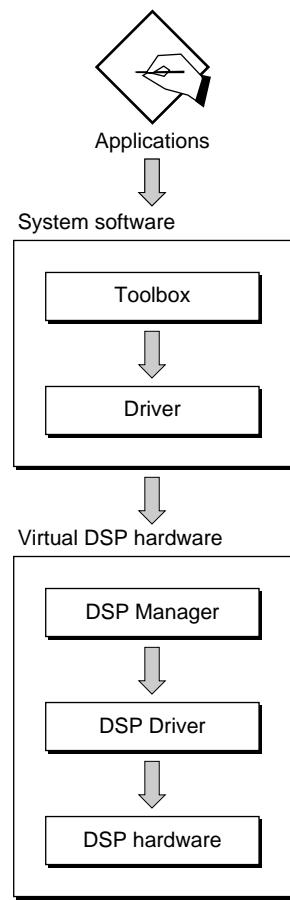
As shown in Figure 3-5, an application that accesses the Real Time Manager is hardware-dependent. This means the application would require that a DSP coprocessor be present in the system in order for it to operate. This is true even though the Real Time Manager is hardware-independent. The emphasis here is on implementation. The Real Time Manager assumes that there is a DSP available, otherwise there is no reason for the manager to be installed. Additionally, it provides the necessary isolation from the specific implementation details. By accessing a higher toolbox layer the application also becomes DSP-independent and will operate across multiple Macintosh platforms.

Introduction to Real-Time Data Processing

If the original Macintosh model is combined with the DSP model, the DSP software and hardware must be viewed as *virtual hardware*. This concept is illustrated in Figure 3-6.

Figure 3-6   Six-layer model
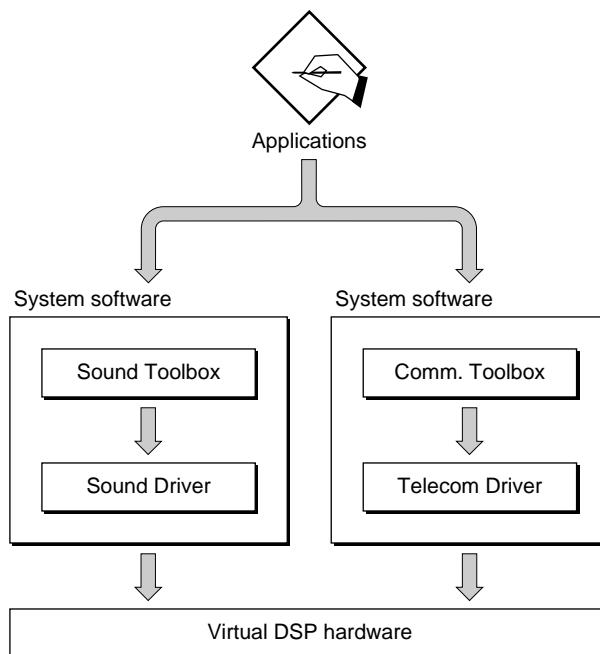
The model shown in Figure 3-6 is used for the DSP software. Notice that the driver layer is specific for the virtual hardware. If the DSP is available, this layer must be able to install tasks in the task list and must deal with any specific characteristics of this machine that may affect its operation. If there are no such characteristics, then the driver is not dependent on the machine implementation, but only on the availability of the DSP. In either case, the driver is specific for the virtual hardware.

Figure 3-7 shows two sample toolbox/driver combinations for the Real Time Manager.

In the case of sound, there are no hardware-specific features that the Sound Driver needs to deal with. Hence only a single-layer driver is needed. The driver is capable of working with the DSP in any supported configuration, and does not need to deal with specific implementation details. This results in a six-layer model.
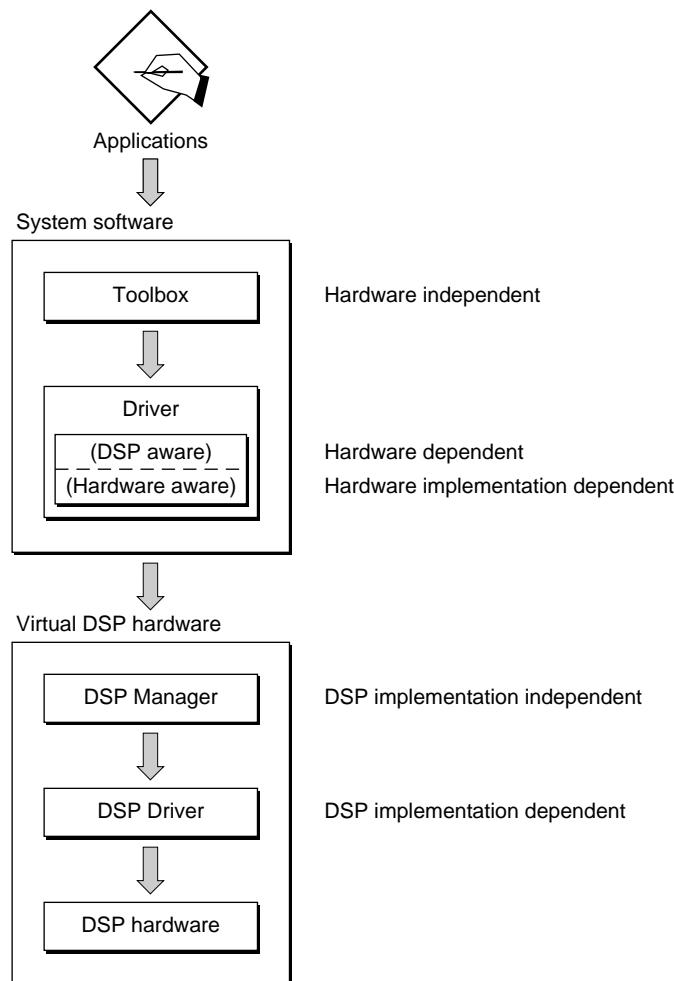
Figure 3-7 Example of toolbox and driver layers

For the communications case, the Telecom Driver deals specifically with the way that the DSP I/O subsystem is connected to the telephone line. Thus, specific bit input and output (BIO) pins on the DSP perform functions that the Telecom Driver uses. The driver takes control of these functions if the appropriate external hardware is present on the telecom port. This makes the Telecom Driver hardware-specific relative to the telecom subsystem. It is also hardware-dependent on the DSP virtual hardware.

To the extent that the same configurations are used for all CPUs and cards, the Telecom Driver becomes universal, and seemingly hardware-independent. However, different arrangements of telecom subsystems for different implementations of the DSP will require a different telecom driver. Notice that a different telecom driver must be supplied for a NuBus card and for a CPU, even if the configuration is identical. This is because the CPU Driver can recognize a specific CPU but cannot recognize a specific NuBus card. If the wiring of the I/O subsystem is identical in both cases, then the only change to the driver is the hardware recognition code.

To facilitate this, the driver layer should be divided into two separate parts: the DSP-handling layer on top that uses Real Time Manager routines, and the hardware-specific layer on the bottom that deals with specific hardware wiring. This allows simple modification of the driver to support different hardware platforms. This arrangement is shown in Figure 3-8.

Figure 3-8   Seven-layer real-time model

The addition of this seventh "H/W driver" layer is only necessary if the driver requires specific access to I/O subsystems.

DSP-Aware Applications

A DSP-aware application can be designed to operate in two different ways:

- to recognize and use the DSP if it is there, for enhanced performance of specific application functions
- to require the DSP and not run at all if no DSP is available

There are many interesting applications in both categories. It is important to realize that the Real Time Manager's implementation independence makes it possible to write a DSP-aware application that will run, without change, on different DSP implementations,

assuming the same (instruction set compatible) DSP is used. Such an application can make direct calls to the Real Time Manager for service. Different instruction sets can be supported by the appropriate processing modules.

It is important to note that if a desired function is available from a high-level toolbox then the DSP connection will be made automatically, providing enhanced performance without the application being written for the DSP. A good example of this is any application that plays sound. If it calls the Sound Manager then the processing will be handed over to the DSP. However, if a sound application needs more service than the Sound Manager provides then the application should directly access the Real Time Manager. Depending on the application, either of these DSP-aware models could be used.

Software Architecture
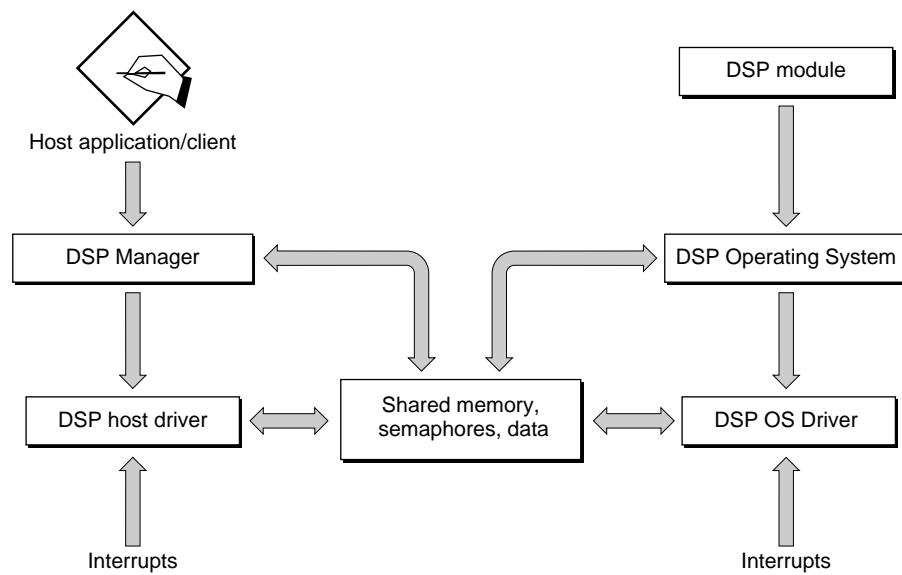
The real-time data processing software is based on a data flow model. It is important for a real-time signal processing system to accept and process incoming samples at the average rate that they are being produced by the input process. It is equally important for it to create outgoing samples at the average rate that they are being consumed by the output process.

By buffering the samples, it is possible to process groups of samples at a time rather than single samples at a time. This approach is called **frame-based processing.** During each frame the application loads the required program code, variables, and input data into a high-speed cache on the DSP. The program code is executed from this cache, and the resulting output data is dumped from the cache back into off-chip memory. Alternately, the input data may already be in the cache from a previous operation, and the output data may be kept in the cache if it is needed for following operations.

The operating software for real-time data processing works on a team processing basis. In particular, careful attention has been paid to the division of labor between the main processor and the DSP. The goal is to maximize the processing throughput of the DSP while minimizing the processing requirements and bus loading of the main processor. The operating software consists of a part of the Macintosh toolbox (the Real Time Manager and its driver) and a DSP control program (the DSP operating system and its driver). A block diagram of this concept is shown in Figure 3-9.

These two programs interact with one another through shared memory, interrupt processing, and semaphores. The Real Time Manager supports application software on the main processor, while the DSP operating system supports DSP program modules on the DSP. Thus, there are two completely separate application program interfaces in real-time data processing: one for the main processor program and one for the DSP program.
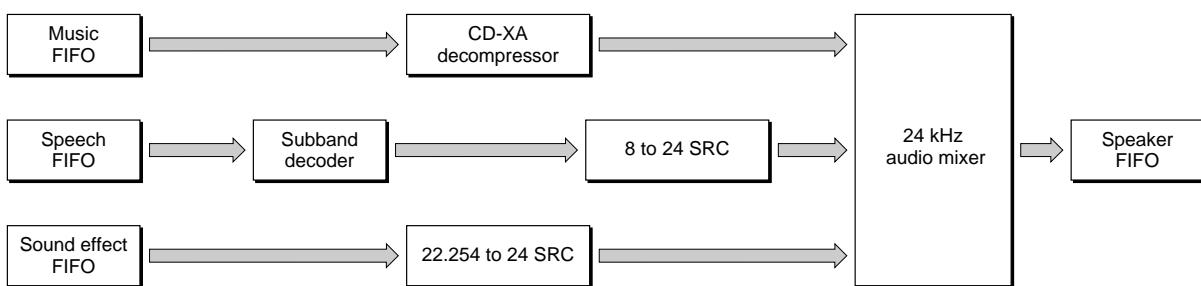
Introduction to Real-Time Data Processing

Figure 3-9 Real-time software organization

In most applications the DSP will need to run several different code modules or algorithms in sequence to process blocks of data. For example, five different DSP modules are required for a sound player to mix the following three channels of sound:

■ compressed music requiring data decompression

■ compressed speech requiring a subband decoder and an 8-to-24 kHz sample rate converter

■ sound effects requiring a 22.2545-to-24 kHz sample rate converter

Each module must be cached and executed in the proper order to accomplish the desired results. See Figure 3-10 for a diagram of the data flow in this process.
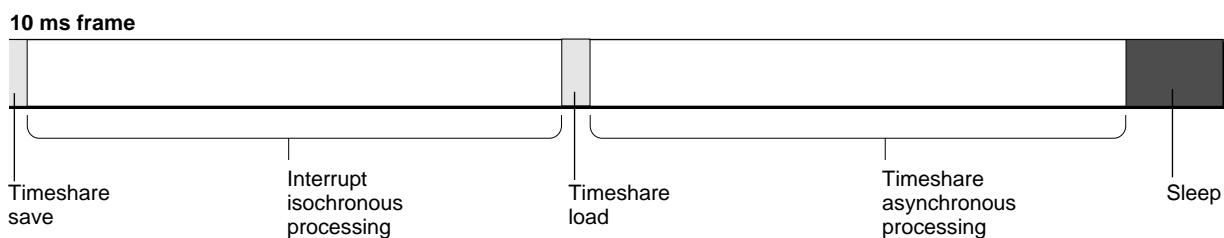
Figure 3-10 Sound player example data flow

Frame Organization

Figure 3-11 shows the processing divisions that occur during a frame. Each frame begins with the frame interrupt. If a timeshare task is running, its context is saved in external memory. Then the list of real-time tasks is parsed and each of the active tasks are executed in sequence. When all real-time tasks are completed, the timeshare processing is resumed. If there was a task being executed when the frame interrupt occurred, it is reloaded; otherwise, the list of timeshare tasks is checked. The next active task is located using a round-robin scheduling algorithm. This selected task is then loaded and executed. Processing continues until the next frame interrupt or until all timeshare tasks are completed or become inactive.

Figure 3-11   Frame-based processing

10 ms frame

| Timeshare save | Interrupt isochronous processing | Timeshare load | Timeshare asynchronous processing | Sleep |
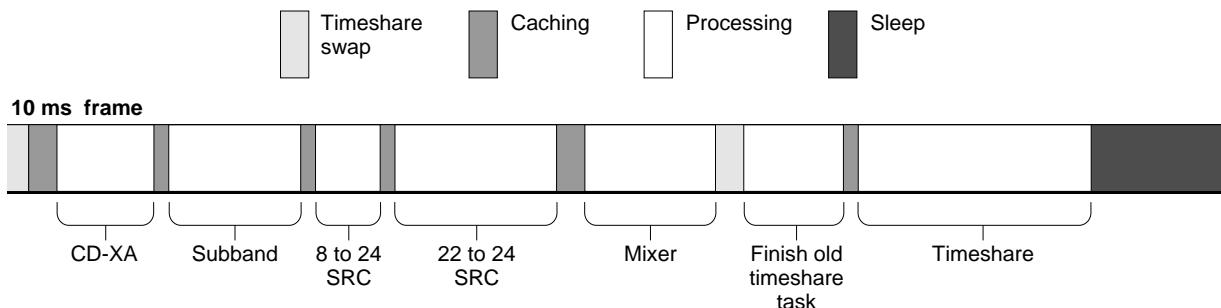
If there are no active timeshare tasks to be done, the DSP goes into **sleep mode** (shuts itself down), using the wait-for-interrupt instruction. The DSP will then be brought back online automatically at the next frame interrupt. This provides automatic power control for portable computers based on the DSP's processing load. If no DSP tasks are active, the Real Time Manager will go even further and shut down all DSP-related circuits, including the timer, serial ports, and other related hardware.

Note
During a frame all real-time tasks are executed once and only once. Timeshare tasks use cooperative multitasking, similar to Macintosh applications, and are executed in sequence until all timeshare tasks become inactive or the end of the frame is reached. ◆

Using the sound player example given earlier, a detailed diagram of a frame is shown in Figure 3-12. This figure is not to scale and shows only general content.
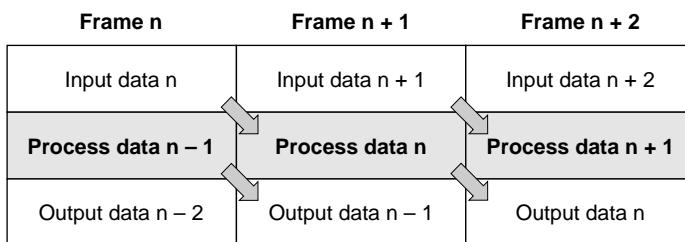
Figure 3-12 Multiple code module processing

As Figure 3-12 shows, the five required real-time processing modules are run in sequence. The timeshare algorithm that was running when the frame started is reloaded and completed. One more timeshare algorithm is run, and since no more timeshare algorithms are active in this example, the DSP goes to sleep and waits for the next frame interrupt.

Frame Size Selection

Frame-based processing requires some latency in the data flow. In particular, the input port must collect a full frame's worth of samples before the DSP can process them. Likewise, the DSP must generate a full frame's worth of samples before the output port can start transmitting them. This requires a latency of two frames between input and output data. Figure 3-13 illustrates this basic concept.

Figure 3-13 Process data flow

There are four factors that influence the selection of the time interval of the frame. They are:

■ *Size of buffer.* This is proportional to the frame time interval. The longer the frame, the more cache memory is needed for each buffer.

■ *Overhead reduction.* This is inversely proportional to the frame time interval. The shorter the frame, the greater percentage of DSP processing time is used in overhead. For example, if the frame represents 240 samples then the overhead is 1/240 of the algorithm on a sample-by-sample basis. Algorithm caching is needed only once for every 240 samples or 0.42% compared to processing a single sample at a time.

■ *Granularity of access.* During a frame the processing sequence cannot be interrupted. Changes in process configurations must happen on frame boundaries.

■ *Input/output latency for important algorithms.* The longer the frame the higher the latency between input to output data streams.

Buffer size and overhead reduction pull in opposite directions. Granularity of access is dependent on the application; sound synthesis with MIDI is probably the most demanding potential application, putting the lower limit at approximately 2 to 4 ms per frame. Input/output latency sets the upper limit on the frame time. The most demanding known algorithm for latency is the V.32 data protocol, which sets an upper limit of 13 ms per frame.

The default frame time for the Macintosh Quadra 840AV and Macintosh Centris 660AV is 10 ms. This is a convenient value for the following reasons:

■ many common sample rates have an integer number of samples in 10 ms

■ the buffers are small enough to have several in the cache at the same time (only 240 samples for 24 kHz)

■ a decimal-based frame time is easier to work with

■ a 10 ms frame time reduces the DSP operating system overhead

The software architecture of the Macintosh Quadra 840AV and Macintosh Centris 660AV is flexible and supports multiple frame rates up to four. The standard alternate frame rate is 5 ms. In the Macintosh Quadra 840AV and Macintosh Centris 660AV implementation, the frame rate can be changed only when no programs are using the DSP.

Visible Caching

The basic assumption for visible caching is that there is not enough high-speed cache to hold all of the code the DSP must execute each frame. This difficulty is overcome without increasing hardware costs by caching each algorithm (module) from external memory into high-speed cache when it is needed. Because most algorithms for the DSP consist of some set-up code and a compact set of loops that take up most of the processing time, this method of visible caching results in only a small fraction of the total main processor bus bandwidth being used by the DSP.

▲ **WARNING**
If you are writing a system extension that uses real-time processing, be aware there is only a limited amount of memory available because the system heap is not expandable. You will need to include a 'zsys' resource in your system extension to enlarge the system heap before the system extensions run. The amount of memory needed may be more than required by your system extension because some of the memory may be used by LocalTalk, EtherTalk, TokenTalk, and A/ROSE. ▲

The visible caching approach works for many signal processing algorithms. The assumption is that only a small processing loop is needed with a small amount of data per frame, resulting in a fairly short caching time overhead. The loop is run many times per sample and takes considerable processing time. For audio and telecommunication

algorithms, the ratio of processing instruction cycles to caching cycles is often in the 40:1 range. Hence, the caching overhead cost in processing power is in the 10 percent range. This is a fairly low impact considering the cost savings from eliminating fast SRAM and its related support circuitry. However, this processing model does not work well for applications where these assumptions do not hold.

The signal processing algorithms, variables, and data tables are all stored in locked contiguous memory blocks (called **sections**) and are loaded into cache memory either automatically or by calls to the DSP operating system's visible caching routines. With this approach the DSP programmer has complete control of the caching process, unlike most hardware caches that are invisible to the programmer and to the executing program.

Code can also be executed directly from external memory. This is useful for small code blocks, such as set-up and control code, or blocks that contain only single instruction loops that are cached automatically on the DSP chip. It also allows very large code blocks to be run by the DSP, although the execution speed will be substantially lower.

Assuming support for DRAM page mode is provided in the hardware, the caching function (block move) is likely to be three times more efficient than single accesses. Single external accesses are used when executing from external memory or when fetching or updating data in external memory. Even for fairly short control and set-up code blocks it is often faster to cache them before execution. The break-even point can be calculated based on the cache speed, single access speed, and block move speed of any given implementation, and is often as low as 25 instructions. For information about DRAM timing, see "Access Timing," in Chapter 2.

Under normal circumstances, the DSP should demand only a low percentage of the CPU bus bandwidth. This allows graphics and other main processor functions to proceed as rapidly as possible. However, there are DSP applications that take a significant amount of the CPU bus time, in which case the main processor runs slower. But since much of the work is being done on the DSP, the total system runs faster than a computer without a DSP.

As explained in "Real-Time Processing Architecture," earlier in this chapter, here are two visible caching execution models that are supported by the DSP operating system: AutoCache and DemandCache. With AutoCache the programmer specifies which code and data blocks are to be loaded and saved. The DSP operating system performs all load and save functions automatically. In DemandCache the programmer explicitly moves code and data blocks on and off-chip whenever needed by making the appropriate calls to the DSP operating system from the module. Both models have advantages and disadvantages.

The AutoCache model provides a simple easy-to-use method of visible caching for small DSP algorithms (for example, sample rate converters, compressors and expanders, filters, and others). Whenever possible, the AutoCache model should be used, for simplicity of operation and programming.

In the DemandCache model, caching is explicitly handled by the DSP programmer. In the simple case, the programmer provides a single main program and one or more cacheable functions. A cacheable function is made up of one or more code blocks. The

main program resides in external memory and calls the DSP operating system to cache functions on-chip and run them. The programmer can thus select functions in any order and can repeat functions as needed, at the cost of increased program size and complexity.
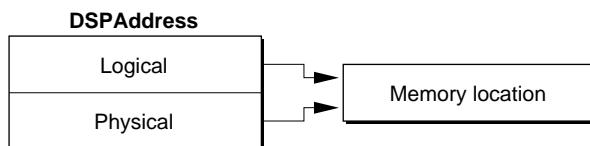
DemandCache is used for algorithms that must select different signal processing functions depending on conditions or commands. A good example of such an algorithm is a multimode modem program. The actual data processing program selected depends on the kind of modem on the other end of the telephone line. The required program would be cached explicitly by the main program.

Another way to build complex functions is by combining multiple simple modules and using the skip function. This is described in "Grouped Modules," later in this chapter.

DSP and Main Processor Addressing

Real-time data processing is designed for systems that include a memory management unit (MMU). However, the DSP3210 does not use an MMU to translate logical addresses to physical addresses. As a result, the main processor uses logical addresses for all of its memory accesses while the DSP uses physical addresses. Addresses that are used by both the Macintosh and DSP operating systems are stored in `DSPAddress` structures that contain both the logical and physical form of the address. A diagram of the structure is shown in Figure 3-14.

Figure 3-14   `DSPAddress` structure

Note
The Real Time Manager is responsible for setting up and maintaining these `DSPAddress` data structures. Since the DSP uses locked–down memory, this approach allows the DSP to operate in a virtual memory (VM) system without actually having an MMU. The local memory addresses are translated from logical to physical form by the Real Time Manager before the DSP chip uses them. ◆

All blocks of memory indicated by a `DSPAddress` data structure are by definition locked contiguous and non-cacheable. They are locked contiguous so that the DSP does not have to worry about scatter/gather operations when using a `DSPAddress` data structure. The blocks are locked non-cacheable to eliminate conflicts that would occur when the DSP modifies a memory location that the main processor had cached.

The `DSPAddress` is a general type. There are also specific types, including `DSPFIFOAddress`, `DSPTaskAddress`, `DSPModuleAddress`, and `DSPSectionAddress`. Each has the same data structure as a `DSPAddress` but points to a specific structure.

Containers

Each memory location that a given section may occupy is called a **container.** For example, if a section can be cached on-chip from an off-chip location it has two containers—one in main memory and one in the DSP's on-chip memory. Containers are fully discussed in "Sections Defined," later in this chapter. The DSP operating system keeps track of the active container by means of data structure called a **section table.**

Primary and Secondary Pointers

Each section has a primary and a secondary pointer. There are two possible values for these pointers, depending on whether the section uses one container or two containers. You must be careful when examining or using these pointers when DSP code is running because in DemandCache the DSP operating system can change the sections from one-container to two-container when caching sections on-chip, and from two-container to one-container when moving sections off-chip. The pointers are summarized in Table 3-1, where X and Y are pointers to sections.

Table 3-1   Primary and secondary pointers

Primary	Secondary	Where applicable
X	nil	One-container section
X	Y	Two-container section
X	X	Not applicable
nil	X	Not applicable
nil	nil	Not applicable

The pointer to the section in the exception vector table is always the same as the primary pointer. This invariant is maintained by the DSP operating system during both AutoCache and DemandCache.

One-Container Sections

Sections that have only one container have a primary address and a `nil` secondary address. The primary address can point either on-chip or off-chip. Whenever the section data is accessed by the DSP, the primary address is used.

Two-Container Sections

Sections that have two containers are slightly more complicated. There are valid addresses in both the primary and secondary pointers. The primary pointer is where the DSP user code will access the section. The secondary pointer is where the DSP operating system will load the section from and save it to. Both the primary and the secondary address may point on-chip or off-chip.

On-Chip and Off-Chip Addressing

Initially the application will want to find out if the addresses discussed in the previous section point to locations that are on-chip or off-chip. The following rules apply:

■ The application can tell if the address points on-chip by looking at the physical and logical components of DSPAddress. If the logical value is nil and the physical value is not nil, the address points to on-chip memory.

■ Pointers to off-chip memory can be recognized because the logical and physical pointers are both not nil.

■ It is not valid to have a logical address without a physical address.

■ If the logical and physical components of DSPAddress are both nil, the pointer is nil.

These rules are summarized in Table 3-2, where X and Y are addresses.

Table 3-2   On-chip and off-chip addresses

Physical address	Logical address	Where located
X	nil	On-chip
X	Y	Off-chip
X	X	Off-chip
nil	X	Not valid
nil	nil	Not valid

Guaranteed Processing Bandwidth

A system of measuring and controlling execution time guarantees that real-time tasks will execute completely every frame. This system is called guaranteed processing bandwidth (GPB).

GPB is measured in processor instruction cycles. For example, with 10 ms frames, 166,666 cycles are available for a 60 ns instruction cycle and 125,000 cycles for an 80 ns instruction cycle. Therefore, if a processor is running 60 ns instruction cycles instead of 80 ns instruction cycles, more instruction cycles are available for a given frame time.

Each code module is assigned a GPB number during development by the DSP programmer. This number is called the *GPB estimate*. It is an estimate because certain portions of the processing time depend on bus latency and other factors that are not the same for different machines or implementations.

When the DSP program tries to install a task in the real-time task list, its estimated GPB requirement is compared with the remaining GPB available (calculated by subtracting the GPB values for real-time tasks already installed from the total available GPB). If there is enough time available, the new real-time task is installed. Otherwise, an error message is sent back to the application attempting to do the installation.
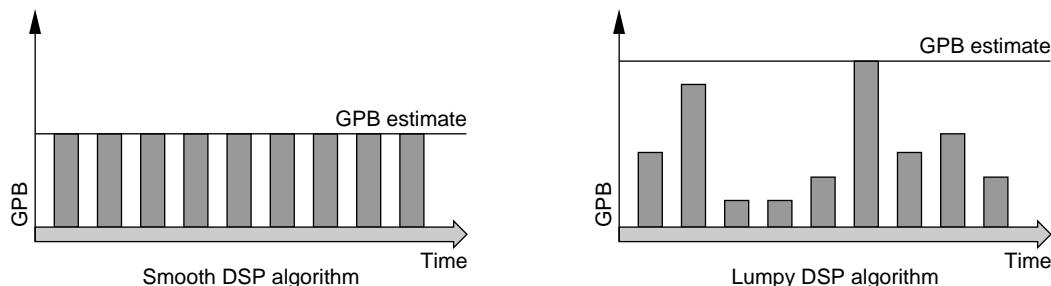
Each time a real-time task runs, the DSP operating system calculates the *GPB actual value* for the task. This actual value is used for future calculations in determining if additional real-time tasks can be installed. Also, this revised GPB actual value can be used to update the modules value in the DSP Prefs file to improve the GPB estimate for the current target machine. In this way, the estimate becomes adapted to faster or slower hardware implementations.

Smooth and Lumpy Algorithms

The simple model described above works well for smooth DSP algorithms. A **smooth algorithm** is one that always takes the same or almost the same time to execute every frame. The "almost" comes from variations outside the control of the algorithm, including I/O time handled by the DSP operating system, and bus overhead, which may vary depending on other bus traffic. There can also be minor variations within the algorithm, but these must be kept to a small percentage if the model is to work correctly.

The other type of DSP algorithm is called a **lumpy algorithm.** In this case, the algorithm uses various levels of processing for each frame This may depend on the data being processed, the status of the function it is controlling, or other variables. A diagram comparing the two types of algorithms is shown in Figure 3-15.

Figure 3-15   Smooth and lumpy DSP algorithms

As you can see from the diagram, the GPB estimate for the smooth algorithm is also the GPB actually used on a regular basis. On the other hand, the GPB estimate for the lumpy algorithm must indicate the maximum level of processing required. To guarantee DSP processing availability, the maximum level of processing must always be used in GPB calculations. Thus there is often additional timeshare processing available when a lumpy algorithm is running. The DSP programmer must indicate whether each module is using a smooth or lumpy algorithm.

Calculating GPB

For real-time algorithms, the actual GPB is recalculated by the DSP operating system every frame. If the new GPB actual value is larger than the stored GPB actual value from previous frames, the new value is stored. This is called the *peak detection* algorithm. It is designed to maintain the actual maximum GPB used, including any bus or I/O variations. The GPB actual value starts off at zero when the real-time task is installed.

When the Real Time Manager wants to determine if there is enough processing time still available to install a new task, it uses a simple algorithm to decide which of the two available values, the GPB estimate or the GPB actual value, it should use for each module in its calculations. This selection is based on the state of the `UseActualGPB` flag in each module header.

For smooth algorithms, this flag is always set. The selection algorithm is this: if the actual value is non-zero and the flag is set, use the GPB actual value as the current value; otherwise, use the GPB estimate. This algorithm is designed to give the most accurate accounting of the available GPB at any given time. However, the estimated value is used until the module has a chance to run at least once. After that, the actual value is used, whether it is smaller or larger than the estimate. This is how the GPB system automatically adapts to different CPU configurations.

GPB for Lumpy Algorithms

The simple approach to GPB used for smooth algorithms does not work for lumpy algorithms. A somewhat different approach is required for this case. First, it is necessary to separate lumpy algorithms into two different classes: **smart lumpy algorithms** and **dumb lumpy algorithms.** A smart lumpy algorithm determines cases when it is executing code that will result in maximum utilization of GPB. A dumb lumpy algorithm cannot determine when this may be the case.

An example of a smart lumpy algorithm is a multirate modem. There are various stages of the modem, including initialization, setup, and data transfer. The maximum GPB use is usually taken by one of the steady-state data processing programs. When this algorithm is reached, the DSP program calls the `GPBSetUseActual` routine.

An example of a dumb lumpy algorithm is a Huffman decoder. This decoder takes longer to decode some bit streams than others, and there is no way to tell beforehand how long it will take. In fact, the processing time can grow without limit in the case of random noise input.

Two different mechanisms handle these two cases. For smart lumpy algorithms, the DSP program knows where the maximum GPB usage is in the code, and so is required to set the `UseActualGPB` flag with the `GPBSetUseActual` routine. The DSP operating system does not actually set the flag until the GPB calculations for this module are completed. This forces the Real Time Manager to continue using the estimated value until after the peak use frame has occurred. After that, the actual value correctly reflects the processing needed by this module on this hardware configuration. The DSP operating system continues to use the peak detection algorithm for computing the actual value, so future peaks may slightly increase the actual value because of variations in I/O and bus utilization.

For dumb lumpy algorithms, the DSP program can check on the available processing time left in the real-time frame, and shut down the process if an overrun is about to happen or has already happened.

There are two macro calls to the DSP operating system that support the dumb lumpy algorithm. The `GPBExpectedCycles` macro returns the expected processing time; the `GPBElapsedCycles` macro returns the amount of processing time used so far. If

the amount used so far is getting close to the expected time, the module must execute its processing termination procedure. This procedure should end the processing in whatever manner is appropriate for this algorithm. If the processing duration has exceeded the time, the `UseActualGPB` flag should be set, and the processing termination procedure should be followed.

If the dumb lumpy algorithm exceeds its GPB estimate, it may cause a frame overrun. If this happens, the offending real-time task that includes this module is set inactive by the DSP operating system, and the application is notified by an interrupt. This process is described in "Frame Overruns," later in this chapter.

Dumb lumpy algorithms are tricky to program correctly. If at all possible, such algorithms should not be done in real time, but in timeshare, where length of execution is not a vital factor.

Fast Execution Versus Real-Time Execution

A task executes faster as a real-time task than as a timeshare task only if the real-time task list is using most of the processing bandwidth of the DSP. In many cases, running in the timeshare list will yield more processing time. By carefully analyzing what applications need real-time processing and what need "run as fast as you can go" processing, you can decide which tasks should go into the timeshare list. Candidates for timesharing normally include tasks such as lossless compression of disk files, graphics animation, and video decompression. All such tasks should use as much DSP bandwidth as possible, because the more they run the sooner they finish. Such tasks must not be confused with real-time tasks, which require a specific amount of data be processed during a specific time period.

Processor Allocation for Timeshare Tasks

Timeshare processing is considerably different than real-time processing. A timeshare task often has no way to determine how much processing time it will have in a given frame. It is even possible to load the real-time task list so that no timeshare task execution is possible. Bear in mind that it takes a significant amount of processing time to load and unload a timeshare task. If there is not sufficient time to perform both operations the task will not execute during that frame.

Two numbers can help an application determine if it is worth installing a timeshare task in a given DSP task list. The two numbers are

■ **Average timeshare available (ATA).** This is effectively the average sleep time that the DSP is getting per frame. It represents actual unused DSP processing, averaged over several frames.

■ **Average timeshare used (ATU).** This number is effectively the average amount of timeshare being consumed by timeshare tasks that are already installed.
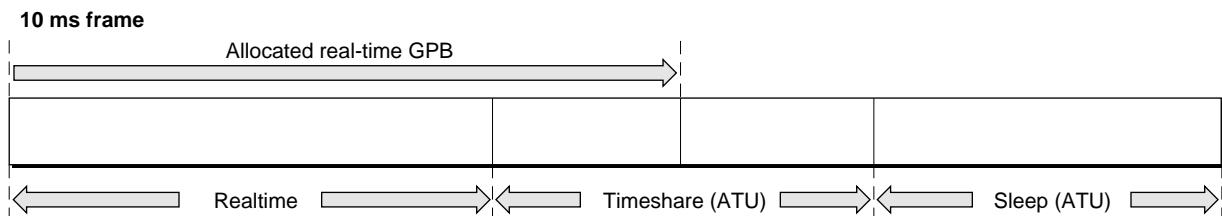
Adding the two numbers above yields the **average total timeshare (ATT)** available. Figure 3-16 diagrams this concept.

Note
The application calculates the ATA and ATU using the maximum
number of cycles for the processor, the number of real-time cycles
allocated, the number of real-time cycles used during the last frame,
and the number of timeshare cycles used during the last frame. ◆

Figure 3-16   Timeshare capacity figures

As shown in Figure 3-16, the ATT value is not necessarily the difference between the
frame processing capacity and the GPB allocated to real-time tasks. It is often the case
that real-time tasks are inactive, or not running at full processing bandwidth. This makes
additional timeshare processing available.

The averaging process is used to calculate the timeshare processing numbers because
they will usually fluctuate with time. The numbers are provided to allow an application
to determine if installing a timeshare task is effective at any given time.

Once a timeshare task is installed, it is recommended that the application check the value
of the ATT every so often to make sure that it is still getting service from its timeshare
task. Alternatively, the timeshare task itself can report to the application on its activity
level. The Real Time Manager does not warn the application when timeshare tasks are
not being executed.

Frame Overruns

When several tasks have been installed on the DSP, or if one large and lumpy task is
installed, and if the estimated GPB requirements are not accurate, it is possible for the
DSP to still be processing data when the next frame interrupt is received. This results in a
frame overrun. There are three categories of frame overrun:

■ Category one: the DSP acknowledges the current frame interrupt after the next
 interrupt is received, but before a second interrupt. The DSP misses only one interrupt.

■ Category two: the DSP acknowledges the frame interrupt after two interrupts have
 been received, but before a third interrupt. The DSP misses two interrupts.

■ Category three: the DSP has not acknowledged the frame interrupt for five successive
 interrupts. The DSP misses five or more interrupts.

Category One Frame Overrun

The DSP operating system detects a frame overrun if the interrupt line is asserted before it has been acknowledged. When a category one frame overrun occurs the DSP operating system attempts to recover during the next frame. Since the DSP operating system cannot tell if one or more frames has passed it assumes only one frame has been skipped. To recover, the DSP operating system checks all modules for their current GPB usage, the task with the module having the worst overage is set inactive, and the application is notified. All other clients (such as toolbox routines) are notified that the DSP has skipped a frame.

When an application receives a task inactive message from the Real Time Manager it should deallocate the offending module. This will update the DSP Prefs file with the correct GPB value for that module. The application can then reallocate the module and attempt to reinstall the task. When an application receives a skipped frame message it can do anything from ignoring it to removing and reinstalling the task.

Category Two Frame Overrun

Since the DSP cannot determine how many frames have passed, the external interrupt logic must detect a category two frame overrun. To recover, the interrupt logic sends a hardware interrupt to the main processor and the Real Time Manager executes its DSP overrun recovery code. The Real Time Manager checks with the DSP operating system for the offending module and sets it inactive. If the DSP operating system cannot identify the worst-case module then the Real Time Manager will determine which module is at fault. The Real Time Manager then issues the DSP a reset command and the application that installed the offending module is notified that the task is inactive. All other clients (such as Toolbox routines) are notified that the DSP has been restarted.

The application that receives the task inactive message should respond in the same way as for a category one overrun. When an application receives the DSP restart message it should check the task's memory for possibly corrupted data or code. The recommended response is to remove, rebuild, and reinstall the task.

Category Three Frame Overrun

In the event that the DSP does not respond to interrupts by the sixth frame, the frame overrun logic will issue a hardware reset to the DSP and I/O subsystems. In this case it is assumed that both the DSP and the main processor have crashed. It is important that the DSP and I/O subsystems be reset to prevent possible problems in the output subsystems—for example, a fixed sound on the speaker or the telecom system left offhook.

Recovery from a category three frame overrun is impossible. All clients, including application and Toolbox routines, must start over and install their tasks from the beginning.

Data Structures

As explained in "Real-Time Processing Architecture," earlier in this chapter, it is important to distinguish DSP modules from DSP tasks:
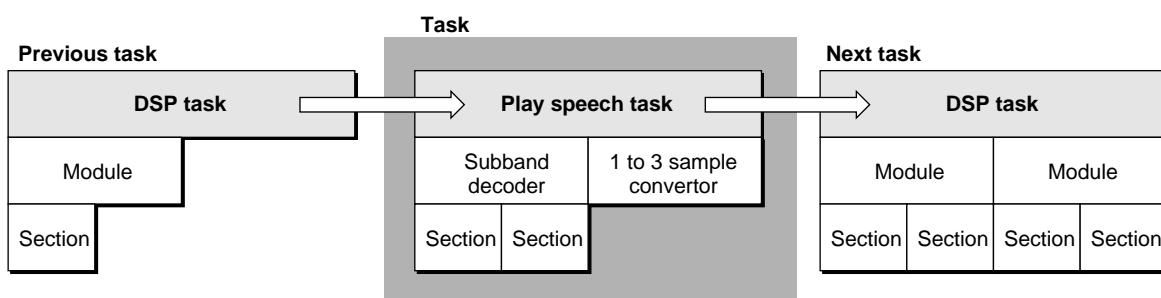
- DSP *modules* are the basic building blocks of digital signal processing software. They always include DSP code. They also usually include some data, input and output buffers, and parameter blocks. There are an infinite number of combinations possible, depending on the desired function.

- A DSP *task* is made up of one or more DSP modules. The purpose of this grouping is to place together, in the appropriate order and with the appropriate I/O buffer connections, all of the DSP modules needed to complete a particular job. A DSP task will frequently contain only one DSP module.

The DSP module is *provided to* the Macintosh program as a resource, and is loaded into a DSP task using the Real Time Manager. A DSP task is *constructed by* the Macintosh application using a series of calls to the Real Time Manager. These calls create the task structure, load and connect modules in the desired arrangement, allocate the required memory, and install the completed task into the DSP task list. The reason for combining modules into tasks is to ensure that the combined function is always executed as a set.

A good example of a task is one that plays compressed speech that was recorded via the telecom subsystem. The data is recorded via the subband decoder at 8 kHz sample rate and compressed before being stored on a disk drive. To play the data over the speaker, it must be decompressed back to 8 kHz samples, and then the sample rate must be converted to 24 kHz data to match the sample rate of the speaker system. A diagram of this example is shown in Figure 3-17.

This task is executed by following the chain of modules from left to right. The task is activated or deactivated as a single unit. It is also installed and removed from the DSP task list as a unit.

Figure 3-17   Task with two modules
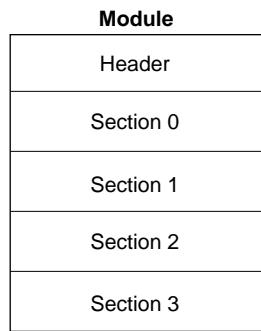
Sections Defined

The internal structure of the DSP module is compartmentalized into code and data blocks. It is this design of the DSP module that gives the real-time data processing architecture its real power and flexibility. Each module is made up of a header and one or more *sections*, as shown in Figure 3-18.
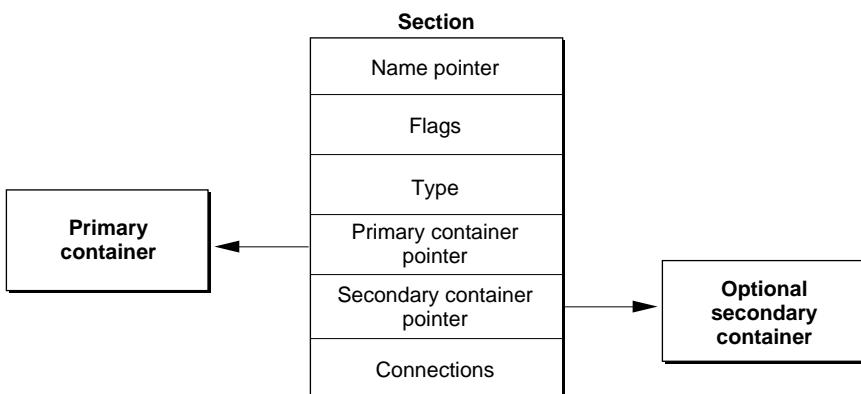
Figure 3-18   The module data structure

Module

Header
Section 0
Section 1
Section 2
Section 3

The header contains information about the entire module, such as its name, GPB information, and control flags. Also included in the header is a count of the number of sections in the module. This allows the module data structure to be of variable length.

Each section also has a name, flags, and data-type fields. In addition, each section has pointers to two containers. It is the containers that actually hold the data or code for the section. The sections are the building blocks of the module. A section can point to code, data tables, variables, buffers, parameters, work space, or any other resource needed to provide the desired function. The only requirement is that the first section must always point to code. A simplified diagram of a section is shown in Figure 3-19.

Figure 3-19   The section data structure

Section

Name pointer
Flags
Type
Primary container pointer
Secondary container pointer
Connections

Primary container

Optional secondary container

Note
The section does not contain the actual code or data used by the DSP chip. Rather, it is a data structure that contains pointers to the code or data block. The DSP operating system uses the section structure and flags to cache the actual code or data block as required. ◆

The Section Control Flags and Data Types are used to control caching and manage buffers. The connection data is also used for buffer management internally to the Real Time Manager. These operations are discussed in "Buffer Connections Between Modules," later in this chapter.

The two containers are called the *primary container* and the *secondary container*. A primary container is always required. The secondary container is optional. The primary container is usually allocated in the cache, but can also be in local memory. The secondary container is usually allocated in local memory, but in special cases can be allocated in the cache. Allocated memory for the containers must be in either local or cache memory.

The visible caching system moves data from the secondary container to the primary container, which is usually moving the contents from local memory to cache memory. This is called a **cache load.** The visible caching system also moves data from the primary container to the secondary container, which is usually moving the contents from cache memory to local memory. This is called a **cache save.**

In cases where no caching is required, only one container is needed. The primary container in this case is located in local memory if it contains fixed data or parameters for communication between the main processor application and the module, or in cache memory if it is simply work space.

The section concept was developed to facilitate creating modules with generic functions that can be used in many different applications. It also forms the basis of the plug-and-play module architecture, where input and output data streams can be interconnected between off-the-shelf modules to create new functions. In addition, it supports several different execution models and is easily adapted to future hardware advances, such as significantly larger cache memories and hardware instruction caches.

AutoCache

With the AutoCache caching model (discussed in "Visible Caching," earlier in this chapter), the section data is moved from the secondary to the primary container, before the module runs, if the Load flag is set. Likewise, the section data is moved from the primary to the secondary location after the module runs if the Save flag is set. During execution of an AutoCache module, the primary and secondary pointers never change.

DemandCache

With the DemandCache caching model, two container sections are used much the same way as they are used in AutoCache. The only difference occurs when a section is pushed or popped.

When a section is pushed it changes from a one-container to a two-container section. Data is moved from the secondary to the primary location if the Load flag is set. When a section is popped it changes from a two-container to a one-container section. Again, data is moved from the primary to the secondary container if the Save flag is set.
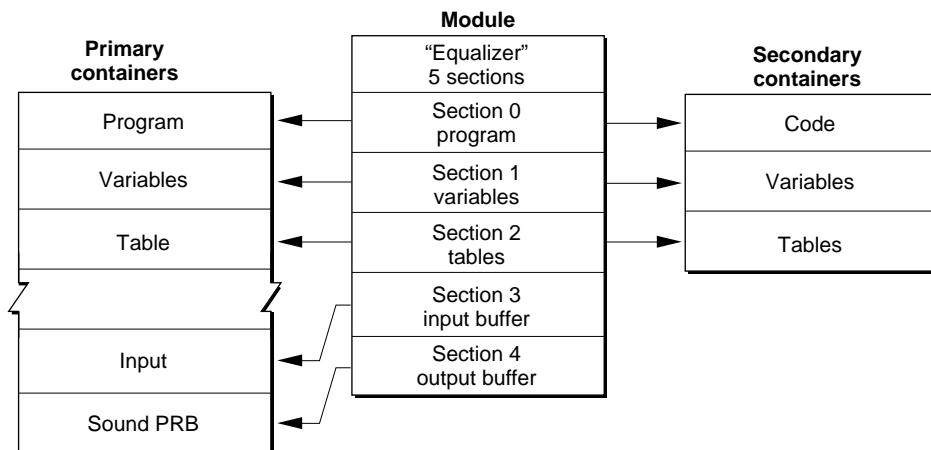
Sections and Caching

The actual operation, with either AutoCache or DemandCache, loads code or data by section into the cache prior to its use, and then saves data back from the cache when completed. The section data structure contains flags, pointers, and other information to support these functions.

For every section there are two possible containers (buffers): the primary container and the secondary container. The caching function moves data between the secondary and primary containers prior to module execution, and moves data between the primary and secondary containers after module execution. Only the minimum required moves are made. For example, it is only necessary to move code into the cache from the secondary container. It is not necessary to move it back, assuming the code is not self-modifying.

A diagram of a sample AutoCache module, including its primary and secondary containers, is shown in Figure 3-20. This example shows five sections in the module: the program (code) section, state variables, a data table, an input buffer, and an output buffer. The first three sections have two containers each, while the last two have only a primary container.

Figure 3-20   Dual-container AutoCache example

In the example, the code, variables, and table sections are loaded into the cache before the code section is executed. After execution completes, only the variables are saved back to local memory. It is important to recognize that the input and output buffers are not moved, but exist in the cache. This buffer mechanism is described in "Buffer Connections Between Modules" and "Buffer Connections Between Tasks," later in this chapter.

Note
This discussion of the caching system is primarily applicable to
AutoCache. More detailed information about AutoCache and
DemandCache, including the differences between them, is presented in
"Execution Models," later in this chapter. ◆

Container Memory Allocation

The structure of modules and sections requires several different blocks of memory. The
example shown in Figure 3-20 uses nine different blocks: the module itself, five primary
containers, and three secondary containers. The module and the secondary containers
are in local RAM, and the primary containers are in the cache.

Substantial memory management and allocation effort is needed to support this type of
data structure. Fortunately for the programmer, the work is done automatically by the
DSP operating system. The allocation and memory management is done in two phases.
When the client loads the module into memory from a resource file, the Real Time
Manager allocates all the required blocks in local memory to hold the structure. In the
example shown in Figure 3-20, the allocation includes the module itself and three
secondary containers. The containers are then loaded with data from the resource file.
This completes the first phase of memory allocation.

The application must also specify the I/O connections for the module, a process covered
in "Buffer Connections Between Modules," later in this chapter. Once all of this is done,
the Real Time Manager calls one of its routines to take care of cache allocation; this is the
second phase of allocation. The task is now ready to install. For DemandCache,
additional allocation is performed by the DSP operating system at run time.

There are many factors that the Real Time Manager must take into consideration when
placing section containers in the cache. First, it must be aware of any reserved memory
in the cache. This includes areas for the DSP operating system as well as buffers. Next, it
must be aware of the bank configuration of the cache. For some DSP implementations, it
is important to locate different sections in different banks to ensure highest performance
operation. This is not true for the DSP3210, but it was for the DSP32C and will be true for
future versions of the DSP3200 family.

It is important to properly mark the sections for bank preference to ensure correct
placement for all future DSP3200 processors. This takes the form of Bank A and Bank B
preference flags. If both are set, this indicates that any bank will do. If neither are set, it
indicates the section should be located outside of the cache. In the example above, the
program, variables, and table sections (primary containers) are located in Bank A. The
I/O sections (primary containers) are located in Bank B. The architectural concept
behind this bank organization is explained in the AT&T DSP3210 manual.

Other allocation decisions are related to the connections between module I/O buffers.
The Real Time Manager attempts to arrange the sections in the cache in such a way as to
eliminate as much buffer movement as possible. If a buffer can be set and left in one
place without being moved between modules or tasks, it reduces the overhead for
maintaining the buffer.

A Complete Software Example

Figure 3-21 diagrams a typical structure of digital signal processing software with sections, modules, and tasks. It shows a dual task list (real-time and timeshare) and adds multiple DSPs and DSP client controls.

Figure 3-21 shows two DSP devices (two separate DSP subsystems), where the structure detail is shown only for the first device. The first device might be a DSP located on the main logic board and the second device might be a DSP located on either a PDS or NuBus card. In machines not having a DSP on the main logic board both DSPs would be located on accessory cards.

For each device, there can be a number of clients. A DSP **client** is either a system Toolbox routine or an application that wishes to use a DSP. An application cannot use a DSP without first signing in as a client. The client must sign in to each device that it intends to use.
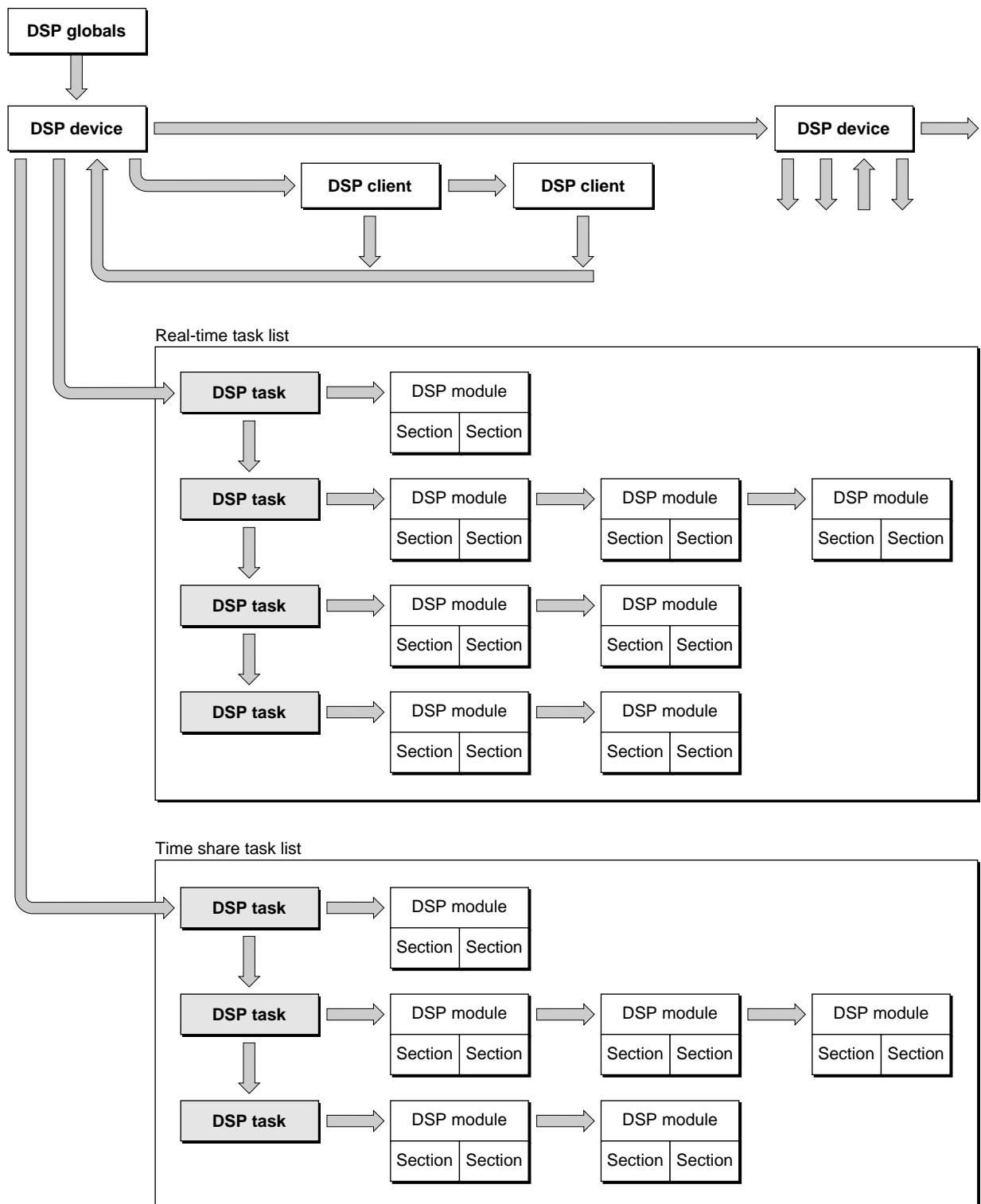
Each device has two task lists. The primary one is for real-time task execution; it is executed once and only once in each frame. The Real Time Manager ensures that the clients do not install too much work in this list, so that the entire list can always be executed by the end of the frame.

The second list is the timeshare task list. It is executed using any time left over in each frame after all real-time tasks have been run. The DSP operating system will repeatedly execute timeshare tasks until it either runs out of time (the next frame begins) or until it makes it through the list once without finding anything to do. If the DSP operating system does not find an active task prior to the frame ending, the DSP is put into sleep mode until the start of the next frame.

Data Buffering

In digital signal processing it is often desirable to connect input and output buffers from several different algorithms, using signal flow techniques. There are routines in the Real Time Manager that accomplish this. The programmer needs to specify the number and format of these buffers (for example, input or output buffers, 32-bit floating-point format, other formats). The buffers can be connected at run time to similar buffers in other, separately designed, algorithms. The application makes calls to the Real Time Manager to specify which connections are desired. The Real Time Manager must attempt to connect these buffers in an efficient manner to minimize the loss of DSP time used in moving buffers around.

Introduction to Real-Time Data Processing

Figure 3-21   Data structure overview

FIFO Buffers

First-in, first-out (FIFO) buffers are used to buffer data between processors or processes. Essentially a FIFO is an asynchronous buffer. In the sound player example (see "Software Architecture," earlier in this chapter), FIFOs are used as buffers between the main processor application and the DSP system for music, speech, and sound-effect data. Likewise, a FIFO is used between the DSP and the speaker I/O port, as shown in Figure 3-22.

Figure 3-22 Example of FIFO buffers
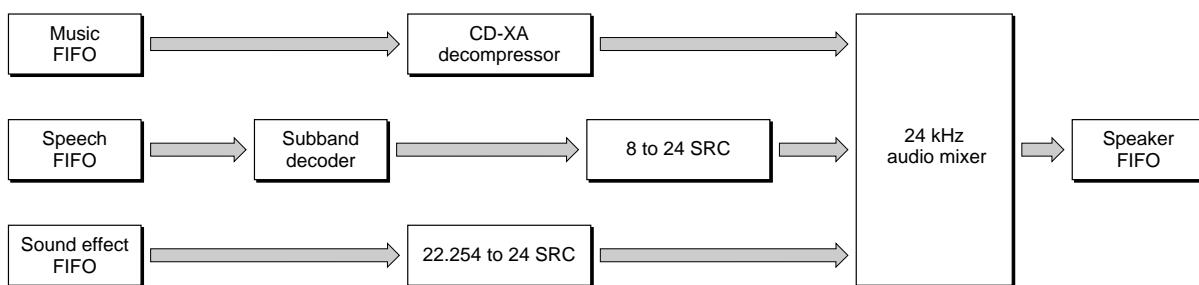
Figure 3-22 shows how FIFOs can be used in a typical application. The speaker FIFO is required because the DSP must keep one frame ahead of the audio serial DMA port. The data FIFOs are necessary because of the slow response time of the disk drive and main processor application. Typically, a buffer in the range of 20 KB to 40 KB is used to buffer the disk to the DSP, depending on the data rate. The disk fills the buffer, and the DSP removes a block every frame. When the FIFO is half empty, the DSP operating system, which handles the FIFO for the DSP module, sends a message to the main processor application. This message tells the main processor application to refill the FIFO from the disk.

FIFOs are also used to buffer output from the DSP to a main processor application—in sound recorders, for example. They work exactly like the FIFOs described above, except in the opposite direction.

Another use for FIFOs is to handle data that is not synchronized to the frame rate. For example, if data is produced at a rate of 22,254.54 samples per second, the amount of data per frame is either 222 or 223 samples (at 100 frames per second). Using a FIFO allows the processes that are filling and emptying the buffer to read or write exactly the amount of data they need. One prime characteristic of any FIFO is its status. It can be empty or full, half empty or half full, or it can be overrun (following an attempt to put more data into the FIFO than it can contain). An overrun happens if the data consumer cannot keep up with the data producer. It is important to make the FIFO large enough to prevent this from occurring or provide a mechanism in the application to halt data production.
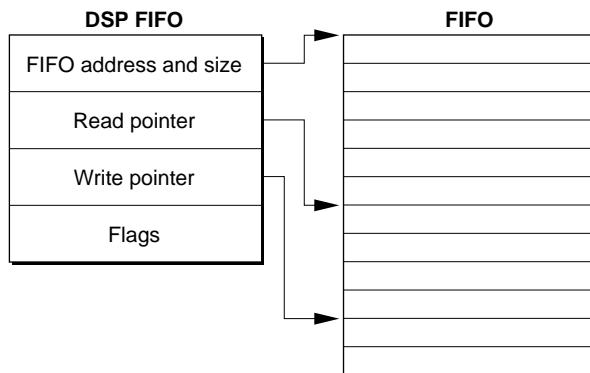
Note
Large FIFOs are usually placed in main memory, because
local memory is limited and the data rate is usually small.
Small FIFOs can be located in DSP local memory. ◆

FIFOs can also be underrun. This happens when the data receiver is not able to read as
much data from the FIFO as it needs to produce one frame's worth of data. The FIFO
routines help by automatically doing a zero fill of the unused buffer. For sound, either in
DSP floating-point format, 8-bit integer packed format, or 16-bit integer packed format, a
zero fill is equivalent to silence. For those functions that require it, the actual amount of
data retrieved is reported.

FIFOs are accessed by making `DSPFIFORead` and `DSPFIFOWrite` calls to the DSP
operating system. The DSP operating system is responsible for handling status
conditions, such as empty or full, half-empty or half-full, and overrun or underrun. The
DSP operating system is also responsible for updating the FIFO pointers, and sending
messages to the client as required. Typical messages include FIFO Empty (DSP is reading
from the FIFO) and FIFO Full (DSP is writing to the FIFO). In order for the DSP
operating system to manage this, the FIFO has a header block called `DSPFIFO`. This data
structure is shown in Figure 3-23.

Figure 3-23   The FIFO and its data header

Each FIFO requires two separate blocks of memory: the `DSPFIFO` structure located in
local memory and the FIFO itself located in either local memory or main memory.
Usually, large FIFOs are placed in main memory by the client, to conserve the limited
local memory space.

You must write to a FIFO to add data to it and you must read from a FIFO to look
at the data in it. Hence you need two separate move operations for each datum: a
`DSPFIFOWrite` and a `DSPFIFORead`. Usually, two different processors or processes
are responsible for the two operations. For example, the application playing the
sound writes to the FIFO, while the sound player task reads from the FIFO.

Real-time data processing FIFOs can read from or write to the DSP side only in longwords. This restriction is necessary because of the real-time cost of reading bytes and reordering them. However, the Real Time Manager supports byte reads and writes to FIFOs from the main processor side. It is also important to note that the DSP operating system masks the lower two bits of the main processor write pointer (for DSP FIFO reads) before using the value to determine the amount of data available in the FIFO. Thus, if the main processor writes six bytes to the FIFO, the DSP will process only four of them. If the main processor writes another three bytes, the DSP will process four more bytes, and so on. This forces all FIFO read/write operations from the DSP to use longwords.
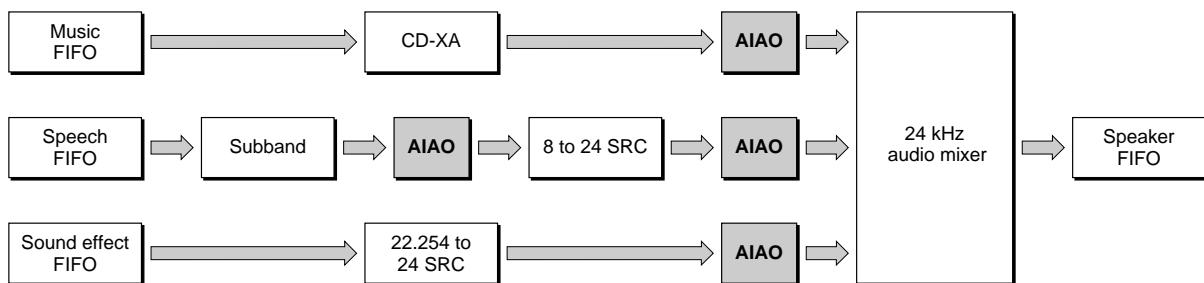
While the FIFO algorithm is ideal for many buffering operations, it requires the DSP operating system to manage the `DSPFIFO` structure and its flags and pointers and also requires dual data movements. These operations make it inefficient for many common buffering operations. It was this realization that resulted in the creation of a new type of buffer, called an AIAO buffer, described in the next section.

AIAO Buffers

AIAO stands for *all-in/all-out,* a naming convention derived from FIFO. AIAO buffers transfer data from one module to another during a given frame. The buffer is transient and acts like a data bucket between modules; the first module fills the buffer, the following module empties it. Another way of thinking about an AIAO is as a frame-synchronous buffer containing only one frame of data.

To understand the need for AIAO buffers, consider the sound player example shown in Figure 3-10. The expanded diagram in Figure 3-24 shows the addition of the AIAO blocks.

Figure 3-24   Code module data flow with AIAOs

AIAO buffers are structured differently from FIFO buffers; they do not contain the header block found in the FIFO. They have a standard section structure and can be used in place, without being moved. It is these characteristics that make the AIAO buffer so efficient.
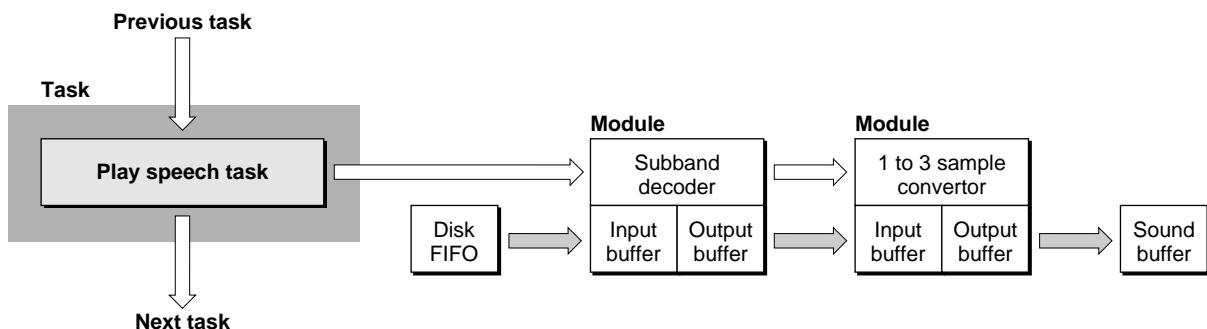
Each module in the example has a fixed number of input and output data streams. These data streams are connected to one another through AIAO buffers. These connections are made between modules by appropriate calls to the Real Time Manager when the task is built. In effect the modules are wired together in a processing network. This plug-and-play module architecture is an important feature of Macintosh real-time data processing.

AIAO buffers can be used to pass data between modules within a task, or to pass data between tasks. The latter case is called an intertask buffer (ITB). The buffers are handled in a similar way, but additional calls are necessary to support them. ITBs require setting aside DSP on-chip memory to pass the data between tasks. See "Buffer Connections Between Tasks," later in this chapter, for more information about ITBs.

Buffer Connections Between Modules

As previously described, AIAO buffers allow very efficient buffering between modules and tasks. To illustrate how they are used in this context, the speech player task shown in Figure 3-17 is expanded in Figure 3-25.

Figure 3-25   Connections between modules

Notice that there are two types of connections in the diagram. First is the control flow, indicated with solid arrows. These are the connections that the DSP operating system follows to execute the real-time task list. The second type is the data flow connection, indicated by shaded arrows. In this example, data flows from the disk FIFO to the subband decoder module, from the decoder to the sample rate converter module, and from the converter module to the sound buffer.

If FIFOs were used for all of these transactions, a total of six DSPFIFORead and DSPFIFOWrite calls would be needed, not including the FIFO I/O needed to get data into the disk buffer from the disk or to get the sound data into the sound output DMA channel. These six data moves would be:

■ read the disk FIFO to cache (10 samples)

■ write the decoded data to a connection FIFO (80 samples)

■ read the decoded data from the connection FIFO to cache (80 samples)

- write converted data to the sound buffer FIFO (240 samples)

- read the sound buffer FIFO to cache for mixing (240 samples)

- write the mixed sound to the speaker FIFO (240 samples)

The output data from the decoder would be collected in the cache and then written to the FIFO as a single block, for efficiency. This is because the FIFO would usually be in local memory, which would take advantage of the multiple byte move hardware mechanism described in "Access Timing," in Chapter 2. Also, there is considerable overhead for each FIFO DSP operating system call, so it would be time-consuming to make a call for each data point.

Likewise, the data would have to be read from the FIFO into a cache buffer where it could be directly accessed by the conversion algorithm. Once the conversion process was completed, the results would have to be written to the sound buffer FIFO. This process would require a separate stage mixer for the sound buffer. Each sound player would have to feed to a separate sound buffer FIFO, which in turn would have to be read by a mixer and summed, then written to the speaker FIFO. This FIFO would have to be in local memory, since an unknown number of channels of sound must be mixed. The cache is too small to maintain very many buffers and leave enough room for the code and data for the module.

All of this data movement would eat up enormous amounts of DSP processing bandwidth. This would be especially true since many of the FIFOs would be located outside the high-speed cache. Using FIFO buffers would reduce the processing capacity by at least half, due to the relatively slow external memory access.

The enormous FIFO overhead just described is eliminated by using AIAO buffers. Using AIAO buffers in this example, the software needs move the data only once, resulting in a sixfold reduction in buffer overhead. The single move is the initial `DSPFIFORead` by the subband decoder module. Only 10 samples are moved between external memory and the cache with AIAOs, but 890 samples must be moved without AIAOs. This represents an 89 to 1 improvement in bus bandwidth utilization.

This model assumes that the output data from the decoder module is left in the cache, and becomes the input data for the rate converter module. The converter samples the data, summing the results to the on-chip sound buffer.

Note

This interconnection of on-chip buffers is accomplished by the Real Time Manager using the `DSPConnectSections` routine. In effect, this routine sets the section pointers for both AIAOs to the same primary container in cache memory, which flags the DSP Manager to reserve the memory space and not to move the data off-chip between modules. ◆

Notice two important points in the example just given:

- First, the input data is directly available to the sample rate conversion algorithm. The converter can access the data as many times as it needs in the high speed cache without a memory-access speed penalty. Multiple accesses are often required for filtering algorithms, such as rate converters.

■ Second, it uses a partial result buffer (PRB) for the sound buffer. A PRB is a buffer that contains a partial mix, and must be summed into, thereby adding to the mix. This is different than a complete result buffer (CRB), which is written into rather than summed into.

The application can directly access the on-chip PRB sound output buffer in the high speed cache. It can read, add to the signal, and write the new results with very little processing impact. This eliminates the entire mixer chain; the mixing function becomes part of the output architecture for the module.

Note

This particular use of AIAOs is specifically for sound functions. It is not a general requirement that AIAO buffers have summing outputs, nor that they have any particular data format. However, if an AIAO is to be connected to the AIAO of another module, the size and data type must match. The specifics of the sound architecture for Macintosh real-time data processing are covered in "Standard Sound Task List," later in this chapter. ◆

The method of operation just described depends heavily on connections being made between the I/O sections of modules. The connection process is handled by the Real Time Manager under the direction of the client. The client specifies what connections are to be made, using calls to the Real Time Manager. Once all connections are completed, Real Time Manager allocation routines place each section in the cache in an appropriate location that ensures the best use of available resources. Every attempt is made to avoid moving buffers. If this is not possible, an attempt is made to move them within the cache between module execution. If all else fails, buffers are temporarily moved into local memory.
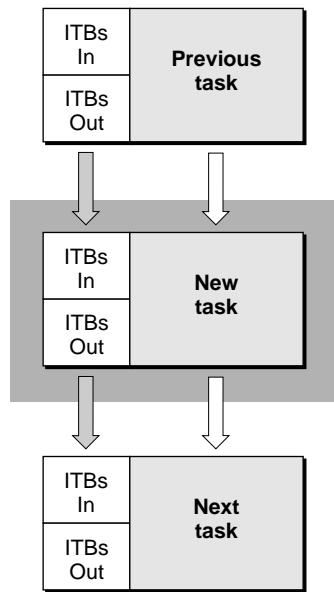
It is this connection mechanism that makes it possible to create generic modules that can be connected in various configurations, depending on the function desired. It is possible to create completely new functions by connecting together existing modules in novel ways.

Buffer Connections Between Tasks

In the speech player task example described in the previous section, the sound buffer is an intertask buffer (ITB). In operation, the AIAO buffer connection between the two modules is an **intermodule buffer.** The Real Time Manager also allows similar connections between tasks, using ITBs.

When the Real Time Manager installs a task, it automatically connects the task to any existing ITBs. The outgoing ITB list from the previous task becomes the incoming ITB list for the new task. Likewise, the incoming ITB list from the next task now becomes the outgoing ITB list for the new task. A diagram of this arrangement is shown in Figure 3-26.

Figure 3-26 ITB connections for previous and next tasks
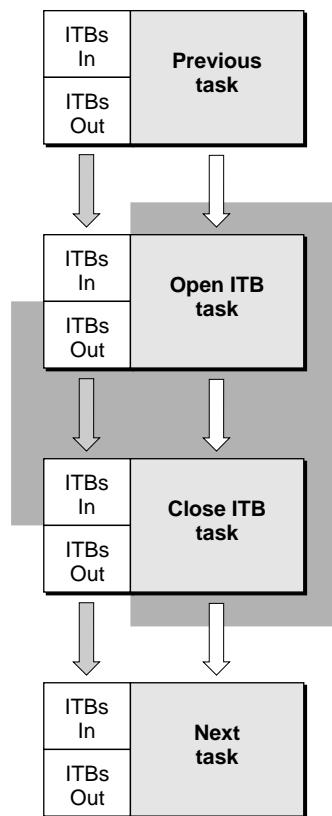
Notice that the diagram above does not represent the data structures associated with the task list, but rather the concept of ITB handling. As shown, each task can import ITBs from previous tasks, and can export ITBs to the next task. Usually, the import and export lists are identical. For sound tasks installed at the appropriate location in the task list, the ITBs are used for stereo PRBs. If a task is installed at the end or beginning of the task list, there will be no ITBs—the ITB pointers will be `nil`.

When the Real Time Manager is asked to create a new ITB, it adds the new ITB to the ITB out list. There is then a mismatch between the output list of the new task and the input list of the next task, because the new ITB was installed prior to the creation of the next task. This new task thus *opens* a new ITB.

The next task the application should install is the *close* task, directly after the first new task. Using the same approach for creating the ITB lists, this task will have the same input list as the previously installed task's output list, and should have the original list as its output list. This concept is illustrated in Figure 3-27.

In Figure 3-27, any additional tasks installed between the open ITB and close ITB tasks will have the additional ITB in both its input and output ITB list. This ITB can be used to pass data between the tasks without requiring off-chip memory. Multiple ITBs can be added in this same way.

Figure 3-27   ITB open and close task configuration

Note

Intertask buffers should be created by the first task installed by an
application and should always be deleted by the last task installed. An
application installing tasks does not have any information about existing
ITBs and cannot pass along information about ITBs that it creates.
The application must keep track of creating and deleting ITBs.
Subsequent applications installing tasks must assume that only system
ITBs are available and cannot use ITBs created by other applications. ◆

IMPORTANT

Each ITB uses additional on-chip memory. Installing ITBs can cause
a section to be saved and restored between modules because of a lack
of on-chip memory. ITBs should only be allocated when absolutely
necessary and should be removed as soon as they are not needed. ▲
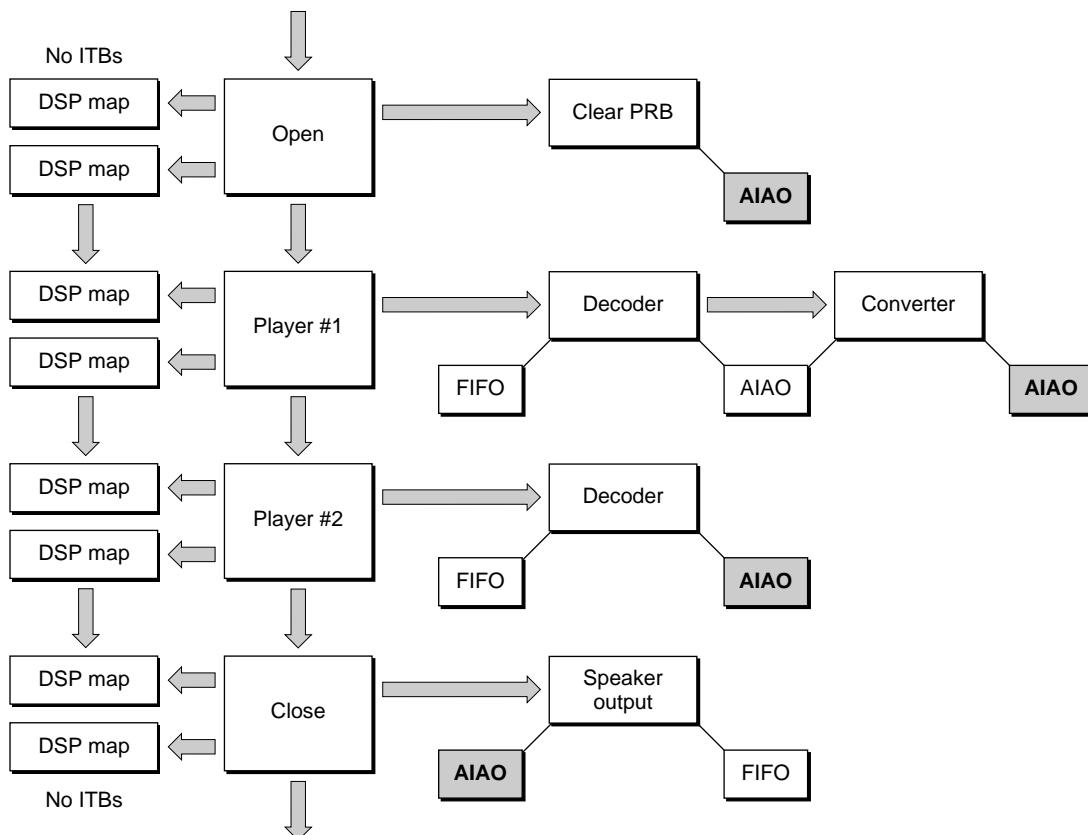
In the sound player example, a sound output buffer is required to pass the summed output data between all sound player tasks. At the beginning of the chain of sound player tasks, the buffer must be opened and cleared. Then, the output buffer of each player must be connected in turn to this buffer. This is accomplished by reserving cache memory for the buffer using the mechanism described above. A structure called the **DSP map** is used to contain the ITB information. The DSP map provides information to the Real Time Manager's cache allocation routines to make the ITB connections and forestall overruning reserved memory.

During initialization, the Sound Driver automatically signs in to the DSP subsystem and installs the standard sound task list. This creates system ITBs that can be used by other tasks inserted into the standard sound task list. Once the last sound task has finished its operation on the ITB, the ITB can be sent to the speaker FIFO. Once the ITB is dumped to the output FIFO, it must be closed so subsequent tasks will have access to the cache memory that the ITB had been using.

Figure 3-28 is a diagram illustrating this concept. This diagram includes an open buffer task that defines the intertask buffer and clears it, followed by two sound player tasks, and then the close task that closes the ITB after dumping the results to the speaker FIFO.

Figure 3-28   Example of intertask buffers

The intertask sound AIAO buffers in Figure 3-28 are shown in gray. They are connected as the input and output sections of the modules using the ITB connection mechanism. This results in the DSP maps that are used to define the ITBs. There is always an input DSP map and an output DSP map for every task. An open task always has more buffers in the output DSP map than in the input DSP map. Likewise, a close task always has more buffers in the input DSP map than are in the output DSP map.

Note
In Figure 3-28 the open and close operations have been shown as separate tasks. These operations could also be included as part of player 1 and player 2, respectively. ◆

Routines are provided in the Real Time Manager to open and close ITBs. ITBs are also used as the backbone of the sound subsystem, as described in "Standard Sound," later in this chapter.

Unified I/O Architecture

There are many cases where a DSP module could be connected to either an AIAO buffer or a FIFO buffer. For example, the CD-XA decompressor module can be used to supply the input to another module, using an AIAO buffer, but can also be used to create a disk file of uncompressed sound, using a FIFO buffer.

For a module with one input and one output buffer, there are four possible cases for using FIFOs and AIAOs: FIFO to FIFO, AIAO to AIAO, FIFO to AIAO, and AIAO to FIFO. There are two ways to avoid having four different modules to support these four cases:

■ A standard FIFO-to-AIAO and AIAO-to-FIFO module can be used whenever an application needs to connect a module to a FIFO.

■ A mechanism can be provided to allow either a FIFO or AIAO connection to an AIAO input or output buffer. This has the advantage of reducing the overhead associated with parsing and loading a module. It also provides a somewhat more general unified I/O concept.

The Real Time Manager provides the second capability, by letting a FIFO section be connected to an AIAO section. In this case, the AIAO section is "converted" to a FIFO section. Likewise, an AIAO section can be connected to a FIFO rather than another AIAO section, by creating a FIFO with the `DSPNewFIFO` routine. This mechanism operates without requiring control by the module. Data appears in the input buffer automatically (having been piped from the output AIAO of the previous module) or by having been read from a FIFO by the DSP operating system. This works equally well for output buffers.

This automatic connection feature provides a powerful generic I/O system for DSP modules. It allows modules to be used in many different ways without having to add "helper" modules. It also results in substantially less processing overhead.

Another extension of the FIFO system is the support of linked buffers, which allow the DSP to work in a virtual memory environment. When data is being passed from the disk file system, it is also not necessary to move the data from the disk buffer to a separate FIFO to play it. This reduces the overhead for disk-related I/O.

Execution Models

As explained in "Software Architecture," earlier in this chapter, there are two different execution models for real-time data processing: AutoCache and DemandCache. AutoCache works for many types of functions and is easier to program and use, making it the model of choice in most cases.

DemandCache is the more general model, and can be used to implement very complex functions. Unlike AutoCache modules, DemandCache modules can generate run-time errors. In general, the DemandCache model requires more work from the programmer.

The difference between the two models is in the caching mechanism. For AutoCache, the caching is done automatically, based on source-code flags the programmer sets up for each section in the module. Caching is further supported by a series of "standard" section macros, such as `NewInputFIFOAndScalableBufferSection`, `NewCachedProgramSection`, `NewOutputPRBSection`, `NewParameterSection`, and others. Each macro sets all the necessary flags for the proper caching operation for the section, making it much less likely for errors to be made. For example, the `NewCachedProgramSection` macro sets up caching for a load operation, but not for a save operation. It also specifies the correct cache bank (Bank A) and the section data type (code).

DemandCache, on the other hand, requires that the program explicitly activate the caching functions for each section in the module, using DSP operating system routines. The programmer is responsible for the proper sequencing of these routines. This adds to the workload but also allows various combinations of sections to be cached as needed.

Both models require that the first section be a code section. However, in DemandCache, this section is in off-chip memory when called, and therefore will run much slower than in AutoCache. Normally only a small bootstrap program should be included in this section, which caches the "real" code section and calls it once it is in on-chip memory. It is often more efficient to cache even small code sections on-chip rather than executing them off-chip.

Both models use a section table that is constructed by the Real Time Manager's allocation routines during the installation process. This table contains a list of pointers to appropriate containers, and is used to determine the active location of each section by the module code. These pointers provide the basis for *section-relative addressing* for the Real Time Manager. Relative addressing is required to allow the Real Time Manager the freedom to locate sections in the cache in the most efficient manner. The program must find the location of the sections by using the appropriate macros.

Section Control Flags

The flags that affect how a section is handled by AutoCache and DemandCache are listed below. It is important to distinguish the difference between source-code flags and the run-time flags. Since the Real Time Manager can clear some of the flags when making connections, the flag settings may be different at run time than are indicated by the source code.

■ The Load flag indicates that a section should be loaded on-chip. In the general case, this flag indicates a move from one container to another, and can be cleared by the Real Time Manager to eliminate a buffer move. This is useful if the buffer is already on-chip from the module that generated the data.

■ The Clear flag indicates that a section should be zeroed before use. This flag should not be set if the Load flag is set, since loading a section and then clearing it does not make sense. It is normally used in conjunction with the Save flag. The Real Time Manager can clear this flag if data already exists in a PRB, and will leave it set if this module is the first to be connected to a PRB.

■ The Save flag indicates that a section should be saved off-chip. In the general case, this flag indicates a reverse move from one container to another, and can be cleared by the Real Time Manager to prevent a buffer move. This would be the case if a following module wanted to use the on-chip buffer as input data.

■ The Static flag indicates that a section is to be allocated using the static allocation method rather than the dynamic allocation method. Static allocation is done at installation time by the Real Time Manager. Dynamic allocation is done at run time by the module program. This flag facilitates connected buffer management for DemandCache. It is ignored in AutoCache modules; the Real Time Manager will always set the Static flag for AutoCache sections, so its setting doesn't matter in the source code when using AutoCache.

■ Bank flags include two preference flags, the Bank A flag and the Bank B flag. If neither are set, this condition specifies external memory for the primary container. If either flag is set, it specifies a cached section and which on-chip bank is preferred. If both flags are set it also indicates a cached section; however, either bank may be used. Notice that it is an error if you specify a Static flag section with no Bank flags set. It is also illegal to specify a Load flag or Save flag with no Bank flags set. For a better understanding of the purpose of banked memory for DSP's, refer to the AT&T DSP3210 manual or other signal processing literature.

It is very important to correctly understand these flags and how they operate. One of the most common errors in programming the DSP is setting an illegal or inappropriate combination of flags. Table 3-3 and Table 3-4, later in this chapter, list combinations of flag settings to watch out for.

Setting Up Input and Output for Connections

When specifying a section as an input buffer, the Load flag should be set, along with a Bank flag (usually Bank B). If you connect this section to another module's section using the Real Time Manager, the Load flag will be cleared if the section is already on-chip from a previous module. On the other hand, if the section is off-chip, or has to be moved off-chip to make room in the cache for an intervening module, the flag will not be cleared.

When specifying a section as a partial result buffer, the Clear flag and Save flag should be set, along with a Bank flag (usually Bank B). If the buffer is connected to another module and will remain on-chip, the Real Time Manager will clear the Save flag. The Clear flag is always cleared except for the first module summing into a PRB. The PRB operations pertaining to sound are covered in "Standard Sound Task List," later in this chapter.

When specifying a section as a complete result buffer, only the Save flag should be set. This is the usual setting for modules other than sound modules.

AutoCache Execution Model

The AutoCache function is activated by a flag in the module header. When the DSP operating system prepares to execute the module it begins with a pre-cache process that preloads any sections indicated by the Load flag in the section data structures. Likewise, any section having the Clear flag set in the section structure is cleared. This is accomplished by the DSP operating system during the parsing of the section structure. This process supports up to 32 sections.

In the example shown in Figure 3-20, the program, variables, and table sections are loaded into the cache. The input buffer is connected to an existing AIAO buffer in the cache, so the Load flag is cleared by the Real Time Manager. Since the output buffer is not the first to connect to the sound output PRB, the Clear flag is cleared. The Save flag is also cleared, since the buffer is to remain on-chip after the module completes execution.

The section table for AutoCache always contains the primary container pointers. These pointers are on-chip for cached sections and workspace, and off-chip for noncached sections, such as parameter buffers used for main processor/DSP interaction. The section table does not change during the execution of the module and is always on-chip.

Once the precache process is complete the first section is called. The first section must always be a code or program section. The code section uses the section table to access information in the other sections. Programmers should use the standard macros to obtain the base address of the section (`GetSectionAddress`) or the address of a label within a section (`GetSectionLabel`). When execution is complete, the module returns control to the DSP operating system.

At this point, the postcache process occurs. All sections with Save flags are copied back to their secondary sections. In the example shown in Figure 3-20, only the variables section must be cached back to local memory. The output buffer is left on-chip. It is important to keep in mind that the Real Time Manager can clear only unneeded flags, based on the connections made. This prevents the Real Time Manager from overriding cases where the programmer decides that a section should always be off-chip to conserve cache space.

Table 3-3 summarizes all the possible flag combinations for AutoCache at run time. The three groups of cases are for one-container sections, two-container sections, and illegal combinations. The Real Time Manager uses the Bank flags to determine the number of containers. If no Bank flags are set, there is a single container off-chip. If one or more Bank flags are set, and either the Load flag or Save flag is set, then two containers are used: one off-chip and one on-chip. If a connection is made which eliminates the need for a secondary container, it is deleted, and the section becomes a one-container section.

The Real Time Manager can also set up cases where both primary and secondary containers are on-chip. It uses this mechanism to automatically move sections to conserve cache space.

Table 3-3   Run-time AutoCache flag combinations

Flags					
Bank	Static	Load	Save	Clear	**Comments**
					Single-container case—Bank flags not set
–	■	–	–	–	Off-chip section (parameters, workspace)
–	■	–	–	■	Off-chip initialized workspace
■	■	–	–	–	Allocate on-chip workspace
■	■	–	–	■	Allocate a cleared on-chip workspace
					Dual-container case—one or more Bank flags set
■	■	–	■	–	Save primary to secondary on exit
■	■	–	■	■	Allocate a clear primary on entry, save to secondary on exit
■	■	■	–	–	Load primary from secondary on entry
■	■	■	■	–	Load primary from secondary on entry, save to secondary on exit
					Illegal cases—caching with one container or dynamic allocation
?	–	?	?	?	Illegal—can't AutoCache with a dynamic section
–	■	■	?	?	Illegal—can't AutoCache with one container
–	■	?	■	?	Illegal—can't AutoCache with one container
■	■	■	–	■	Useless/illegal
■	■	■	■	■	Useless/illegal

NOTE   ■ = flag set,   – = flag cleared,   ? = do not care what flag is set to

▲   **W A R N I N G**
The foregoing method of providing buffer moves cannot
be used for DemandCache. ▲

DemandCache Execution Model

DemandCache is used for complex situations such as the following:

■ More than 32 sections are required.

■ A module must make a decision as to which sections must be cached, based on some program status.

■ A module has one or more large functions which must be broken down into smaller steps that will fit in the cache. This requires a sequence of cached sections.

The DemandCache execution model supports any number of sections, but the section table cannot be placed in the cache if the number of sections gets too large. The programmer must make the trade-off between faster execution (in-cache section table) and more cache space available (out-of-cache section table). This selection is done by setting a flag in the module header.

The Real Time Manager determines if two containers are needed by observing the Static flag. If it is set (static allocation), two containers are required. Otherwise, only a single container is required (dynamic allocation). In effect, a static section is allocated in the same way sections are allocated in AutoCache, at module installation rather than during run time.

Note
Since the Static flag is ignored for AutoCache, standard section macros, such as `NewPRBOutputSection`, set this flag. This eliminates the need to have two sets of macros. ◆

Note
While two containers are allocated initially for static sections, the section may become a one-container section once the Real Time Manager has done its work. Clearly, if neither Save flags nor Load flags are set the secondary container is not needed. ◆

Caching is accomplished with calls to the DSP operating system. There are only two routines: `PushSection` and `PopSection`. These routines can do different things, depending on the state of the section control flags. Whenever a section is accessed with these routines, the section table is updated.

The DSP operating system names `PushSection` and `PopSection` reflect the stack-like structure of DemandCache. The `PushSection` routine pushes a section onto the cache stack, and the `PopSection` routine pops a section off of the cache stack. There are actually two different stacks: the Bank A stack and the Bank B stack. Two stacks are necessary to support the maximum performance operation for dual bank caches.

▲ **WARNING**
The `PushSection` and `PopSection` routines will cause a run-time error if neither Bank flag is set, because a section without a bank preference is an external section. ▲

As in AutoCache, the section table is prebuilt during the load process by the Real Time Manager. For AutoCache, the values in the table are always the primary container pointers. DemandCache is more complex—the values in the section table are different for dynamic and static sections. Each case is described separately below.

DemandCache for Dynamic Sections

Most of the sections in a DemandCache module are dynamic, and are allocated by the programmer on the stack. Only a single container is required for these sections. The primary container is always an off-chip buffer, and the secondary container pointer is nil. The `PushSection` routine creates a temporary container on the stack, and the `PopSection` routine removes it. Both `PushSection` and `PopSection` routines update the section table: `PushSection` changes the pointer from the primary container to the newly created stack container and `PopSection` does the reverse.

Note
The container pointers in the section data structure
are not modified by either of these routines. ◆

All section flags affect the operation of the `PushSection` and `PopSection` routines. A `PushSection` routine uses the Load flag, Clear flag, and Bank flags. The `PopSection` uses only the Save flag. The Static flag's impact on `PushSection` and `PopSection` operations is discussed next in "DemandCache for Static Sections."

When `PushSection` is called, a block of memory is allocated in the appropriate stack (selected by the Bank preference flags). If the Load flag is set, the contents of the primary container is copied to the stack. Otherwise, the allocated space is work space. If the Clear flag is set, the allocated stack space is cleared. In effect, you can specify that the work space be either cleared or not cleared with this mechanism. Both flags should not be set.

When `PopSection` is called, the stack space is copied back to the primary container if the Save flag is set. Then the stack space is deallocated.

If you access a dynamic section prior to a `PushSection` call or after a `PopSection` call, you will access the primary container in external memory. Accesses between the `PushSection` and `PopSection` calls will be directed to the section in the cache stack.

▲ **WARNING**
It is the programmer's responsibility to update the pointers after using
the `PushSection` or `PopSection` routine. The DSP operating system
only updates the section table, not the registers used by the programmer
in the module. ▲

Notice that there are run-time errors that can occur with these routines. For `PushSection`, insufficient space in the stack or no Bank flags will cause an error. For `PopSection`, an error will result if the section space in the stack is not at the top of the stack. In either case, the DSP operating system will mark the task as inactive and send a message to the client indicating the type of error.

DemandCache for Static Sections

Static sections are used in DemandCache as a mechanism to provide efficient inter-module buffer management. This use is supported by the Real Time Manager and is one of the central services of the real-time data processing architecture. This connection mechanism allows the creation of generic modules that can be interconnected in many different ways for different purposes, without explicit programming. To support all possible cases, however, the module programmer must follow the rules for the I/O sections that are to be connected.

Static sections are allocated at module installation time by the Real Time Manager. This is very similar to the allocation of sections in the AutoCache model. When `PushSection` is called, the section table is updated with the primary container pointer. If the Load flag is set, the contents of the secondary container is copied to the primary container. Otherwise, the primary container is work space. If the Clear flag is set, the primary container is cleared. In effect, you can specify that the work space be either cleared or not cleared with this mechanism. Both flags should not be set.

If the Save flag is set when `PopSection` is called, the primary container is copied back to the secondary container and the section table is updated with the secondary container pointer. If the Save flag is not set, the section table is updated with the secondary container pointer only if the pointer is not `nil`. The pointer is `nil` for single container sections and contains a valid address for loaded sections.

To summarize, static sections work very much like AutoCache except that caching operations are carried out under the control of the module programmer rather than prior to and after module execution.

No run-time errors can occur with static sections.

Connections in DemandCache

It is possible to make connections to either dynamic or static sections using the Real Time Manager. If a connection is made to a dynamic section the primary container is shared with the other connected sections in local memory, external to the cache. This can be used to reduce local memory requirements or pass parameters between modules. This type of connection is made without specific module programming. In effect, the primary container pointer is changed to point to some other container and the original container is deleted.

For static sections, connections are made as with AutoCache, using two containers. Such connections are used to pass buffers between modules without the overhead of moving them off-chip and then back on-chip for a subsequent module. The section table contains the secondary container pointer until a `PushSection` call is made, after which it contains the primary container pointer. When a `PopSection` call is made, the section table is restored to the secondary container pointer.

For some types of connections, the secondary container is not required. For example, if an input section is already on-chip from a previous module, the Load flag is cleared and there is no use for the secondary container. In this case, the secondary container is

eliminated and the secondary pointer is set to `nil`. The primary pointer is placed in the section table in place of the `nil` secondary pointer. Notice that neither `PushSection` nor `PopSection` actually move memory in this case.

▲ **WARNING**
This means that the DemandCache programmer must always make the `PushSection` and `PopSection` calls in the code for static input and output sections. By making the calls, the correct function is executed, depending on the connections made. This makes the generic connection mechanism available to the DemandCache programmer. ▲

For example, suppose you have a simple module with a single input section (ignoring other sections). This input can be connected by the client that is installing the module into the task. Since it is a general-purpose input buffer, the Load flag and Static flags should be set.

If the client makes a connection to an output buffer from a previous module, and the buffer is already on-chip, then the Load flag will be cleared, and the primary container pointer will point on-chip to that buffer. The secondary container is deleted, and `nil` is stored in the secondary pointer. The primary container pointer is placed in the section table. When `PushSection` and `PopSection` are called, no action is taken. Notice that the program may access the buffer before the push operation or after the pop operation, if desired

If the client instead makes a connection to a previous module with an off-chip output buffer, the Real Time Manager will not clear the Load flag, and the secondary container will be valid. The secondary container pointer will be put into the section table. The `PushSection` routine will load the off-chip buffer into the on-chip primary container and update the section table pointer. The `PopSection` routine will restore the secondary container pointer in the section table. Accessing the secondary container prior to the push gives access to the input data off-chip (slow access). It does not make sense to access the data after the pop operation, since the data is not saved.

Hence, off-chip buffer accesses are not recommended prior to a push or after a pop operation. Since the location pointed to by the section table is certain to be on-chip only after the push and before the pop operation, accessing data on-chip can be guaranteed only between these routines.

Table 3-4 summarizes the flag combinations for DemandCache at run time.

Notice in Table 3-4 that although it is not illegal to specify or use a section that does not have any Bank flags set, it is always illegal to push or pop such a section. Other illegal combinations include static sections with no bank preference flags, and cached sections (Load flag or Save flag set) with no bank preference.

Introduction to Real-Time Data Processing

Table 3-4   Run-time DemandCache flag combinations

Flags					`PushSection`	`PopSection`
Bank	Static	Load	Save	Clear		
					Single-container dynamic case—Static flag is not set	
■	–	–	–	–	Allocate space in stack	Deallocate space in stack
■	–	–	–	■	Allocate and clear stack workspace	Deallocate space in stack
■	–	–	■	–	Allocate space in stack	Save to primary and deallocate
■	–	–	■	■	Allocate and clear stack workspace	Save to primary and deallocate
■	–	■	–	–	Allocate and load stack from primary	Deallocate space in stack
■	–	■	■	–	Allocate	Save to primary and deallocate
					Single-container static case—Static flag is set	
■	■	–	–	–	No operation	No operation
■	■	–	–	■	Clear primary	No operation
					Dual-container static case—Static flag is set	
■	■	–	■	–	Section table change only	Save primary to secondary
■	■	–	■	■	Clear primary	Save primary to secondary
■	■	■	–	–	Load primary from secondary	Section table change only
■	■	■	■	–	Load primary from secondary	Save primary to secondary
					Illegal cases—if no Bank flags are set	
	–	–	–	?	Off-chip section, push illegal	Off-chip section, pop illegal
–	■	?	?	?	Static illegal if no Bank flag set	Static illegal if no Bank flag set
–	?	■	?	?	Load illegal if no Bank flag set	Load illegal if no Bank flag set

continued

Table 3-4   Run-time DemandCache flag combinations (continued)

Bank	Static	Load	Save	Clear	**PushSection**	**PopSection**
–	?	?	■	?	Save illegal if no Bank flag set	Save illegal if no Bank flag set
■	–	■	–	■	Useless/illegal	Useless/illegal
■	–	■	■	■	Useless/illegal	Useless/illegal
■	■	■	–	■	Useless/illegal	Useless/illegal
■	■	■	■	■	Useless/illegal	Useless/illegal

NOTE   ■ = flag set,   – = flag cleared,   ? = do not care what flag is set to

FIFO Connections

It is possible to connect standard AIAO input and output buffers to a FIFO buffer. This can be done in one of two ways:

■ Connect a FIFO section to an AIAO section with the `DSPConnectSections` routine.

■ Create a FIFO for an AIAO I/O section with the `DSPNewFIFO` routine.

In the first case, the AIAO section is marked as a FIFO section, and the secondary container becomes the `DSPFIFO` data structure. One section should be an output section (Save flag set), and the other should be an input section (Load flag set). In the second case, a new `DSPFIFO` structure is created as the secondary container, and the AIAO section is marked as a FIFO section.

The DSP operating system recognizes these flag combinations, FIFO plus Load flag or FIFO plus Save flag, and does a `DSPFIFORead` or `DSPFIFOWrite` automatically, in place of a normal block copy. This allows the client to make connections with FIFOs without changing the module code. For AutoCache, the FIFO operations occur during the precache and postcache processes. For DemandCache, the FIFO operations occur during push and pop operations.

▲ **W A R N I N G**
In DemandCache, it is inappropriate to access an input or output section before a push or pop operation, because a FIFO connection may have been made by the client. ▲

The `SetTaskInactive` flag in FIFO operations makes it possible to do useful things with FIFO connections. For example, it is possible to set up a sound player and connect its input to a sound FIFO. By setting the `SetTaskInactiveOnEmpty` and `SetTaskInactiveOnUnderrun` flags, the player will automatically stop when the buffer has played. A zero fill is done on the last `DSPFIFORead`, which creates silence at the end of the sound if it is not an integral number of frames in length.

Grouped Modules

AutoCache and DemandCache provide enough flexibility for most functions. If a job cannot be accomplished with AutoCache, it is almost certain to be possible with DemandCache.
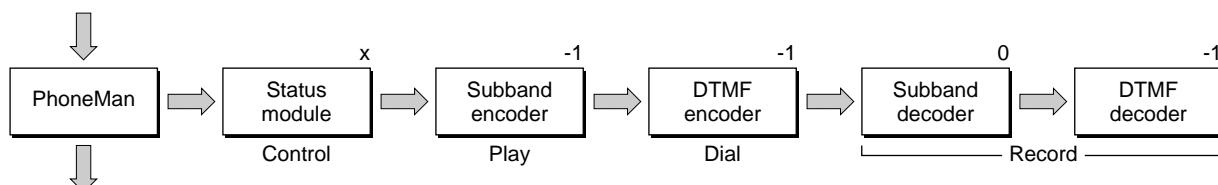
There are cases where having one big complex module to handle a complex case is not a good solution. This is where a set of already existing standard modules, if properly used, could get the job done. Another case is when automatic function replacement is desirable. For example, a system might consist of several very distinct functions, all implemented as separate modules. It is possible to replace a single module with a new and better one if the functions are separate rather than integrated into a single module. This is especially important if the modules were created by different programmers.

To support this type of building-block philosophy, a field called *skipcount* is added to each module. The skipcount number is normally set to zero, which indicates that the DSP operating system should proceed normally to the next module in the task or jump to the next task if the next module pointer is `nil`. The skipcount can also be set to –1, which means that the DSP operating system should exit the task at the completion of the module, ignoring any additional modules that may follow.

If the skipcount has any value from 1 up, it tells the DSP operating system how many modules to skip over in the current task before continuing module execution. If the DSP operating system runs out of modules in the task while skipping, it automatically exits the task, and begins the next one. This is not considered a run-time error. The skipcount from one task cannot affect a following task.

Here is an example of how the skipcount can be used in a telephone answering function, where a status and control module is required. The module is responsible for detecting rings, taking the phone line offhook, and hanging up the phone. Other functions needed include a recorder function, a play message function, and a dial function. The recorder function may need a DTMF decoder to detect remote control functions. The recorder may use a compression module, and the player may use a decompression module. One possible arrangement of these modules is shown in Figure 3-29.

Figure 3-29   Example of DSP task for telephone answering

In this example, the status module contains the control function. It normally has a skipcount of –1 when the answering machine is idle. When a play function is needed the skipcount is set to 0. When the dial function is needed the skipcount is set to 1. When the record function is needed the skipcount is set to 2. The fact that the skipcount has several possible values is indicated with an "x" in Figure 3-29.
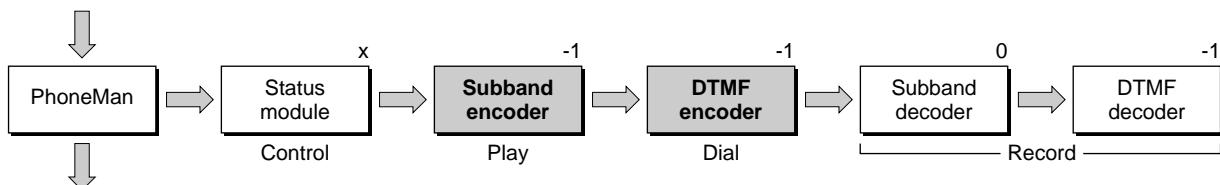
The skipcount can be set initially by calls to the Real Time Manager. Since the count is normally 0, the value –1 should be stored in the status module, subband encoder, DTMF encoder, and subband decoder modules. The last module count does not matter but should be set to –1. The status/control module will change its own skipcount as needed using the DSP operating system routines provided.

GPB for Grouped Modules

The telephone-answering task described in the last section never actually uses all of the modules within it at the same time. Hence it would be inappropriate to add up the GPB requirements for all the modules and allocate that much time for this function. A flag, called `DontCountThisModule`, is included in the module to handle this case. If set, it tells the Real Time Manager not to include the GPB of a module in the total GPB requirements for the task. It is up to the client that is installing this group to determine which is the worst-case utilization of modules, and to mark all of the other modules with the flag. In the example, the record case is probably the worst case. Therefore, the dial and play modules should have their `DontCountThisModule` flags set. This is shown in Figure 3-30, where modules with the flag set are shown with shading. Only the unshaded modules will have their GPB requirements added to the task total.

Figure 3-30   Controlling GPB in grouped modules

Module Scaling

The Macintosh real-time data processing architecture makes it possible to use modularized digital signal processing functions in different configurations, reducing or eliminating the need to rewrite DSP modules whenever a slightly different operation is required. To accomplish this, the architecture provides unified I/O buffers and connections to make up new applications from existing modules. It also includes the ability to scale a module's buffers to accommodate different frame rates, sample rates, and GPB requirements.

It is possible for a client to encounter one of three sample rates for sound or telecom, plus one of two (at least) frame rates. It would be unfortunate if a different module was required for each of the possible cases. The real-time data processing architecture solves this problem by providing a means of scaling any module to fit the current frame and sample rate, as well as select the appropriate GPB estimate for the module's functionality.

There are several characteristics that change for a module when either the frame rate or sample rate change. First, the required GPB number changes. A module that takes 5000 GPB units to execute at 100 frames per second will only take 2500 GPB units at

200 frames per second. The second characteristic that changes is the number of repeat loops the module must execute to complete processing for a frame. Again, at the 200 frames per second rate only half as many samples must be processed or generated per frame. A module may also change its GPB requirements based on an outside loading factor.

There are some types of modules that normally do not operate at different frame rates or sample rates—for example, a DTMF decoder. This is because a DTMF decoder looks for specific frequencies and must assume a sample rate. A more advanced version of a DTMF decoder could be created that would adjust automatically to the sample rate.

A good example of a module that can easily function at various sample rates and frame rates is a sample rate converter. If the converter has a fixed conversion rate, it can operate just as well on 32 kHz data as 8 kHz data. Likewise, it can operate just as well on 240 samples per frame as 120 samples per frame. A more advanced sample rate converter would be able to adjust automatically to changes in the input frequency by changing the conversion factor. This capability requires that a module be able to request more or less GPB even though the sample rate and frame rate have remained fixed.

The Real Time Manager provides a mechanism for the module programmer to specify scalability, whenever it is possible. This is done with the use of *scaling vectors* and the GPB mode (AutoCache or DemandCache). A scaling vector contains three values: the frame rate, the scale factor, and the GPB value. Any number of scaling vectors can be supplied for a module. They work for both AutoCache and DemandCache modules. In addition, a scaling vector can have one frame rate and scale factor with multiple GPB estimates.

The module code itself determines the number of samples it has to work with by using the `GetSectionSize` macro. By using this on its input buffer, for example, the module can determine the repeat count of its processing loop. Generally, this is the only information required by the module. It contains both sample rate and frame rate information in a single number. The GPB mode must be specified if the module has variable GPB requirements.

The frame rate value is used to select the appropriate scaling vector. The scale factor is used to determine the size of the scalable I/O buffers for the module.

The scalability of I/O buffers is indicated by setting the `ScalableSection` flag. This flag should be set only for AIAO I/O buffers. The Real Time Manager determines the actual size of the section for a given scaling vector by multiplying the size set in the source code by the scaling vector's scale factor.

In a 2 to 1 sample rate converter, for example, scaling vectors might be generated for the following cases:

■ 100 frames/sec. 24 kHz to 12 kHz

■ 200 frames/sec. 24 kHz to 12 kHz

■ 100 frames/sec. 8 kHz to 4 kHz

- 200 frames/sec. 8 kHz to 4 kHz

- 100 frames/sec. 16 kHz to 8 kHz

- 200 frames/sec. 16 kHz to 8 kHz

- 100 frames/sec. 48 kHz to 24 kHz

- 200 frames/sec. 48 kHz to 24 kHz

- 100 frames/sec. 32 kHz to 16 kHz

- 200 frames/sec. 32 kHz to 16 kHz

Note
The only information the module would need to operate at any of these rates is the amount of cache space and the size of either the input or output buffer. All of the required information to support these 10 cases can be supplied by 10 scaling vectors. ◆

Since this module is a 2 to 1 converter, the size of the input buffer in the source code should be set to 2, and the output buffer should be set to 1. The 10 scaling vectors would look like this:

Mode 0	100, 120, 5000	[100 f/s, scale = 120 (I/O size of 240/120), GPB = 5000]
Mode 1	200, 60, 2500	[200 f/s, scale = 60 (I/O size of 120/60), GPB = 2500]
Mode 2	100, 40, 1666	[100 f/s, scale = 40 (I/O size of 80/40), GPB = 1666]
Mode 3	200, 20, 833	[100 f/s, scale = 20 (I/O size of 40/20), GPB = 833]
Mode 4	100, 80, 3333	[100 f/s, scale = 80 (I/O size of 160/80), GPB = 3333]
Mode 5	200, 40, 1666	[200 f/s, scale = 40 (I/O size of 80/40), GPB = 1666]
Mode 6	100, 240, 10000	[100 f/s, scale = 240 (I/O size of 480/240), GPB = 10,000]
Mode 7	200, 120, 5000	[200 f/s, scale = 120 (I/O size of 240/120), GPB = 5000]
Mode 8	100, 160, 6666	[100 f/s, scale = 160 (I/O size of 320/160), GPB = 6666]
Mode 9	200, 80, 3333	[200 f/s, scale = 80 (I/O size of 160/80), GPB = 3333]

Notice that there are several cases where the same values for GPB and scale factor are used, but a different frame rate is selected. This is necessary to provide a mechanism for the module programmer to exactly specify the conditions under which the module will operate. There may be cases where the module will not operate correctly when run at a different frame rate with the same GPB and scale factor. An example of this is a module that is designed to operate at specific frequencies, such as a DTMF decoder.

Note
In the example, the GPB numbers are neatly set to round numbers. In reality, the numbers may not be so round, because of bus and DSP operating system overhead. ◆
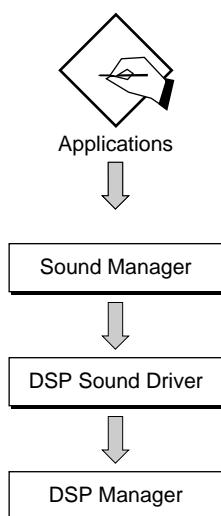
Selecting Module Scale Factor

The process for selecting the scale for a module is simple. When the client wishes to load a module, a Real Time Manager call to `DSPLoadModule` is made. One of the parameters provided to the Real Time Manager is the scale factor. The current DSP frame rate is provided automatically. If there is a matching scaling vector, the scalable sections are scaled appropriately and the correct GPB is used. If there is no matching scaling vector, an error is returned. The client must make a selection based on the required I/O buffer sizes. The associated scale factor is then passed to the loader.

Standard Sound

Standard sound consists of a set of tasks installed in the real-time task list plus the DSP Sound Driver. The Sound Driver acquires control over the sound port, installs and maintains the sound tasks, and handles the interface to the Macintosh Sound Manager. The goal of the Sound Driver is to provide DSP sound support to all applications that do not use the DSP, and to provide a plug-board architecture for sound applications that use the DSP. The relationship between these different layers of software is shown in Figure 3-31.

Figure 3-31 DSP Sound Manager and Sound Driver

The function of the Sound Driver interface is to provide a hardware-independent interface to the Macintosh Sound Manager that can accept requests for work to be done. The interaction between available GPB and work requests is handled automatically by the Sound Driver. If there is insufficient DSP processing time available, the Sound Driver
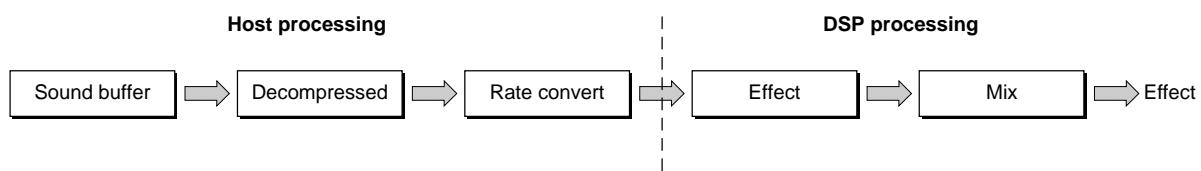
rejects the work request, and the Sound Manager does the work itself. This mechanism prevents a heavily loaded DSP from preventing essential sound functions from operating. The DSP is used if it is available, and normally results in better sound.

Sound Manager Interface

The Sound Manager approach is to provide a series of blocks for processing sound from the raw data to the output. This diagram is shown in Figure 3-32.

Figure 3-32   Sound Manager processing

The dividing line between 68000 processing and DSP processing moves from right to left in Figure 3-32, depending on the amount of DSP processing available; the more DSP processing available, the farther left the line goes. The mix and speaker output is always done in the DSP. However, if additional channels of sound are required, and the DSP is fully loaded, the additional channels will have to be "premixed" before arriving at the DSP. If sufficient DSP processing is available, the entire processing chain for each channel will be handled by the DSP. The processing of each channel is done by a task installed in the player patch point, described in the next section. In the example above, the task consists of a decompressor module, a sample rate converter module, and a special effects module. All modules are interconnected via AIAO buffers. The input buffer for the decompressor is a FIFO and the output buffer is the sound output buffer.
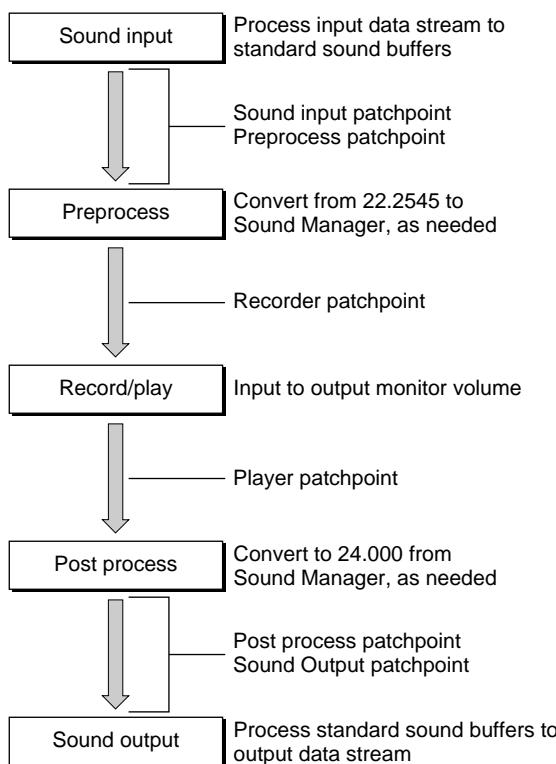
Standard Sound Task List

The standard sound task list consists of five separate tasks, as shown in Figure 3-33. The first and last task are responsible for handling the serial port data stream and for opening and closing two ITBs that pass stereo data in DSP floating-point format between all five of the tasks. These buffers are used to provide the standard sound patch points.

Note
The standard sound tasks are not named to represent what the task does but rather are named for the patch points associated with them. ◆

Figure 3-33   Standard sound task list

The sound input task takes the serial data stream from the DMA channel coming from the stereo A/D converter and converts it to DSP floating-point format. For 10 ms frames and 24 kHz sound, this takes the form of two 240-longword buffers. These buffers are established as ITBs by the DSP map structure associated with sound input.

If an application wishes to get access to the raw input data stream, it installs a task after the sound input task. Such a task should not modify the sound input.

The preprocess task provides a stereo or mono data stream to the sound input at any of the standard 8-bit rates of 7, 11, and 22 kHz, and 16-bit sound at the current rates of 24, 32, or 48 kHz.

If an application wishes to preprocess the sound datastream prior to it reaching any recorder tasks, it installs a task prior to the preprocess task.

The record/play task handles the monitor function of the standard sound plug board. This function allows none or all of the input sound to be passed to the output sound.

If an application wishes to record sound, it installs a task before the record/play task. If an application wishes to play sound, it installs a task after the record/play task.

The postprocess task accepts a mono or stereo sound channel from the Sound Manager at the standard 8-bit rates of 7, 11, and 22 kHz, or 16-bit sound at the current rates of 24, 32, or 48 kHz.

If an application wishes to postprocess the mix from all players, it installs a task after the postprocess task.
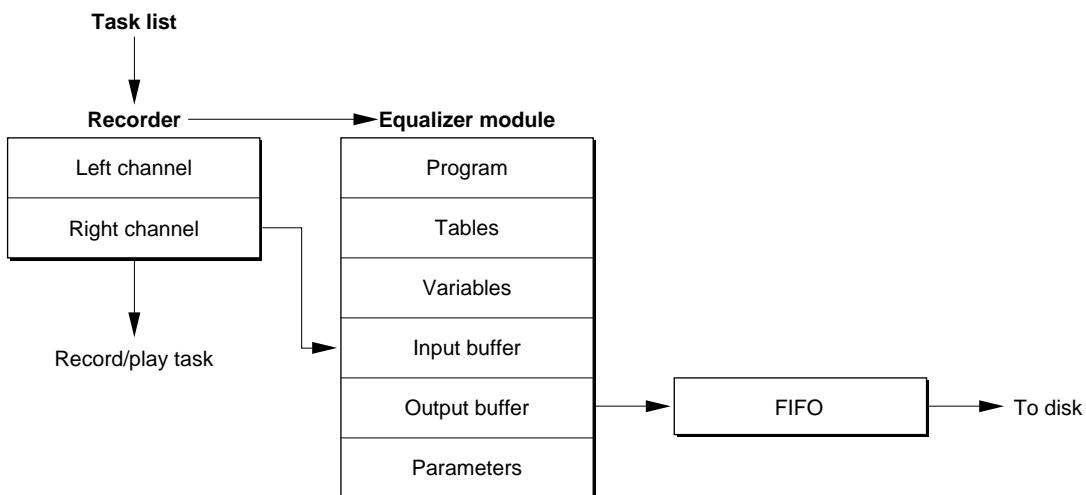
The sound output task accepts the final mixed and processed output data stream in DSP floating-point format. It takes care of the system volume control function, converts the data to the serial port format, and transports it to the sound output buffer. The DMA channel passes this data to the sound D/A converters.

If an application wishes to access the data stream after all players and postprocessing, it installs a task before the sound output task. Such a task should not modify the output sound stream.

The operation of the standard sound plug-board depends on the function of AIAO buffers and the Real Time Manager. The two sound ITBs are opened by the sound input task, and passed to each task in the list. The sound output task closes the ITBs. This operation is covered in "Buffer Connections Between Tasks," earlier in this chapter.

There are two different ways a sound module can be installed in this task list. For these examples, assume the application is installing a five-band active equalizer module (forming one task). This module can be installed as a preprocessor or part of a more complex preprocessor function, as a recorder or part of a recorder, as a player or part of a player, and as a postprocessor or part of a postprocessor. Figure 3-34 shows how the module can be used as a recorder.

Figure 3-34   Equalizer used as a recorder task

In this example, assume that the equalizer is monophonic. Its input buffer is a scalable AIAO buffer, and its output buffer is a scalable AIAO buffer. This means that the Load flag is set for the input buffer, and the Clear flag and Save flags are set for the output buffer. These buffers were created using the `NewScalableInputBuffer` and `NewScalablePRBOutputBuffer` macros.
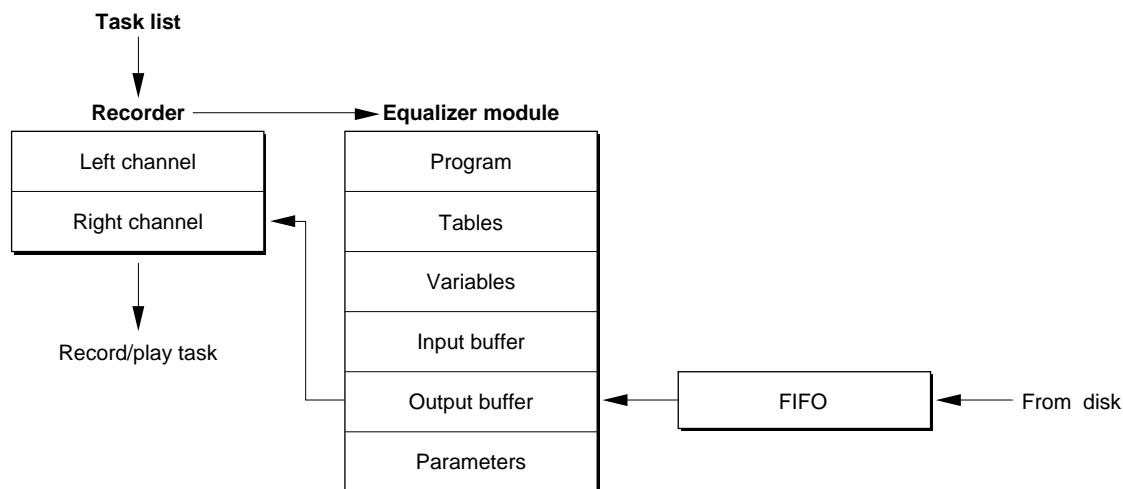
The application uses the Real Time Manager to make a connection between the right channel ITB (shown as part of the record/play task) and the input buffer section of the equalizer. Since the ITB already exists on-chip, the Load flag is cleared by the Real Time Manager, and the input buffer primary container is made to be the same as the right channel buffer. This completes the input connection.

The output buffer section is connected to a FIFO, using the `NewFIFO` routine. The application uses this FIFO to buffer the data to the disk. In this case neither the Clear flag or Save flags are cleared by the Real Time Manager.

When executing, the output buffer is cleared, the equalizer reads its input data from the right channel buffer, and then the equalizer sums its output into the output buffer. Finally, the DSP operating system writes the buffer to the FIFO. The data recorded is the same data format and sample rate as the input data, but has been processed through the equalizer filters.

The next example connects the same module as a player. A player task is installed after the record/play task, as shown in Figure 3-35.

Figure 3-35   Equalizer used as a player task
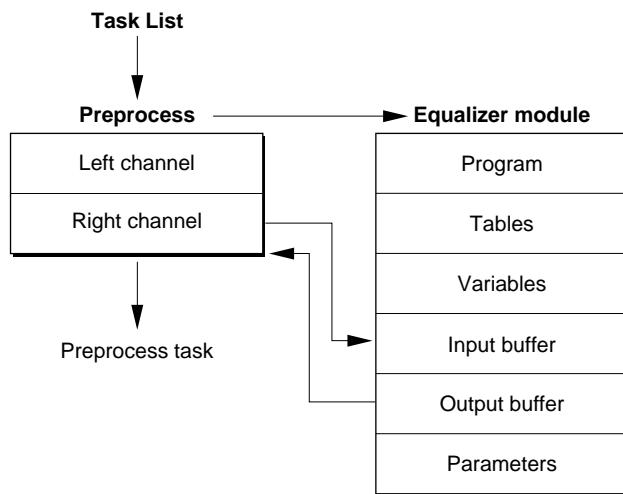
When the connection is made from the output buffer to the right channel buffer, the Clear flag is cleared, since this module is not the first to input to the right channel buffer. Likewise, the Save flag is cleared, since the right channel buffer already exists on-chip. Thus, any already existing signal in the buffer is directly summed into by the equalizer. This is a normal or *direct connection*.

Note
The function of this buffer as a summing buffer is a fundamental property of the standard sound plug board. ◆

In Figure 3-36, the equalizer task is used as a sound preprocessor.

Figure 3-36 Equalizer used as a preprocess task

In this case, both the input buffer and the output buffer are connected to the same ITB. If the normal summing connection was made between the output buffer and the right channel buffer in this case, the desired result would not be achieved. Rather, the sum of the signal and the filtered signal would be in the right channel buffer. In this case, the client must make an *indirect connection* between the two buffers. This type of connection is selected with one of the `ConnectSections` routine parameters.

An indirect connection is the same as a direct connection, except a buffer move is required—either the Save flag of the source buffer or the Load flag of the destination buffer must be set (or not cleared). The Clear flag of the source buffer must be left set. This results in the equalizer summing into a cleared buffer, followed by the DSP operating system saving the filtered sound into the right channel buffer, overwriting the previous sound data.

The final example is to use this same equalizer as a postprocessor. The layout would look exactly like Figure 3-36, except the location of the task would be different.

These examples show the versatility of the standard sound plug-board architecture. In addition, the utility of the Real Time Manager to create connections between FIFOs and AIAOs, and to provide both direct and indirect connections, allows the standard sound modules to be used in very flexible ways.

Sample Rate and Frame Rate Changes

There are two factors that affect the real-time DSP processing of sound: the sample rate of the sound I/O subsystem, and the frame rate of the DSP operating system. While it is possible to handle these two factors separately, it is more convenient to handle them together. This simplifies the standard sound task list.

This approach offers three different "gear shifts" for sound:

- standard sound (24 kHz sample rate, 10 ms frame rate)

- high fidelity sound (32 kHz sample rate, 5 ms frame rate)

- professional sound (48 kHz sample rate, 5 ms frame rate)

The advantage of this approach is that there is sufficient room in the cache to support all of these selections, including the dual intertask buffers. If 48 kHz was supported at 10 ms frames, for example, the buffers would be too big to both fit in the cache along with the DSP operating system and module code.

Real Time Manager

This chapter describes the Macintosh system software routines for accessing, controlling, and monitoring the digital signal processor (DSP) subsystem in the Macintosh Quadra 840AV and Macintosh Centris 660AV. The DSP subsystem provides real-time processing for applications that require a guaranteed throughput. It also provides processing for applications that perform timeshare processing.

Before you read this chapter you should already be familiar with

■ Macintosh programming

■ using resource files

■ the concepts of digital signal processing given in Chapter 3, "Introduction to Real-Time Data Processing"

A brief synopsis of the Real Time Manager is provided first. Valuable information about getting started with the DSP is included in "About the Real Time Manager." This is followed by a top-down explanation of how an application accesses the different levels of the DSP architecture. Next is an example showing the necessary commands in the order they would probably be used. Optional commands are included to provide you with programming alternatives. At the end is a Real Time Manager Reference, organized in top-down order. Routines are listed for each level of the Real Time Manager in logical groupings.

For information about installing and debugging DSP programs in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Appendix A, "DSP d Commands for MacsBug," Appendix B, "BugLite User's Guide," and Appendix C, "Snoopy User's Guide."
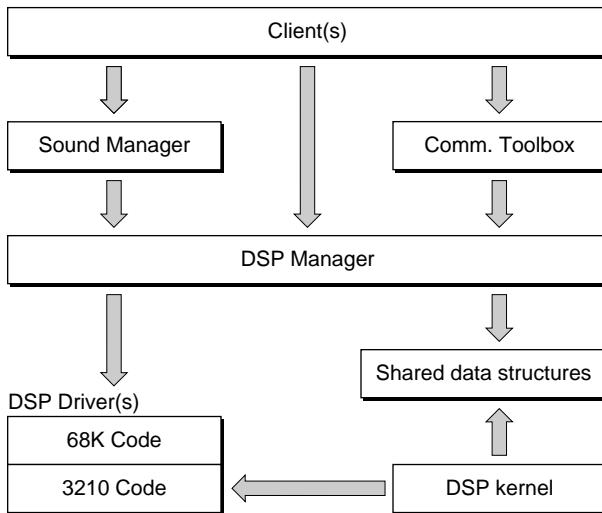
About the Real Time Manager

The Real Time Manager provides access to the capabilities of the Macintosh DSP subsystem. DSP code modules for performing the desired data manipulations must already be available as resources. Apple provides a standard set of DSP code modules as resources for performing sound and telecommunications input and output. These functions include data compression, sample-rate conversion, disk store and recall, and playing sound files directly from disk. Examples are described in "Standard Sound Task List," in Chapter 3.

Real Time Manager Structure

Figure 4-1 gives a high level architectural view of the DSP subsystem. To maintain implementation independence, the Real Time Manager must perform all operations on the DSP subsystem. Programmers wanting to maintain compatibility should always use the Sound Manager and communications toolboxes provided by Apple. For maximum performance the Real Time Manager is accessible directly. However, error handling should be included for use on computers without the DSP.

Figure 4-1   DSP subsystem overview

To determine if the DSP subsystem is available use the Gestalt command. See "Machine Identification," in Chapter 1.

To check compatibility with the version of the Real Time Manager installed in the system, the `DSPManagerVersion` routine must be called. This routine should always be called early in the application to determine if its DSP tasks can be installed.

```
pascal unsigned long DSPManagerVersion(void)
```

The Sound Manager includes code that sets up and maintains a set of standard system tasks to support input and output for the stereo codec and mono/stereo sound player, as well as multiple sample rates and sample rate changes. Calls made to the Sound Manager are always performed by the system. If there is no DSP subsystem available, the system performs the desired function using the main processor.

The Communications Toolbox handles similar functions for the telephone and data communications serial port. Calls made to the Communications Toolbox will only be performed if a DSP subsystem is available.

Memory allocation routines in the Real Time Manager handle the DSP's on-chip memory, module bandwidth allocation, and intertask buffer (ITB) allocation. Since these routines are specifically for the DSP subsystem there are no equivalent system routines.

All blocks of memory indicated by a value of type `DSPAddress` are by definition locked contiguous and non-cacheable. They are locked contiguous so that the DSP does not have to worry about scatter/gather operations when using a `DSPAddress` value. The blocks are locked non-cacheable to eliminate conflicts that would occur when the DSP modifies a memory location that the host processor had cached. Since the DSP can only address physical memory, it cannot use virtual memory.

The type `DSPAddress` is a general type. Subordinate types include `DSPFIFOAddress`, `DSPTaskAddress`, `DSPModuleAddress`, and `DSPSectionAddress`. Each has the same data structure as `DSPAddress`, but is used for a specific purpose. This distinction allows type-checking of pointers in the source code.

Guaranteed Processing Bandwidth

Guaranteed processing bandwidth (GPB) provides a simple but flexible mechanism that allows the Real Time Manager to monitor and limit execution time, thereby guaranteeing real-time execution. This mechanism dynamically adapts to systems with different hardware characteristics. GPB is defined by parameters of type `DSPCycles`, as shown in Listing 4-1.

Listing 4-1   DSP bandwidth structure

```
struct DSPBandwidth {              // bn = bandwidth
   DSPCycles      bnEstimate;    // worst-case pre-runtime
   DSPCycles      bnActual;      // worst-case runtime
   unsigned long  bnFlags;       // control flags
   };
```

The units for `DSPCycles` are DSP instruction cycles. The parameter `cpuMaxCycles` that is part of the `DSPCPUDeviceParamBlk` data structure, shown in Listing 4-3, indicates the maximum number of instruction cycles available for each DSP CPU device. At any given instant, the instruction cycles consumed by the modules installed in the real-time task list of a device cannot exceed this maximum allowable limit.

Each module has three parameters to support the GPB system. These are: `bnEstimate`, `bnActual`, and `bnFlags`. The estimated value is included in the module resource. It is the DSP programmer's predicted worst-case prediction for how many DSP cycles it takes to execute this module. The `bnActual` and `bnEstimate` values are maintained in the `DSPBandwidth` data structure, shown in Listing 4-1.

The parameter `bnActual` is a computed value and is updated by the DSP operating system at run time. The actual and estimated processing requirements are controlled by the `bnFlags` parameter, using these values:

```
kdspLumpyModule       // use bnEstimate
kdspSmoothModule      // use bnActual
```

The `kdspSmoothModule` flag indicates that the `bnActual` value of processing time should be used. The flag is preset if the module always takes the same amount of execution time, or set by the module itself using the `GPBSetUseActual` routine. If the flag is not set, the `bnEstimate` value is used instead. If the flag is set by the module, it is set one or more frames after determining that the module's worst-case process has been executed.
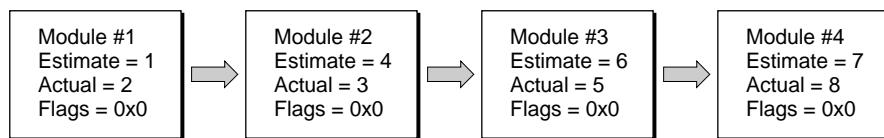
The `kdspDontCountModule` routine specifies that this module is used in a grouped set, and should not be included in the overall GPB calculations. In a grouped set of modules the module with the worst-case GPB is used by calling `DSPCountModule`.
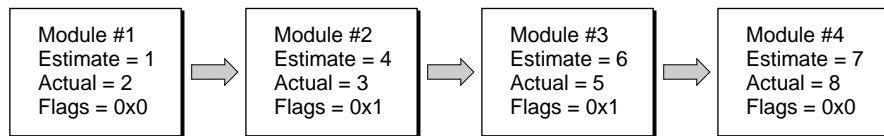
Figure 4-2 shows some examples of module sequences and their associated `DSPBandwidth` numbers. The `DSPBandwidth` numbers are used differently according to the values of `bnFlags`.

Figure 4-2   Examples of different GPB values
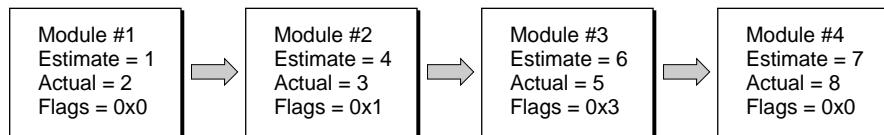
Case where GPB = 1 + 4 + 6 + 7 = 18 `DSPCycles`

Module #1	Module #2	Module #3	Module #4
Estimate = 1	Estimate = 4	Estimate = 6	Estimate = 7
Actual = 2	Actual = 3	Actual = 5	Actual = 8
Flags = 0x0	Flags = 0x0	Flags = 0x0	Flags = 0x0

Case where GPB = 1 + 3 + 5 + 7 = 16 `DSPCycles`

Module #1	Module #2	Module #3	Module #4
Estimate = 1	Estimate = 4	Estimate = 6	Estimate = 7
Actual = 2	Actual = 3	Actual = 5	Actual = 8
Flags = 0x0	Flags = 0x1	Flags = 0x1	Flags = 0x0

Case where GPB = 1 + 3 + 0 + 7 = 11 `DSPCycles`

Module #1	Module #2	Module #3	Module #4
Estimate = 1	Estimate = 4	Estimate = 6	Estimate = 7
Actual = 2	Actual = 3	Actual = 5	Actual = 8
Flags = 0x0	Flags = 0x1	Flags = 0x3	Flags = 0x0

Devices and Clients

The Real Time Manager provides a flexible but efficient interface that allows multiple applications, working either separately or together, to operate simultaneously on multiple DSP subsystems without risk of conflict. To accomplish this it is necessary to arbitrate the use of the DSP device drivers.

The term *client* identifies any software entity that is using the Real Time Manager. Before a client can request resources from the Real Time Manager, it must first find what DSP devices are available using the `DSPGetIndexedCPUDevice` routine. Then the client signs into the DSP device using the `DSPOpenCPUDevice` routine and receives a unique identification number, its `pbhClientRefNum`. The identification number is used for arbitration of the individual DSP device drivers. This client reference number is

also used to facilitate a controlled sign-off of device driver clients. If for any reason a client calls `ExitToShell` before deallocating all of its resources (primarily memory and processor bandwidth) the Real Time Manager will automatically deallocate those resources.

▲ **WARNING**
If you are writing a system extension that will use the Real Time Manager, be aware there is only a limited amount of memory available because the system heap cannot expand. You will need to include a `'zsys'` resource in your system extension to cause the system heap to be enlarged before the extensions run. The amount of memory needed may be more than required by your system extension because some of the memory may be used by LocalTalk, EtherTalk, TokenTalk, and A/ROSE. ▲

Note
The `pbhClientRefNum` value is unique only to the DSP device driver that issued it. Both `pbhClientRefNum` and the `pbhDeviceIndex` value must be used for complete identification of a specific client. ◆

A DSP CPU device is defined to be one DSP subsystem device driver. The data structures `DSPDeviceParamBlk` and `DSPCPUDeviceParamBlk` are used to pass information about a DSP CPU device between the Real Time Manager and the client. These data structures are shown in Listing 4-2 and Listing 4-3. It is up to the client application to keep track of information it needs from the parameter list. Initially all values except `pbhClientName` and `pbhClientICON` are 0.

Listing 4-2   DSP device parameter block structure

```
struct DSPDeviceParamBlk {
    unsigned short    pbhDeviceIndex;      // index for this device
    unsigned short    pbhClientPermission; // client's read/write permission
    DSPClientRefNum   pbhClientRefNum;     // reference number for this client
    Str31             pbhClientName;       // name of this client
    Handle            pbhClientICON;       // handle to client's icon
    Str31             pbhDeviceName;       // name for the I/O or cpu device
    Handle            pbhDeviceICON;       // icon for the I/O or cpu device
    };
```

Note
The `DSPDeviceParamBlkHeader` data structure is defined in the include file as `DSPDeviceParamBlk`. This construct is used so that the information can be more easily included in other data structures; for example, in the `DSPCPUDeviceParamBlk` structure shown in Listing 4-3. ◆

Listing 4-3   CPU device parameter block structure

```
struct DSPCPUDeviceParamBlk {
   DSPDeviceParamBlkHeader
   unsigned char   cpuSlotNumber;            // slot number the card is in
   unsigned char   cpuProcessorNumber;       // processor #, zero-based
   OSType          cpuProcessorType;         // type of the processor
   DSPCycles       cpuMaxCycles;             // max processor execution
                                             //   cycles per frame
   DSPCycles       cpuAllocatedCycles;       // number of real-time cycles plus
                                             //   overhead currently allocated
   DSPCycles       cpuCurRealTimeLoading;    // number of cycles used
                                             //   during the last frame
   DSPCycles       cpuTimeShareLoading;      // number of cycles it took
                                             //   for timeshare list
   DSPCycles       cpuTimeShareFreq;         // how often the timeshare
                                             //   list is run
   unsigned long   cpuFrameRate;             // number of frames per second for
                                             //   this cpu device
   MessageActionProc cpuClientMessageActionProc; // client's message handler
   };
```

Note

For more information about `MessageActionProc`
see "Sending Messages," later in this chapter. ◆

Information can be retrieved about a specific client by using `DSPGetClientInfo`. The
information is returned in the `DSPClientInfoParamBlk`, shown in Listing 4-4.

Listing 4-4   Client information parameter block structure

```
struct DSPClientInfoParamBlk {
   DSPClientRefNum   ciClientRefNum;       // returned ref num of the client
   unsigned short    ciClientPermission;   // returned read/write
                                           //   permission of the client
   Handle            ciClientICON;         // returned icon of the client
   Str31             ciClientName;         // returned name of the client
   };
```

Tasks

The `DSPTask` structure is used for storing information about a digital signal processing task. To generate a task follow the procedure in "Using the Real Time Manager," later in this chapter.

Notice that it is at the task level that the Real Time Manager checks to see if there is enough DSP processing bandwidth (described as GPB) to accept the additional work load. This is a logical choice for three reasons. First, it does not make sense to install some of the algorithms that make up a task only to find out that there is not enough GPB to install the rest. Second, if algorithms (as DSP modules) were installed one at a time they could potentially each start on a different frame. By installing them all at once as a task, they automatically all start on the same frame. Third, if one of the modules takes excessive processing time for some reason, it makes sense to remove the entire task rather than remove just one piece of the task. Such a partial removal would most likely prove disastrous.

A client can retrieve information about a specific task with the `DSPGetTaskInfo` routine. The information is returned in `DSPTaskInfoParamBlk`, shown in Listing 4-5. For more information about `MessageActionProc` see "Sending Messages," later in this chapter.

Listing 4-5   Task information parameter block structure

```
struct DSPTaskInfoParamBlk {
   DSPTaskRefNum      tiRefNum;   // returned reference number for this task
   unsigned long      tiRefCon;   // returned application-specific info
   MessageActionProc tiVector;    // returned vector of task action proc
   unsigned long      tiFlags;    // returned flags for DSPTask control
   Str31              tiName;     // returned name of this DSPTask
   };
```

Passing data between tasks is accomplished with intertask buffers (ITBs). The ITBs must be allocated in the first task needing to pass data to another task. The ITBs must be de-allocated by the last task using them. For more information about ITBs see "Data Buffering," in Chapter 3.

Modules

This section discusses the Macintosh system software that controls DSP modules.

Module Definition

DSP modules are the basic building blocks for constructing tasks for the DSP. A module is typically broken into a number of sections. A section can contain code, data, variables, and can be an input or output buffer. The module structure contains the information needed to let the Real Time Manager and DSP operating system perform their functions, including linking modules, controlling the execution of a module, and so on. Each module also contains information about how much processing bandwidth it requires and how many times it has executed since it was installed.

Note

To install a module into a task follow the procedure in "Using the Real Time Manager," later in this chapter. ◆

A client can get information about a specific module by using the kdspGetModuleInfo routine. The information is returned in the DSPModuleInfoParamBlk, shown in Listing 4-6.

Listing 4-6   Module information parameter block structure

```
struct DSPModuleInfoParamBlk {
    DSPModuleRefNum      miRefNum;         // reference number of this module
    struct DSPBandwidth  miGPB;           // guaranteed processing bandwidth
    unsigned long        miFlags;         // module flags
    unsigned long        miNumSections;   // number of sections in module
    Str31                miName;          // name of the DSP module
    unsigned long        miExecutions;    // number of executions
    unsigned long        miSkipCount;     // number of modules to skip
    };
```

Execution Flow for Modules

The DSP operating system executes the DSP modules that are installed in a task. In a typical operation, it simply scans the list executing each of the modules sequentially. However, there are cases when modules need to be executed in a selective fashion. To facilitate this functionality at the DSP operating system level, the miSkipCount field is included in the module data structure. This field controls the execution flow within a task, not between tasks. It allows a module to specify that one or more modules
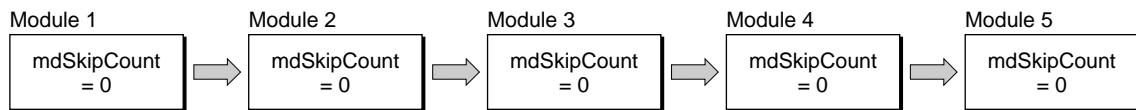
following it are to be skipped. The `miSkipCount` field is set equal to the number of modules to be skipped. To set a modules skip count, use

```
pascal OSErr DSPSetSkipCount (DSPModuleRefNum theModuleRefNum,
    unsigned long theCount)
```
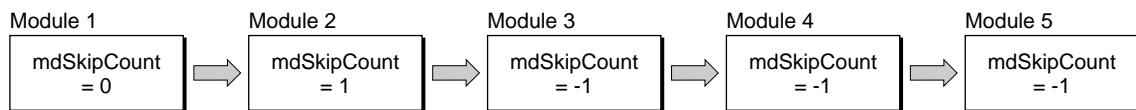
The DSP operating system always executes the first module in a module list. After executing a module it copies the module's `miSkipCount` field. It decrements through the module list until the copied value reaches –1, then resumes execution at the selected module. (For example, a skip count of 0 causes execution of the next module. A skip count of 2 jumps two modules and continues execution at the third module.) If a module has a skip count of –1, execution of the module list is stopped and execution resumes at the next task. If the list of modules within the task ends before the skip count reaches –1 it is considered an error condition and the DSP operating system continues execution at the next task; however, the DSP operating system does not report this error. Figure 4-3 shows examples of how the skip count mechanism is used.

Figure 4-3   Examples of different execution paths

Execution path = 1, 2, 3, 4, 5

Module 1		Module 2		Module 3		Module 4		Module 5
mdSkipCount = 0	⇒	mdSkipCount = 0	⇒	mdSkipCount = 0	⇒	mdSkipCount = 0	⇒	mdSkipCount = 0

Execution path = 1, 2, 4

Module 1		Module 2		Module 3		Module 4		Module 5
mdSkipCount = 0	⇒	mdSkipCount = 1	⇒	mdSkipCount = -1	⇒	mdSkipCount = -1	⇒	mdSkipCount = -1

Execution path = 1, 2, 3

Module 1		Module 2		Module 3		Module 4		Module 5
mdSkipCount = 0	⇒	mdSkipCount = 0	⇒	mdSkipCount = -1	⇒	mdSkipCount = -1	⇒	mdSkipCount = -1

Execution path = 1, 4, 5

Module 1		Module 2		Module 3		Module 4		Module 5
mdSkipCount = 2	⇒	mdSkipCount = -1	⇒	mdSkipCount = -1	⇒	mdSkipCount = 0	⇒	mdSkipCount = -1

Sections

This section discusses the Macintosh system software that controls DSP sections within modules.

Section Definition

The `DSPSection` structure is used to define a section type. Sections are usually buffers in host memory that can be moved (cached) to internal DSP static random-access memory (SRAM) prior to program execution, and optionally saved back to host memory after program execution. Sections include the program, tables or coefficients, state variables, temporary variables, input and output data, and control information (parameter buffers).

There are two basic types of sections: one-container and two-container. Any section that is going to be cached from external memory to DSP memory or vice versa must be a dual container section. One container is in main memory, the other is in DSP internal memory. The one-container section needs no caching or saving, and usually represents one of these forms of memory:

■ temporary storage on-chip, in automatic variables of the work space or intermediate buffers

■ large areas in main memory, such as data arrays or FIFO buffers

■ common areas shared between modules, such as AIAO buffers, or between a module and the Macintosh system, such as parameter buffers

The section structure includes two pointers to handle the two-container case: `scPrimary` and `scSecondary`. In the one-container case, the `scSecondary` pointer is `nil`.

A client can retrieve information about a specific section by using the `DSPGetSectionInfo` routine. The information is returned in the `DSPSectionInfoParamBlk`, shown in Listing 4-7.

Listing 4-7 Section information parameter block structure

```
struct DSPSectionInfoParamBlk {
   DSPSectionRefNum  siRefNum;     // reference number for this section
   unsigned long     siSize;       // size of data in actual section
   unsigned long     siFlags;      // section flags
   unsigned long     siType;       // for section connection type checking
   Str31             siName;       // name of this section
   Ptr               siPrimary;    // current location of section data
   Ptr               siSecondary;  // optional section data storage location
```

```
DSPSectionRefNum  siPrevSection; // previous connection
DSPSectionRefNum  siNextSection; // next connection
DSPFIFORefNum     siFIFORefNum;  // RefNum of the FIFO
};
```

Section Flags and Data Types

The `siFlags` field of the `DSPSectionInfoParamBlk` contains flags that provide information about the section. The use of the `siFlags` field is detailed in Table 4-1.

Table 4-1   Section flags

siFlags	Usage
kdspLeaveSection	Do not load or save this section
kdspLoadSection	Load this section
kdspSaveSection	Save this section
kdspClearSection	Fill this section with zeroes
kdspSaveOnContextSwitch	Save this section on context switch
kdspStaticSection	This section statically allocated before runtime
kdspExternal	Never loaded on chip
kdspBankA	Load in Bank A if possible
kdspBankB	Load in Bank B if possible
kdspAnyBank	Load anywhere
kdspFIFOSection	Section is a FIFO buffer
kdspNotIOBufferSection	All cases other than below
kdspInputBuffer	Section is an input buffer
kdspOutputBuffer	Section is an output buffer
kdspScalableSection	Section size can be scaled
kdspDSPUseOnly	Only DSP should modify this memory

The section flags assist the Real Time Manager and DSP operating system with some of the buffer management and FIFO operations of the system. The `ScalableSection` flag must be set whenever it is desirable for the system to automatically scale the module for different sample rates or frame rates.

The `siType` field of the `DSPSectionInfoParamBlk` contains flags that provide information that is useful for type checking when connecting sections of several modules, such as the output buffer of one section to the input buffer of the next section. The use of the `siType` field is detailed in Table 4-2.

Table 4-2   Section data-type flags
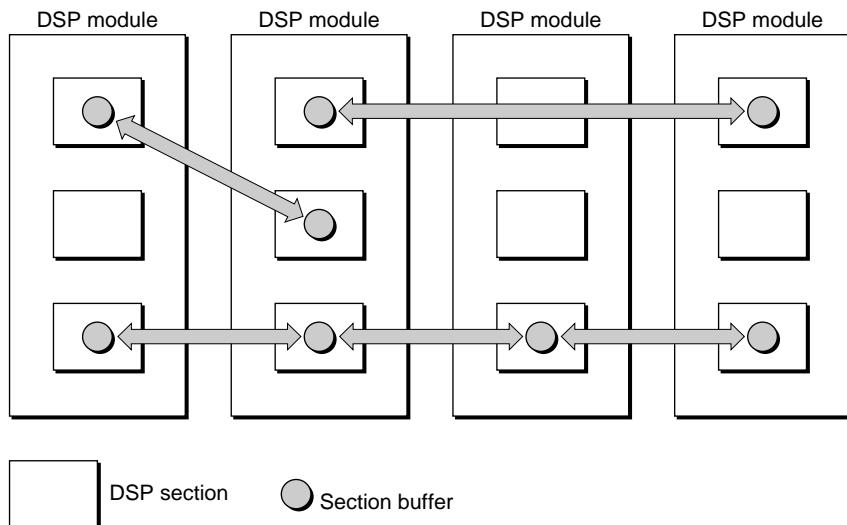
siType	Usage
kdspNonData	Data in section is beyond description
kdsp3200Float	Data is 3200 float format
kdspIEEEFloat	Data is in IEEE float format
kdspInt32	Data is 32-bit integer
kdspInt1616	Data is 16-bit integer packed
kdspInt8888	Data is 8-bit integer packed
kdspmuLaw	Data is `muLaw` format
kdspALaw	Data is `Alaw` format
kdspAppSpecificData	Data is application-specific

Connecting Sections

To improve efficiency, it is often desirable to connect two or more sections from two or more modules. This allows data to be passed from one module to the next without moving the data off the chip and then back on. This feature is useful in patchcording algorithms, where the output of one operation can be left on-chip for the next operation to use as input. It is also useful in optimizing the execution of a task. If one task uses the same section in multiple modules, a performance gain can be achieved by leaving the section on-chip between module executions. It is also possible to connect a FIFO to an AIAO. When this occurs the DSP operating system loads data from the FIFO into the AIAO (or AIAO to FIFO) every frame.

Figure 4-4 shows some of the possible connections of sections between four modules. The double arrows indicate forward/backward pointers.

Figure 4-4   Section interconnection

Using the Real Time Manager

Before building a task and installing it into the DSP, the application must determine if the DSP resources are available. This is done using the Gestalt and DSPManagerVersion routines.

Gestalt

The Gestalt routine is used to determine if the Real Time Manager is available. It must always be called before any attempt is made to install a DSP task.

```
pascal unsigned long Gestalt (void)
```

DESCRIPTION

The Real Time Manager constant for Gestalt is

```
gestaltRealtimeMgrAttr   'rtmr'
```

DSPManagerVersion

The `DSPManagerVersion` routine returns the version number of the installed DSP software.

```
pascal unsigned long DSPManagerVersion (void)
```
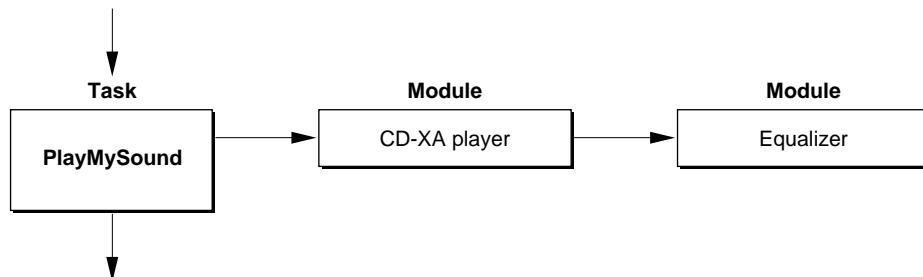
The `DSPManagerVersion` routine is used to maintain version compatibility between the client's tasks and the Real Time Manager. It must always be called before any attempt is made to install a DSP task.

The constant for `DSPManagerVersion` is `kdspManagerBuildVersion`.

Accessing the DSP

An example of a complete task is shown in this section to help clarify how Real Time Manager routines and macros work. Figure 4-5 shows an overview of an example task.

Figure 4-5   Task example

The task PlayMySound is shown installed in the task list. It points to two DSP modules. The first module, CD-XA player, is used to decompress CD-XA data from disk into PCM data. The equalizer module is used to adjust the tone quality of the sound, and feeds the results to the sound partial result buffer (PRB). The sequence of Real Time Manager calls that is necessary to set up and execute a task such as this is given in Table 4-3. Italicized calls are made only if necessary.

Table 4-3   Setting up a task

Routines	Usage
DSPGetIndexedCPUDevice	Get information about the available DSPs
DSPOpenCPUDevice	Sign in as a client to one of the DSPs
DSPNewTask	Create the empty task structure
DSPLoadModule	Load a module (from resource) into the task structure
DSPNewFIFO	Allocate memory for a FIFO section
DSPGetSection	Get pointer to the named section (to connect sections)
DSPConnectSections	Connect sections to pass data between modules
DSPInsertTask	Insert the task into the execution stream of the DSP
DSPGetSectionData	Get pointer to section data (for shared parameters)
DSPSetTaskActive	Tell the DSP to execute the task

DSPGetIndexedCPUDevice

The DSPGetIndexedCPUDevice routine returns a pointer to the indexed CPU device parameter block.

```
Pascal OSErr DSPGetIndexedCPUDevice (DSPCPUDeviceParamBlkPtr
    theCPUParamBlk)
```

Field descriptions

→	pbhDeviceIndex	Index of desired device.
←	pbhDeviceName	Returned name of device.
↔	pbhDeviceICON	Returned icon of device.
←	cpuSlotNumber	Returned NuBus slot # of device.
←	cpuProcessorNumber	Returned processor # of device.
←	cpuProcessorType	Returned type of processor.
←	cpuMaxCycles	Max number of cycles for the processor.
←	cpuAllocatedCycles	Number of real-time cycles plus DSP operating system overhead currently allocated.
←	cpuCurRealTimeLoading	Number of cycles used during the last frame.
←	cpuCurTimeShareLoading	Number of cycles it took for timeshare list.
←	cpuTimeShareFreq	Number of times the timeshare list is run.
←	cpuFrameRate	Frame rate in frames/sec.

The cpuProcessorType constants can be:

```
kdsp3210          '3210'
kdsp32C           '32C '
```

DESCRIPTION

The `DSPGetIndexedCPUDevice` routine is used by a client to find information about the DSP devices installed. By calling `DSPGetIndexedCPUDevice` a client can create a list containing `theCPUParamBlk` for each available device. The client should pass an index starting at 0 (in `pbhDeviceIndex`) and increment the index until `kdspInvalidIndexErr` is returned. The client should also allocate space for the icon buffer `pbhDeviceICON` or pass `nil` if no icon is desired. This routine does not allocate memory.

Note
If the icon is not desired, be sure to set `pbhDeviceICON` to `nil` before calling `DSPGetIndexedCPUDevice`. ◆

DSPOpenCPUDevice

The `DSPOpenCPUDevice` routine requests access to the specified CPU device.

```
pascal OSErr DSPOpenCPUDevice (DSPCPUDeviceParamBlkPtr
    theCPUParamBlk);
```

Field descriptions

→	pbhDeviceIndex	The index for this device.
→	pbhClientPermission	The client's requested permission.
←	pbhClientRefNum	The client reference number.
→	pbhClientICON	The client's icon.
→	pbhClientName	The name of this client.
→	cpuClientMessageActionProc	Location of client's message action procedure.

Valid constants for `pbhClientPermission` are:

kdspReadPermission	Requesting read-only access to the device.
kdspReadWritePermission	Requesting read/write access to the device.

DESCRIPTION

The `DSPOpenCPUDevice` routine is called by a Real Time Manager client to request use of a device driver. If `theCPUDeviceParamBlkPtr` is a pointer to a `theCPUParamBlk` data structure that was set up by calling `DSPGetIndexedCPUDevice`, then the client need only fill in the `pbhClientName`, `pbhClientPermission`, and `cpuClientMessageActionProc` fields. The two parameters `pbhClientICON` and `pbhClientName` are optional.

For more information about `MessageActionProc` see "Sending Messages," later in this chapter.

Creating a Task

The sections that follow describe the sequence of events required to create a task. The task, PlayMySound, has two modules, uses a FIFO buffer, and has one section connected between the two modules.

After calling `DSPGetIndexedCPUDevice` and `DSPOpenCPUDevice` to open and sign into a DSP device, call `DSPNewTask` to create the task structure.

DSPNewTask

The `DSPNewTask` routine returns a reference number to a new task.

```
pascal OSErr DSPNewTask (
    DSPCPUDeviceParamBlkPtr    theCPUDevice,
    MessageActionProc          theVector,
    StringPtr                  theTaskName,
    DSPTaskRefNum              *theRefNum);
```

Field descriptions

→	theCPUDevice	A pointer to info about the CPU device being used.
→	theVector	The message vector for this task. This vector will be used to make calls to the application when the code modules contained in the task fail to operate within the processing requirements specified.
→	theTaskName	The name of this task.
←	*theRefNum	The returned reference number for this task.

DESCRIPTION

The `DSPNewTask` routine allocates a `DSPTask` data structure for a new task in the Real Time Manager's heap. `DSPNewTask` does not actually insert the new task into the task list of any DSP device. The `DSPInsertTask` routine is used to do the actual insertion. The `DSPNewTask` routine generates a reference number for the task.

The client must use the task routines in a specific sequence. First, the task must be created. This only allocates the task structure, it does not allocate any memory for the task. Second, the task must be assembled by loading DSP modules into it. Third, the task must be installed into the task list. Fourth, the task must be activated. This sequence can be accomplished using the following calls:

1. `DSPNewTask` Create a new task structure

2. `DSPLoadModule` Load modules into task structure

3. `DSPInsertTask` Insert task into the appropriate task list

4. `DSPSetTaskActive` Activate task to execute

For more information about `MessageActionProc` see "Sending Messages," later in this chapter.

Loading a Module

You call the `DSPLoadModule` routine to load a module (CD-XAPlayer) into the task structure.

DSPLoadModule

The `DSPLoadModule` routine loads the specified module into the specified task at the specified position.

```
pascal OSErr DSPLoadModule (
    StringPtr           theName,
    DSPTaskRefNum       theTaskRefNum,
    DSPPosition         thePosition,
    DSPModuleRefNum     theReferenceModuleRefNum,
    DSPModuleRefNum     *theNewModuleRefNum,
    long                theScale);
```

Field descriptions

→	theName	The name of the resource you want to load.
→	theTaskRefNum	A pointer to the task structure you want the resource loaded into.
→	thePosition	The position where this module should be loaded into the task. If a reference module is specified, install before or after the reference module (within the already existing list of modules in this task). If no reference module is specified, install at the beginning or end of the module list for this task.
→	theReferenceModuleRefNum	An optional pointer to a reference module.
←	*theNewModuleRefNum	The returned module reference number.
→	theScale	The scale multiplier for scalable sections.

Valid constants for `thePosition` are:

kdspHeadInsert	Insert at head of list.
kdspTailInsert	Insert at tail of list.
kdspBeforeInsert	Insert before reference link.
kdspAfterInsert	Insert after reference link.
kdspAnyPositionInsert	Insert anywhere in list.

DESCRIPTION

DSP object code is stored on disk in the form of a resource. By using the `DSPLoadModule` routine this code can be retrieved from the disk, loaded into the DSP heap, and inserted into the specified task. The `theReferenceModuleRefNum` value can be `nil` if `thePosition` does not require a reference module.

Figure 4-6 is a simplified model of the task and module structure.

Figure 4-6   Task after loading the CD-XA player module
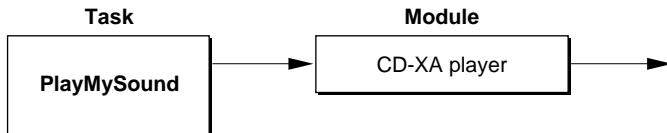
Figure 4-7 shows a closer look at the sections contained within the CD-XA player module.

Figure 4-7   CD-XA player module structure

There are seven sections for this module. The first three are dual buffer sections. There is a buffer in main memory that is used to store the section data between executions, and an on-chip buffer that is used during execution. The code and table sections are load-only, while the variables section is a load and save section. The last four sections are single-buffer sections. They are not loaded or saved. The output buffer, however, must be cleared. This operation is performed by the DSP operating system prior to code execution. The input and output buffers are on-chip, while the FIFO and parameter sections have their buffers off-chip.

Getting Data

The first single buffer section is a FIFO section. It has as its data a FIFO control block called `DSPData`.

Note
Section[3] is a FIFO buffer so you need to call `DSPNewFIFO` to allocate memory for the FIFO buffer. ◆

DSPNewFIFO

The `DSPNewFIFO` routine creates a new FIFO.

```
pascal OSErr DSPNewFIFO (
    DSPSectionRefNum      theSectionRefNum,
    DSPFIFORefNum         *theFIFORefNum,
    unsigned long         theSize,
    Ptr                   logical,
    Ptr                   physical,
    Boolean               fifoFull,
    MessageActionProc     theInterrupt);
```

Field descriptions

→	`theSectionRefNum`	FIFO section reference number.
←	`*theFIFORefNum`	Reference number of the new FIFO.
→	`theSize`	Size of data buffer allocated.
→	`logical`	Logical address.
→	`physical`	Physical address.
→	`fifoFull`	Indication that FIFO is full.
→	`theInterrupt`	Procedure for receiving messages from the FIFO.

DESCRIPTION

The `DSPNewFIFO` routine performs several functions. It creates a new non-linked FIFO if the specified section does not already have a FIFO, or it creates and adds a linked FIFO if the specified section already has a FIFO. The user has the option of passing in a block of memory that already contains data. In this case, the logical address is specified instead of `nil`. A physical address does not need to be specified if the logical address is in the system's main memory (that is, if `GetPhysical` will work).
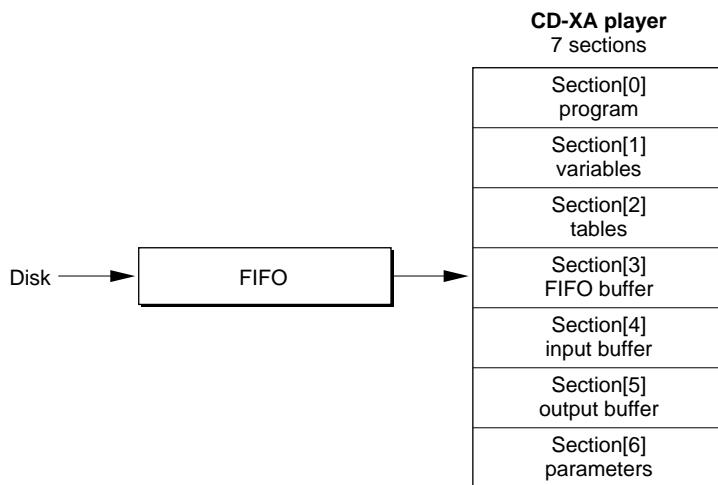
Linked FIFOs are a circular list of FIFO. The list is created before the task is set active and the order of the FIFOs in the list is invariant. The boolean bit `FIFOFull` signifies that the FIFO is full, which is when the read pointer equals the write pointer; it is used only when the program has created the buffer. If there is already data in the buffer pass `true`, otherwise pass `false`.

The `DSPData` control block points to a FIFO buffer set up in system memory by the host application. The `MessageActionProc` routine is provided so the host application will refill the FIFO on request by the Real Time Manager. This is an interrupt-level routine, and is used whenever the FIFO is getting empty. For more information about `MessageActionProc` see "Sending Messages," later in this chapter.

Figure 4-8 shows how `DSPNewFIFO` adds a FIFO buffer to the CD-XA player module in the current example.
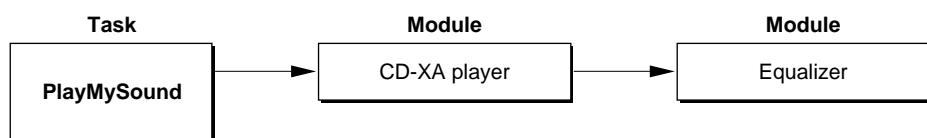
Figure 4-8   Module structure after `DSPNewFIFO` call

CD-XA player
7 sections

Section[0] program
Section[1] variables
Section[2] tables
Section[3] FIFO buffer
Section[4] input buffer
Section[5] output buffer
Section[6] parameters

Disk ⟶ FIFO ⟶

The next two sections are input and output buffers. The last is a parameter section. The parameter section is used to specify the volume of the player's output. It remains off-chip in system memory. The single value stored in that buffer is used to multiply the data before summing into the PRB. Space at the top of the DSP memory is reserved by the Real Time Manager for the PRB. This buffer will be connected to the input buffer of the second module and must always be at the top of available memory to avoid allocation gaps in the DSP memory.

The module reads data from the FIFO into the input AIAO section, decompresses it and places it into the output AIAO. Since this is the first use of this buffer, it must be cleared prior to execution. After the CD-XA player has run, 32-bit floating point sound data will be in the output buffer. In this example equalization is also applied to the sound data so the equalizer module will be loaded after the CD-XA player, as shown in Figure 4-9.

Figure 4-9   Task structure after `DSPLoadModule` call

Task	**Module**	**Module**
PlayMySound	CD-XA player	Equalizer

Now the PlayMySound task has two modules: a CD-XA player and an equalizer. The CD-XA player is executed first, and the equalizer is executed second. Figure 4-10 shows a closer look at the sections contained within the equalizer module.

Figure 4-10 Sections contained in the equalizer module

Equalizer
6 sections

Section[0] program
Section[1] variables
Section[2] tables
Section[3] input buffer
Section[4] output buffer
Section[5] parameters

The output buffer from the CD-XA player module must be connected to the input buffer of the equalizer module so it can process the data from the CD-XA player. This connection eliminates the need to save the buffer off-chip after the decompression module executes only to reload it again for the equalizer module. There are three calls needed to perform the connection:

1. Call `DSPGetSection` to get a pointer to the output buffer section in the CD-XA player module.

2. Call `DSPGetSection` to get a pointer to the input buffer of the equalizer module.

3. Call `DSPConnectSections` to connect these two sections together.

DSPGetSection

The `DSPGetSection` routine returns a section reference number.

```
pascal OSErr DSPGetSection (
    DSPModuleRefNum    theModuleRefNum,
    StringPtr          theSectionName,
    DSPSectionRefNum   *theSectionRefNum);
```

Field descriptions

→	`theModuleRefNum`	Module that contains section.
→	`theSectionName`	Name of the desired section.
←	`*theSectionRefNum`	Returned section reference number.

DESCRIPTION

The `DSPGetSection` routine returns a section reference number when it is passed a module reference number and a section name.

DSPConnectSections

The `DSPConnectSections` routine connects two sections to allow the DSP operating system to leave sections that are shared between modules on-chip.

```
pascal OSErr DSPConnectSections (
    DSPSectionRefNum    sectionOneRefNum,
    DSPSectionRefNum    sectionTwoRefNum,
    DSPConnectionType   theConnectionType);
```

Field descriptions

→	`sectionOneRefNum`	The source section for the connection.
→	`sectionTwoRefNum`	The destination section for the connection.
→	`theConnectionType`	The method of connection.

Valid constants for `theConnectionType` are:

```
kdspDirectConnection
kdspIndirectConnection
kdspHIHOConnection
```
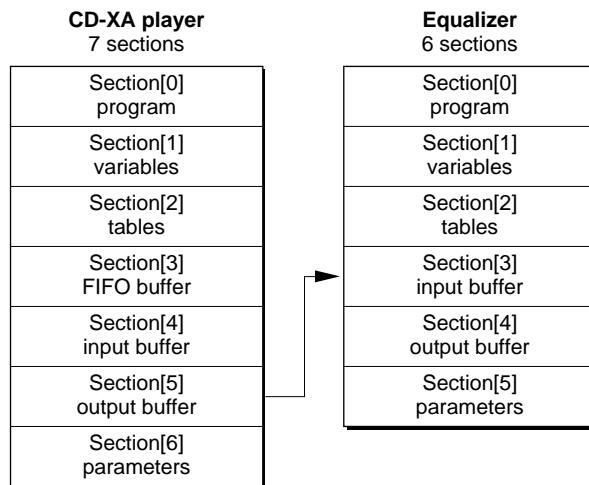
DESCRIPTION

`DSPConnectSections` is used to pass data between two sections in the same task or between a section and an ITB. When setting up the connection, data flow is the controlling factor and should always run from section one to section two. The data type for both sections must also match for the connection to be established.

When a FIFO section is connected to a FIFO buffer using `DSPNewFIFO`, the connection process stores the information about the FIFO in the `DSPFIFO` header. When a FIFO section of one module is connected to an AIAO section in another module the connection process changes the AIAO into a FIFO and creates the `DSPFIFO` header. This change does not affect the operation of the AIAO section within the module. It only facilitates the connection of different section types so data can be transferred between them between modules. Figure 4-11 shows the sample CD-XA player connected to an equalizer, using `DSPConnectSections`.

With this connection established the data from the output buffer of the CD-XA player module will be fed to the input buffer of the equalizer module.

Figure 4-11 CD-XA player with `DSPConnectSections` to equalizer

Putting the Task to Work

Now that the task structure is complete you use `DSPInsertTask` to insert the task into the DSP execution stream and `DSPSetTaskActive` to start the DSP executing it.

DSPInsertTask

The `DSPInsertTask` routine inserts the specified task into the specified task list.

```
pascal OSErr DSPInsertTask (
    DSPCPUDeviceParamBlkPtr    theCPUDevice,
    DSPTaskRefNum              theNewTaskRefNum,
    DSPPosition                thePosition,
    DSPTaskPriority            interruptLevel,
    DSPTaskRefNum              theRelativeTaskRefNum);
```

Field descriptions

→	`theCPUDevice`	The CPU device being used.
→	`theNewTaskRefNum`	Reference number of the task to insert.
→	`thePosition`	Insertion location.
→	`interruptLevel`	`kdspRealTime` or `kdspTimeShare`.
→	`theRelativeTaskRefNum`	Reference number of a task previously installed by this client. If this client has already inserted a task into the task list it may be used as a reference for inserting other tasks.

DESCRIPTION

The `DSPInsertTask` routine is used to insert `theNewTaskRefNum` in the list of tasks for a given device. When the task is inserted, the Real Time Manager performs the calculations necessary to determine how much DSP processing the task will take.

When inserting tasks it is necessary to specify where in the list to insert the task. This is specified in the `thePosition` parameter, as shown in Table 4-4.

Table 4-4   Task insertion locations

Constant	Comments
kdspHeadInsert	Insert at head of list
kdspTailInsert	Insert at tail of list
kdspBeforeInsert	Insert before reference task
kdspAfterInsert	Insert after reference task
kdspAnyPositionInsert	Insert anywhere in list

To start and stop the task call `DSPSetTaskActive` and `DSPSetTaskInactive` respectively. This will leave the task allocated and installed in the DSP's execution list.

DSPSetTaskActive

The `DSPSetTaskActive` routine sets the specified task active.

```
pascal OSErr DSPSetTaskActive (DSPTaskRefNum theRefNum);
```

Field description

→   theRefNum   Reference number of the DSP task to activate.

DESCRIPTION

To activate a task call `DSPSetTaskActive`. After a task has been installed, the DSP will not execute the task until it has been activated. This routine sets the active flag for the task, which tells the DSP operating system to execute it on the next pass through the task list.

Getting Off the DSP Task List

When you are ready to stop and remove a task from the task list, use the sequence of Real Time Manager calls in Table 4-5. Italicized calls are used only if necessary.

Table 4-5   Removing a task

Routines	Usage
DSPSetTaskInactive	Stop the task
DSPRemoveTask	Remove the task from the DSP's execution stream
DSPUnloadModule	Remove the module from the task structure
DSPDisposeFIFO	Free memory used by a FIFO section
DSPDisposeTask	Free memory used by the task structure
DSPCloseCPUDevice	Sign out from the DSP device

DSPSetTaskInactive

The DSPSetTaskInactive routine sets the specified task inactive.

```
pascal OSErr DSPSetTaskInactive (DSPTaskRefNum theRefNum);
```

Field description

→   theRefNum   Reference number of the DSP task to deactivate.

DESCRIPTION

To deactivate a task call DSPSetTaskInactive. This routine does not remove the task from the task list or deallocate memory.

DSPRemoveTask

The DSPRemoveTask routine removes the specified task from the task list.

```
pascal OSErr DSPRemoveTask (DSPTaskRefNum    theTaskRefNum);
```

Field description

→   theTaskRefNum   Reference number of the DSP task to remove.

DESCRIPTION

The `DSPRemoveTask` routine is used to remove a task from the task list of a specific device. If the task is still active at the time this call is made, `DSPRemoveTask` will deactivate the task before removing it from the list.

Note

It may take several frames before the Real Time Manager can correctly remove the task from the DSP. The `DSPRemoveTask` routine will call `WaitNextEvent` until the task goes inactive. ◆

DSPUnloadModule

The `DSPUnloadModule` routine removes the specified module from the task.

```pascal
pascal OSErr DSPUnloadModule (
    DSPTaskRefNum        theTaskRefNum,
    DSPModuleRefNum      theModuleRefNum);
```

Field descriptions

→   theTaskRefNum   The task that contains the module.
→   theModuleRefNum   The module to unload.

DESCRIPTION

The `DSPUnloadModule` routine is used to remove a module from a task. The task must first be removed from the task list using `DSPRemoveTask` before the module can be unloaded. To remove all the modules at once, use the `DSPDisposeTask` routine, described on page 151. This will dispose of the task structure as well as all of the module structures in the task module list.

DSPDisposeFIFO

The `DSPDisposeFIFO` routine disposes of the memory used by the specified FIFO.

```pascal
pascal OSErr DSPDisposeFIFO (
    DSPSectionRefNum        theSectionRefNum,
    DSPFIFORefNum           theFIFORefNum);
```

Field descriptions

→   theSectionRefNum   The section reference number.
→   theFIFORefNum   The FIFO to be disposed of.

DESCRIPTION

> The `DSPDisposeFIFO` routine is used to dispose of the memory used by a single `DSPFIFO` in a set of linked FIFOs, a non-linked FIFO, or all of the linked FIFOs belonging to a section. The FIFO is indicated by both the `theFIFORefNum` and the `theSectionRefNum` values. If the `theFIFORefNum` value is `nil`, all of the linked FIFOs are disposed of; otherwise, only the specified FIFO is disposed of. The memory used by the `DSPFIFO` data structure is deallocated when the module that owns the FIFO section is deallocated by `DSPUnloadModule`.

DSPDisposeTask

> The `DSPDisposeTask` routine disposes the memory of the specified task.

> `pascal OSErr DSPDisposeTask (DSPTaskRefNum theTaskRefNum);`

> **Field description**
> →   `theTaskRefNum`   Reference number of the task to dispose of.

DESCRIPTION

> The `DSPDisposeTask` routine is used to deallocate the `theTask` structure that was allocated using the `DSPNewTask` routine. It deallocates all of the DSP modules, DSP sections, and DSP FIFOs associated with the specified task. If the task is active `DSPSetTaskInactive` will be called before the structures are deallocated. Until the task is inactive `DSPDisposeTask` will call `GetWaitNextEvent`.

DSPCloseCPUDevice

> The `DSPCloseCPUDevice` routine closes access to the specified device.

> `pascal OSErr DSPCloseCPUDevice (DSPCPUDeviceParamBlkPtr;`
>   `theCPUParamBlk);`

> **Field description**
> →   `pbhClientRefNum`   The reference number for this client.

DESCRIPTION

> The `DSPCloseCPUDevice` routine closes a device that was requested by a `DSPOpenCPUDevice` routine.

> The fields of the `DSPCPUDeviceParamBlk` structure are described in the `DSPGetIndexedCPUDevice` section, page 138.

DSPUpdateCPUDeviceInfo

The `DSPUpdateCPUDeviceInfo` routine updates the information about a CPU device.

```
pascal OSErr DSPUpdateCPUDeviceInfo (DSPCPUDeviceParamBlkPtr
    theCPUParamBlk);
```

Field descriptions

→	pbhDeviceIndex	The index for this device.
←	cpuAllocatedCycles	The allocated real-time cycles.
←	cpuCurRealTimeLoading	The number of real-time cycles used.
←	cpuTimeShareLoading	The timeshare loading for the device.

DESCRIPTION

The `DSPUpdateCPUDeviceInfo` routine is called by a Real Time Manager client to update the information about a CPU device. To use this call, the client must first call `DSPOpenCPUDevice` to open the device. Since the loading values for a CPU device will change as clients use or relinquish the device, this call will let a client get the current loading numbers.

Sending Messages
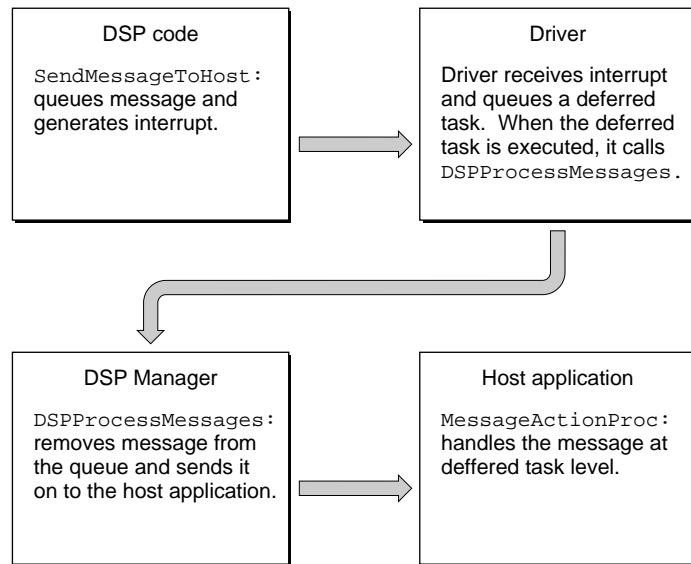
This section describes how real-time data processing software passes information between the currently running Macintosh software (the **host**) and the DSP program.

From DSP to Host

In many situations the DSP will need to notify a host application when a change in status has been detected. In order to free the host from the burden of polling the DSP for changes in status, an interrupt driven DSP–to–host message passing mechanism is provided. This mechanism is used by the Real Time Manager both for sending update messages to the DSP FIFOs, and for sending frame overrun messages to clients. It is generic however, and can be used by any client that needs this capability.

To send a message from the DSP to the host, the code calls `_SendMessageToHost` with a pointer to the message to be sent. The `_SendMessageToHost` routine queues the message and generates an interrupt to the host. When the host receives the interrupt, it queues a deferred task. When the deferred task is removed from the queue and executed, it calls `DSPProcessMessages`, which calls the actual `MessageActionProc` routine. This process is diagrammed in Figure 4-12.

Figure 4-12   Message passing from DSP to host

The reason for deferring the interrupt via the deferred task mechanism is to ensure compatibility with virtual memory. By calling the user's `MessageActionProc` routine from the deferred task, the user code does not have to be locked in physical memory while the risk of causing a double page fault is still avoided. A disadvantage of using the deferred task mechanism is that the messages only get processed after all higher level interrupts have been processed. You should consider this when determining if the message passing mechanism is appropriate for your use.

From Host to DSP

Message passing from the host to the DSP is accomplished by using shared memory. A parameter, data, or table section must be established for use by both the DSP and the host. The host uses this section to pass user-defined messages to the DSP. The DSP must check this section for new information in every frame.

Message Action Procedure

All normal interrupt code restrictions apply to the `MessageActionProc` routine. Register A5 is not guaranteed to be valid and should be explicitly restored if needed. Only those toolbox routines that are re-entrant can be called. The `MessageActionProc` routine should reside in the main segment of the application to ensure that its segment does not get unloaded. The `MessageActionProc` routine has the format shown in Listing 4-8.

Listing 4-8   Message action procedure

```
typedef pascal void (*MessageActionProc)(struct DSPMessage
                        *theMessage);
struct DSPMessage {
   MessageActionProc    msVector;    // vector of routine
   unsigned long        msData[3];  // application-specific data
   };
```

Message Format

The `msVector` field of the `DSPMessage` data structure is always a pointer to the `MessageActionProc` routine that will be called by `DSPProcessMessages` when the message is removed from the deferred task queue. It should be set up by the host at run time prior to executing the DSP code that uses it. If the `DSPProcessMessages` receives a message with a `nil` `msVector` field the message will be ignored. The use of `msData` is application-specific. It provides a convenient place for the DSP sending the message to store parameters the `MessageActionProc` routine will need. The `DSPMessage` data structure has the format shown in Listing 4-9.

Listing 4-9   DSP message format

```
struct DSPMessage {
   MessageActionProc msVector;    // vector of routine
   unsigned long     msData[3];  // application-specific data
   };

DSPTask MessageActionProc
msVector    -> passed in during DSPNewTask routine
msData[0]   -> the message
msData[1]   -> DSPTask
msData[2]   -> application-specific parameter set using the
               DSPSetTaskRefCon routine

DSPFIFO MessageActionProc
msVector    -> passed in during DSPNewFIFO routine
msData[0]   -> the message
msData[1]   -> DSPFIFORefNum
msData[2]   -> application-specific parameter set using the
               DSPFIFOSetRefCon routine
```

Messages that are valid for the `msData[0]` field are listed in "Summary of the Real Time Manager," at the end of this chapter.

Real Time Manager Reference

The remainder of this chapter describes the routines in the Real Time Manager that are not described earlier in this chapter. At the end of this chapter is a summary listing of the DSP's application programming interface.

Client Routines

The client routines support software access to the DSP. Besides the routines described in this section, the client routines include the following:

■ `Gestalt`, described on page 136

■ `DSPManagerVersion`, described on page 137

DSPGetIndexedClient

The `DSPGetIndexedClient` routine gets an index to the specified client.

```
Pascal OSErr DSPGetIndexedClient (
    unsigned long              theIndex,
    DSPCPUDeviceParamBlkPtr    theCPUDevice,
    DSPClientRefNum            *theClientRefNum);
```

Field descriptions

→	`theIndex`	Index of desired client.
←	`*theClientRefNum`	Returned client reference numbers.

`DSPCPUDeviceParamBlkPtr`

Field descriptions

→	`theCPUDevice`	Name of the CPU device.
↔	`theClientICON`	Return the client's icon.

DESCRIPTION

The `DSPGetIndexedClient` routine is used to find all the clients signed into the DSP device specified by `theCPUDevice`. The client should pass an index starting at 0 (in `theIndex`) and increment it until a `kdspInvalidIndexErr` value is returned. The client should also allocate space for the icon buffer `theClientICON` or pass `nil` if no icon is desired. This routine does not allocate memory.

Note
If the icon is not desired, be sure to set `theClientICON` to
`nil` before calling `DSPGetIndexedClient`. ◆

DSPGetClientInfo

The `DSPGetClientInfo` routine gets information about the specified client.

```
Pascal OSErr DSPGetClientInfo
    (DSPClientInfoParamBlkHandletheClientParamBlkHdl);
```

Field descriptions

→	`ciClientRefNum`	Input reference number of the client.
←	`ciClientPermission`	Returned read/write permission of the client.
↔	`ciClientICON`	Returned icon of the client.
←	`ciClientName`	Returned name of the client.

DESCRIPTION

The `DSPGetClientInfo` routine is used to find information about a specific DSP client.
Use either the `DSPGetIndexedClient` or the `DSPGetOwnerClient` routine to find the
`ciClientRefNum` value needed in the `DSPGetClientInfo` call.

Note
If the specified client did not provide an icon then `nil` will be returned
in the `ciClientICON` field. ◆

DSPGetOwnerClient

The `DSPGetOwnerClient` routine gets the client for a specified task.

```
Pascal OSErr DSPGetOwnerClient (
    DSPTaskRefNum       theTaskRefNum,
    DSPClientRefNum    *theClientRefNum);
```

Field descriptions

→	`theTaskRefNum`	Reference number for this task.
←	`*theClientRefNum`	Reference number for this client.

DESCRIPTION

When given the `DSPTaskRefNum` value, `DSPGetOwnerClient` returns the reference
number, in `theClientRefNum`, of the client that created the specified task.

Device Routines

The device routines help you work with CPU devices. Besides the routines described in this section, they include the following:

■ `DSPGetIndexedCPUDevice`, described on page 138

■ `DSPOpenCPUDevice`, described on page 139

■ `DSPCloseCPUDevice`, described on page 151

DSPGetAvailableOnChipMemory

The `DSPGetAvailableOnChipMemory` routine gets available on-chip memory on the specified CPU device.

```
pascal OSErr DSPGetAvailableOnChipMemory (
    DSPModuleRefNum      theModuleRefNum,
    unsigned long        *theSize);
```

Field descriptions

→	`theModuleRefNum`	The reference number of the desired module.
←	`*theSize`	The size of the available on-chip memory.

DESCRIPTION

The `DSPGetAvailableOnChipMemory` routine returns the amount of on-chip DSP memory available for use by the module. The returned value can be used by the `DSPSetSectionSize` routine to set a dynamic section in a demand cache module to this size.

SEE ALSO

`DSPGetSectionInfo` (page 174)
`DSPSetSectionSize` (page 174)
`DSPNewInterTaskBuffer` (page 175
`DSPNewFIFO` (page 176)
`DSPFIFOGetSize` (page 176)

DSPGetIndexedCPUDeviceOption

The `DSPGetIndexedCPUDeviceOption` routine gets information about a specified CPU device. To get all the available options of a given type, repeat this call while incrementing `theOptionIndex` from 0 until it returns `kdspInvalidIndexErr`.

You can then call DSPSetIndexedCPUDeviceOption with the resulting values of theOptionType and theOptionIndex to set the device option.

```
pascal OSErr DSPGetIndexedCPUDeviceOption (
    DSPCPUDeviceParamBlkPtr    theCPUParamBlk,
    unsigned short             theOptionIndex,
    OSType                     theOptionType,
    unsigned long              *theOption);
```

Field descriptions

→	theOptionIndex	The index of the desired option from 0.
→	theOptionType	The desired option.
←	*theOption	The value of the specified option.

The valid selector for theOptionType is:

kdspFrameRate

DSPCPUDeviceParamBlkPtr

Field descriptions

→	pbhClientRefNum	The client reference number.
→	pbhClientPermission	The caller's permission.

DESCRIPTION

The DSPGetIndexedCPUDeviceOption routine allows a client to request information about a device. By calling DSPGetIndexedCPUDeviceOption with theOptionIndex starting at 0 and incrementing until kdspInvalidIndexErr is returned a client can get a list of all available options of a given type for a device. The client must first make a call to DSPOpenCPUDevice before calling DSPGetIndexedCPUDeviceOption.

DSPSetIndexedCPUDeviceOption

The DSPSetIndexedCPUDeviceOption routine sets information about specified CPU device. For information about getting values for theOptionIndex and theOptionType, see the routine description for DSPGetIndexedCPUDeviceOption on page 157.

```
pascal OSErr DSPSetIndexedCPUDeviceOption (
    DSPCPUDeviceParamBlkPtr    theCPUParamBlk,
    unsigned short             theOptionIndex,
    OSType                     theOptionType);
```

Field descriptions

→	theOptionIndex	The index of the desired option.
→	theOptionType	The desired option.

DSPCPUDeviceParamBlkPtr

Field descriptions

→	pbhClientRefNum	The client reference number.
→	pbhClientPermission	The caller's permission.

DESCRIPTION

The `DSPSetIndexedCPUDeviceOption` routine is used to set an option for a CPU device. Set `theOptionType` to the appropriate selector and `theOptionIndex` to the desired option. The client must first call `DSPOpenCPUDevice` with read/write permission to be able to change the device option.

The `theCPUParamBlk` data structure is used to pass information about a DSP CPU device between the Real Time Manager and a client.

DSPSetCPUDeviceBondage

The `DSPSetCPUDeviceBondage` routine configures the master/slave relationship between two DSP CPUs.

```pascal
pascal OSErr DSPSetCPUDeviceBondage (
    DSPCPUDeviceParamBlkPtr    cpuDevice1,
    DSPCPUDeviceParamBlkPtr    cpuDevice2);
```

Field descriptions

→	cpuDevice1	The device to be configured.
→	cpuDevice2	Optional, new master device.

DESCRIPTION

The `DSPSetCPUDeviceBondage` routine is used to configure the master/slave relationship between two DSP CPUs. Parameter `cpuDevice1` identifies the device to be configured. Pass `nil` in `cpuDevice2` to make `cpuDevice1` identify a master device, or pass a valid `cpuDevice2` value to make `cpuDevice1` a slave of `cpuDevice2`.

DSPOpenIODevice

The `DSPOpenIODevice` routine opens access to a specified I/O device.

```
pascal OSErr DSPOpenIODevice
    (DSPIODeviceParamBlkPtr    theIOParamBlk);
```

Field descriptions

→	`pbhDeviceIndex`	The index for this device.
→	`iopbCPUDeviceIndex`	CPU device index for this I/O device.
→	`iopbIODeviceType`	The type of this I/O device.
→	`pbhDeviceName`	Returned name of device.
→	`pbhClientPermission`	The caller's permission.
←	`pbhClientRefNum`	The client reference number.

Valid constants for `pbhClientPermission` are:

`kdspWritePermission`	Requesting permission to write to the device.
`kdspReadPermission`	Requesting permission to read from the device.
`kdspReadWritePermission`	(`kdspWritePermission | kdspReadPermission`).

DESCRIPTION

The `DSPOpenIODevice` routine allows the client to gain access to one of the `DSPIODevice` drivers. This routine is analogous to `DSPOpenCPUDevice` in that if the `DSPIODeviceParamBlk` structure was set up by calling `DSPGetIndexedIODevice` then the client need only fill the `pbhClientPermission` field.

DSPCloseIODevice

The `DSPCloseIODevice` routine closes access to the specified I/O device.

```
pascal OSErr DSPCloseIODevice (DSPIODeviceParamBlkPtr;
    theIOParamBlk);
```

Field descriptions

→	`pbhClientRefNum`	The reference number for this client.
→	`pbhClientPermission`	The read/write permission for this client.

DESCRIPTION

The `DSPCloseIODevice` routine closes a `DSPIODevice` that was opened using `DSPOpenIODevice`.

DSPGetIndexedIODevice

The `DSPGetIndexedIODevice` routine gets an index to the specified I/O device.

```
pascal OSErr DSPGetIndexedIODevice (DSPIODeviceParamBlkPtr;
    theIOParamBlk)
```

Field descriptions

→	pbhDeviceIndex	The index for this device.
→	iopbCPUDeviceIndex	CPU device index for this I/O device.
→	iopbIODeviceType	The type of this I/O device.
←	pbhDeviceName	Returned name of device.
↔	pbhDeviceICON	Returned icon of device.

Valid I/O device types are:

```
kdspSIOTypeDevice    (0)
kdspBIOTypeDevice    (1)
```

DESCRIPTION

The `DSPGetIndexedIODevice` routine is used by a client to find information about the DSP I/O devices installed. By calling `DSPGetIndexedIODevice` a client can create a list containing `theIOParamBlk` for each available device. The client should pass an index starting at 0 (in `iopbCPUDeviceIndex`) and increment it until `kdspInvalidIndexErr` is returned. The client should also allocate space for the icon buffer `pbhDeviceICON` or pass `nil` if no icon is desired. This routine does not allocate memory.

Note
If the icon is not desired, be sure to set `pbhDeviceICON` to `nil` before calling `DSPGetIndexedCPUDevice`. ◆

DSPGetIndexedIODeviceOption

The `DSPGetIndexedIODeviceOption` routine gets information about a specified I/O device. To get all the available options of a given type, repeat this call while incrementing `theOptionIndex` from 0 until it returns `kdspInvalidIndexErr`. You can then call `DSPSetIndexedIODeviceOption` with the resulting values of `theOptionType` and `theOptionIndex` to set the device option.

```
pascal OSErr DSPGetIndexedIODeviceOption (
    DSPIODeviceParamBlkPtr    theIOParamBlk,
    unsigned short            theOptionIndex,
    OSType                    theOptionType,
    unsigned long             *theOption);
```

Field descriptions

→	theOptionIndex	The index of the desired option from 0.
→	theOptionType	The desired option.
←	theOption	The value of the specified option.

Valid values for theOptionType are:

kdspIndexedSampleRate   'srop'

kdspIndexedSampleFormat   'szop'

DSPIODeviceParamBlkPtr

Field descriptions

→	pbhClientRefNum	The reference number of the client.
→	pbhClientPermission	The caller's permission.

DESCRIPTION

A client can find out the different values available for each theOptionType by calling DSPGetIndexedIODeviceOption. For each theOptionType the client should pass a count starting at 0 (in theOptionIndex) and increment it until a kdspInvalidIndexErr error occurs. Using this method the client can get a list of all available options for each theOptionType for a DSP I/O device.

DSPSetIndexedIODeviceOption

The DSPSetIndexedIODeviceOption routine sets information about a specified I/O device. For information about getting values for theOptionIndex and theOptionType, see the routine description for DSPGetIndexedIODeviceOption on page 161.

```
pascal OSErr DSPSetIndexedIODeviceOption (
    DSPIODeviceParamBlkPtr      theIOParamBlk,
    unsigned short              theOptionIndex,
    OSType                      theOptionType);
```

Field descriptions

→	theOptionIndex	The index of the desired option from 0.
→	theOptionType	The desired option.

Valid values for theOptionType are:

kdspIndexedSampleRate   'srop'
kdspIndexedSampleFormat   'szop'

DSPIODeviceParamBlkPtr

Field descriptions

→	pbhClientRefNum	The client reference number.
→	pbhClientPermission	The caller's permission.

The `DSPSetIndexedIODeviceOption` routine allows the client to set an option for a DSP I/O device. The client must first open the device with read/write privileges. Use `DSPGetIndexedIODeviceOption` to determine the relationship of `theOptionIndex` to each `theOptionType`. For example, to set `kdspIndexedSampleRate` to a specific sample rate, use the value of `theOptionIndex` that returned the value you wanted.

Note
In all cases only the client with write permission is allowed to make `DSPSetIndexedIODeviceOption` calls. ◆

The `theIOParamBlk` data structure is used to pass information about a DSP I/O device between the Real Time Manager and a client.

Task API Routines

The task API routines let you create and control DSP tasks. Besides the routines described in this section, the task API routines include the following:

- `DSPNewTask`, described on page 140

- `DSPInsertTask`, described on page 147

- `DSPSetTaskActive`, described on page 148

- `DSPSetTaskInactive`, described on page 149

- `DSPRemoveTask`, described on page 149

- `DSPDisposeTask`, described on page 151

DSPGetIndexedTask

The `DSPGetIndexedTask` routine returns a reference number to the task in the specified task list on the specified CPU belonging to the indexed client.

```
pascal OSErr DSPGetIndexedTask (
    unsigned long               theIndex,
    DSPTaskPriority             interruptLevel,
    DSPCPUDeviceParamBlkPtr     theCPUDevice,
    DSPTaskRefNum               *theTaskRefNum);
```

The `DSPTaskPriority` constants can be:

```
kdspRealTime
kdspTimeShare
```

Field descriptions

→	`theIndex`	Index of desired client.
→	`interruptLevel`	`kdspRealTime` or `kdspTimeShare`.
→	`theCPUDevice`	The `ParamBlkPtr` of the device to search.
←	`theTaskRefNum`	Returned task reference number.

DESCRIPTION

`DSPGetIndexedTask` is used by a client to find information about all the DSP tasks currently installed on a specific CPU device. By calling `DSPGetIndexedTask` a client can create a list containing the clients for each available device. The client should pass an index starting at 0 (in `theIndex`) and increment it until a `kdspInvalidIndexErr` value is returned.

DSPGetTaskStatus

The `DSPGetTaskStatus` routine returns the status of the referenced task.

```
pascal unsigned long DSPGetTaskStatus (DSPTaskRefNum theTask);
```

Field description

→	`theTask`	Reference number of the task.

The task status is one of the following constants:

```
kdspTaskIsActive
kdspTaskIsInactive
kdspTaskIsGoingInactive
kdspTaskIsGoingActive
kdspInvalidTask
```

DESCRIPTION

The `DSPGetTaskStatus` routine returns the status of a task.

DSPGetOwnerTask

The `DSPGetOwnerTask` routine returns the reference number to the task that the module belongs to.

```
pascal OSErr DSPGetOwnerTask (
    DSPModuleRefNum        theModuleRefNum,
    DSPTaskRefNum          *theTaskRefNum);
```

Field descriptions

→   `theModuleRefNum`   Reference number of the module.
←   `theTaskRefNum`   Reference number of the task.

DESCRIPTION

The `DSPGetOwnerTask` routine returns the `theTaskRefNum` reference number of the task that contains the specified DSP module.

DSPGetTaskInfo

The `DSPGetTaskInfo` routine returns information about the referenced task.

```
pascal OSErr DSPGetTaskInfo (DSPTaskInfoParamBlkHandle
    theTaskParamBlkHdl;);
```

Field descriptions

→   `tiRefNum`   Reference number of the task.
←   `tiRefCon`   Returned application-specific information.
←   `tiVector`   Returned vector of task action procedure.
←   `tiFlags`   Returned flags for DSP task control.
←   `tiName`   Returned name of this DSP task.

DESCRIPTION

Given a task reference number, a client can use the `DSPGetTaskInfo` routine to find out the task's `RefCon` information (the application-specific field of the `DSPTask` structure), the task action procedure vector, the task flags, and the task name. The task reference number can be obtained by using either `DSPGetIndexedTask` (page 163) or `DSPGetOwnerTask` (page 165).

DSPGetTaskRefCon

The `DSPGetTaskRefCon` routine returns application-specific information about the referenced task.

```
pascal unsigned long DSPGetTaskRefCon (DSPTaskRefNum theRefNum);
```

Field description
→ theRefNum   Reference number of the task.

DESCRIPTION

The `DSPGetTaskRefCon` routine is used to return the application-specific field of the `DSPTask` structure.

DSPSetTaskRefCon

The `DSPSetTaskRefCon` routine sets the application-specific information field for the specified task.

```
pascal OSErr DSPSetTaskRefCon (
    DSPTaskRefNum     theRefNum,
    unsigned long     theData);
```

Field descriptions
→ theRefNum   Reference number of the DSP task.
→ theData   New application-specific data for the task.

DESCRIPTION

The `DSPSetTaskRefCon` routine is used to set the application-specific data field in the `DSPTask` structure. An application will use this routine to set the parameter when it passes a message to the task's `MessageActionProc`. The `DSPSetTaskRefCon` routine can then be used from inside the task `MessageActionProc` to retrieve the parameter.

For more information about `MessageActionProc` see "Sending Messages," earlier in this chapter.

DSPTaskToSynchronize

The `DSPTaskToSynchronize` routine sets a flag in the task to indicate that this task should be synchronized.

```
pascal OSErr DSPTaskToSynchronize (
    DSPTaskRefNum      theRefNum,
    unsigned long      frameDelay,
    DSPSynchRefNum     *synchRefNum);
```

Field descriptions

→   theRefNum   Reference number of the DSP task to synchronize.
→   frameDelay   Number of frames till synch operation.
→   synchRefNum   Synchronization reference number for this task.

DESCRIPTION

You sometimes need to synchronize multiple tasks so they start on the same frame. For example, when playing back a multiple track sound file where each of the tracks is assigned to a separate task, it is necessary to start each of these tracks on the same frame to maintain synchronization. Of course there are many cases where the same function can be accomplished by placing the required modules under one task; this is the preferred method.

When installing several tasks that you want to synchronize, use the `DSPTaskToSynchronize` routine to set them all to the same `synchRefNum`. Once this flag is set in all of the tasks that you want to start synchronously, a call is made to `DSPSynchronizeTasks`. This causes the DSP operating system to toggle (change the state of the flag, 1 or 0) all of the active flags for the marked tasks.

This technique can synchronously stop as well as synchronously start tasks. You can even start one group and stop another group at the same time with this technique.

Note
Until `DSPSynchronizeTasks` is called, no calls to `WaitNextEvent` should be made. ◆

DSPSynchronizeTasks

The `DSPSynchronizeTasks` routine synchronizes the starting of different tasks.

```
pascal OSErr DSPSynchronizeTasks (DSPSynchRefNum  *synchRefNum)
```

Field description

→   synchRefNum   Synchronization reference number of DSP task to be synchronized.

DESCRIPTION

This routine is used when it is necessary to synchronize two or more DSP tasks that belong to the same or different DSP devices.

▲ **WARNING**
This call should not be made from the interrupt level. ▲

Note
It does not make any sense to try to synchronize both real-time and timeshare tasks. There is no way to guarantee any relationship between the two task lists. ◆

SEE ALSO

DSPTaskToSynchronize (page 167)

Module API Routines

The module API routines let you load and control DSP modules. Besides the routines described in this section, the module API routines include the following:

■ DSPLoadModule, described on page 141

■ DSPUnloadModule, described on page 150

DSPGetOwnerModule

The DSPGetOwnerModule routine returns a reference number to the module for the specified section.

```pascal
pascal OSErr DSPGetOwnerModule (
    DSPSectionRefNum      theSectionRefNum,
    DSPModuleRefNum       *theModuleRefNum);
```

Field descriptions

→   theSectionRefNum   The section reference number.
←   theModuleRefNum   Returned reference number of module.

DESCRIPTION

The DSPGetOwnerModule routine is used by a client to find the reference number of the module that the specified section is a part of.

DSPGetIndexedModule

The `DSPGetIndexedModule` routine returns a reference number to the specified module on the specified task belonging to the indexed client.

```pascal
pascal OSErr DSPGetIndexedModule (
    unsigned long      theIndex
    DSPTaskRefNum      theTaskRefNum
    DSPModuleRefNum   *theModuleRefNum);
```

Field descriptions

→	theIndex	Index of desired client.
→	theTaskRefNum	Reference number of the task.
←	theModuleRefNum	Returned module reference number.

DESCRIPTION

The `DSPGetIndexedModule` routine is used by a client to find information about all the modules that are currently installed on a specific task. By calling this routine, a client can create a list containing the modules for each available task. The client should pass an index starting at 0 (in `theIndex`) and increment it until a `kdspInvalidIndexErr` value is returned.

DSPGetModuleInfo

The `DSPGetModuleInfo` routine provides information about the specified module.

```pascal
pascal OSErr DSPGetModuleInfo (DSPModuleInfoParamBlkHandle
    theModuleParamBlkHdl);
```

Field descriptions

→	miRefNum	Input reference number of this module.
←	miGPB	Returned guaranteed processing bandwidth.
←	miFlags	Returned module flags.
←	miNumSections	Returned number of sections in module.
←	miName	Returned name of the DSP module.
←	miExecutions	Returned number of executions.
←	miSkipCount	Returned number of modules to skip.

DESCRIPTION

The `DSPGetModuleInfo` routine is used by a client to find information about the module specified by `miRefNum`.

DSPSetSkipCount

The `DSPSetSkipCount` routine sets the skip count for the specified module.

```pascal
pascal OSErr DSPSetSkipCount (
    DSPModuleRefNum        theModuleRefNum,
    unsigned long          theCount);
```

Field descriptions

→	`theModuleRefNum`	The reference number of this module.
→	`theCount`	The new skip count value.

DESCRIPTION

The `DSPSetSkipCount` routine sets the skip count value of the module referenced by `DSPModuleRefNum`. It is used to selectively skip over one or more modules, to run later modules in the same task.

DSPSetGPBMode

The `DSPSetGPBMode` routine sets the GPB mode for the specified module.

```pascal
pascal OSErr DSPSetGPBMode (
    DSPModuleRefNum          theModuleRefNum,
    unsigned short           desiredMode)
```

Field descriptions

→	`theModuleRefNum`	The reference number of this module.
→	`desiredMode`	The new mode value for the GPB vector in the module.

`DSPGPBModeIndicator`

Field descriptions

→	`miCurrentMode`	Mode in which the module is operating (only the DSP code should change this value).
→	`miNextMode`	Mode the client wants to change to (only the Real Time Manager should change this value).

DESCRIPTION

The `DSPSetGPBMode` routine is used to change the GPB allocation for a module that has already been installed in the DSP. If the module has more than one scaling vector (sample-rate/scale-factor/GPB-value) then each sample-rate/scale-factor with a different GPB value is a GPB mode.

If there is enough processing bandwidth available the Real Time Manager will change `miNextMode` to the desired mode. The DSP module should be monitoring the `miNextMode` value for changes and switch modes to the desired mode. After the module has made the switch it should update the `miCurrentMode` value so the Real Time Manager can update the GPB numbers on the disk. If there is not enough processing bandwidth to use the new mode the Real Time Manager will respond with an error, `kdspNotEnoughGPB`.

SEE ALSO

DSPLoadModule (page 141)

DSPCountModule

The `DSPCountModule` routine allows the specified module to be included in the GPB calculations.

```
pascal OSErr DSPCountModule (DSPModuleRefNum theModuleRefNum)
```

Field description

→   theModuleRefNum   Module reference number.

DESCRIPTION

The `DSPCountModule` routine clears the `kdspDontCountThisModule` flag so the GPB value of the module is used in calculating the task's GPB requirements. There must be at least one module in a task that has this flag cleared. The `kdspDontCountThisModule` flag is normally cleared by the DSP programmer.

DSPDontCountModule

This routine prevents the specified module from being included in the GPB calculations.

```
pascal OSErr DSPDontCountModule (DSPModuleRefNum theModuleRefNum)
```

Field description

→   theModuleRefNum   Module reference number.

DESCRIPTION

Sets the `kdspDontCountThisModule` flag so the GPB value of the module is not used in calculating the task's GPB requirements. There must be at least one module in a task that has this flag cleared. The `kdspDontCountThisModule` flag is normally cleared by the DSP program.

DSPUpdateGPBPreferenceFile

The `DSPUpdateGPBPreferenceFile` routine updates the GPB information for the specified module in the DSP preferences file.

```
pascal OSErr DSPUpdateGPBPreferenceFile (DSPModuleRefNum
    theModuleRefNum)
```

Field description

$\rightarrow$ theModuleRefNum   Reference number of this module.

DESCRIPTION

The `DSPUpdateGPBPreferenceFile` routine immediately updates the preferences file, but only if the worst-case GPB condition has been found.

DSPDontUpdateGPBPrefs

The `DSPDontUpdateGPBPrefs` routine prevents the specified module from having its actual GPB values saved to the DSP preferences file.

```
pascal OSErr DSPDontUpdateGPBPrefs (DSPModuleRefNum
    theModuleRefNum)
```

Field description

$\rightarrow$ theModuleRefNum   Reference number of this module.

DESCRIPTION

When a module is deallocated, if the module has run its worst-case GPB condition and the `bnActual` GPB flag is set, then the Real Time Manager will automatically save the module's new GPB number(s) to the DSP preferences file. This routine is used when the application does not want the new GPB values stored in the DSP preferences file and must be made before the module is deallocated.

Note
This call must be made before the module is deallocated if the GPB numbers are not to be updated. ◆

SEE ALSO

DSPUpdateGPBPreferenceFile (page 172)

Section API Routines

The section API routines let you control DSP sections. Besides the routines described below, the section API routines include the following:

- `DSPGetSection`, described on page 145

- `DSPConnectSections`, described on page 146

DSPGetSectionData

The `DSPGetSectionData` routine returns a pointer to the actual data that is in a section.

```
pascal OSErr DSPGetSectionData (
    DSPSectionRefNum        theSectionRefNum,
    Ptr                     *theData);
```

Field descriptions

→	theSectionRefNum	The section reference number.
←	theData	The returned pointer to the section's data.

DESCRIPTION

The `DSPGetSectionData` routine is provided to get a pointer to the base of the actual data that is in the section. This is used for sharing parameters between the host and the DSP.

Note
This call can only be made after the task containing the specified section has been installed using `DSPLoadModule`. ◆

DSPGetIndexedSection

The `DSPGetIndexedSection` routine returns the reference number to the indexed section in the specified module.

```
pascal OSErr DSPGetIndexedSection (
    unsigned long          theIndex,
    DSPModuleRefNum        theModuleRefNum,
    DSPSectionRefNum       *theSectionRefNum);
```

Field descriptions

→	theIndex	Index of desired section.
→	theModuleRefNum	Module that contains section.
←	theSectionRefNum	Returned section reference number.

DESCRIPTION

The `DSPGetIndexedSection` routine is used by a client to find information about all the sections that are part of the module designated by `theModuleRefNum`. By calling `DSPGetIndexedSection`, a client can create a list containing the sections for each available module. The client should pass an index starting at 0 (in `theIndex`) and increment it until a `kdspInvalidIndexErr` value is returned.

DSPGetSectionInfo

The `DSPGetSectionInfo` routine returns information about the specified section.

```
pascal OSErr DSPGetSectionInfo (DSPSectionInfoParamBlkHandle
    theSectionParamBlkHdl);
```

Field descriptions

→	`siRefNum`	Input reference number of this section.
←	`siSize`	Returned size of data in actual section.
←	`siFlags`	Returned section control flags.
←	`siType`	Returned section data type.
←	`siName`	Returned name of this section.
←	`siPrimary`	Returned location of section data.
←	`siSecondary`	Returned optional section data storage location.
←	`siPrevSection`	Returned previous connection.
←	`siNextSection`	Returned next connection.
←	`siFIFORefNum`	Returned reference number of the FIFO.

DESCRIPTION

The `DSPGetSectionInfo` routine is used by a client to find information about the specified section. If it is a FIFO section, then the parameter `siFIFORefNum` is returned with the FIFO reference number; otherwise `nil` is returned.

DSPSetSectionSize

The `DSPSetSectionSize` routine sets the size of the specified section.

```
pascal OSErr DSPSetSectionSize(
    DSPSectionRefNum      theSectionRefNum,
    unsigned long         theSize);
```

Field descriptions

→	`theSectionRefNum`	The section reference number.
→	`theSize`	The desired size of the section.

DESCRIPTION

The `DSPSetSectionSize` routine is used to set the size of the section.

DSPNewInterTaskBuffer

The `DSPNewInterTaskBuffer` routine creates a new ITB and returns a reference number to the ITB for the specified task.

```
pascal OSErr DSPNewInterTaskBuffer (
    DSPTaskRefNum        theTaskRefNum,
    unsigned long        theSize,
    unsigned short       theDataType,
    Str31                theInterTaskBufferName,
    short                bankPreference,
    DSPSectionRefNum     *theReturnedSection);
```

Field descriptions

→	`theTaskRefNum`	Pointer to the requesting task.
→	`theSize`	Size of the ITB.
→	`theDataType`	Type of data in the ITB.
→	`theInterTaskBufferName`	Name of the ITB.
→	`bankPreference`	Bank of on-chip SRAM to use for buffer.
←	`theReturnedSection`	Reference number for accessing the ITB.

DESCRIPTION

The `DSPNewInterTaskBuffer` routine is used to create a data buffer that can be shared between two different tasks. The `bankPreference` parameter may be `kdspBankA`, `kdspBankB`, or `kdspAnyBank`. After creating the buffer, use `DSPConnectSections` to connect the buffer to the sections in your tasks.

FIFO API Routines

The FIFO API routines let you create and control DSP FIFO buffers. Besides the routines described in this section, the FIFO API routines include the following:

■ `DSPNewFIFO`, described on page 143

■ `DSPDisposeFIFO`, described on page 150

DSPFIFOGetSize

The `DSPFIFOGetSize` routine returns the size of the specified FIFO.

```
pascal unsigned long DSPFIFOGetSize (DSPFIFORefNum theFIFORefNum);
```

Field descriptions

$\rightarrow$ theFIFORefNum   The FIFO reference number.
$\leftarrow$ FunctionResult   The number of total bytes allocated for the FIFO.

DESCRIPTION

The `DSPFIFOGetSize` routine returns the size of the given FIFO.

DSPFIFOGetReadCount

The `DSPFIFOGetReadCount` routine returns the number of bytes available in a FIFO.

```
pascal unsigned long DSPFIFOGetReadCount (DSPFIFORefNum
   theFIFORefNum);
```

Field descriptions

$\rightarrow$ theFIFORefNum   The FIFO reference number.
$\leftarrow$ FunctionResult   The number of bytes currently in the FIFO.

DESCRIPTION

The `DSPFIFOGetReadCount` returns the number of bytes that can actually be read from a given FIFO.

DSPFIFORead

The `DSPFIFORead` routine reads data from the FIFO into the specified location.

```
pascal unsigned long DSPFIFORead (
   DSPFIFORefNum      theFIFORefNum,
   Ptr                theDestination,
   unsigned long      theCount);
```

Field descriptions

→	theFIFORefNum	Reference number of the FIFO to be read.
→	theDestination	The location in which to place the data being read.
→	theCount	The number of bytes to be read.
←	FunctionResult	The number of bytes that were read.

DESCRIPTION

The DSPFIFORead routine reads data from a FIFO. The theCount parameter is the number of bytes requested. The number of bytes actually read are returned in the FunctionResult parameter. The data is written to theDestination.

DSPFIFOGetWriteCount

The DSPFIFOGetWriteCount routine returns the number of available bytes in a FIFO.

```
pascal unsigned long DSPFIFOGetWriteCount (DSPFIFORefNum
    theFIFORefNum);
```

Field descriptions

→	theFIFORefNum	The FIFO reference number.
←	FunctionResult	The number of empty bytes left in the FIFO.

DESCRIPTION

The DSPFIFOGetWriteCount routine returns the number of bytes that can be written to a FIFO.

DSPFIFOWrite

The DSPFIFOWrite routine writes data into the specified FIFO.

```
pascal unsigned long DSPFIFOWrite (
    DSPFIFORefNum      theFIFORefNum,
    Ptr                theSource,
    unsigned long      theCount);
```

Field descriptions

→	theFIFORefNum	The FIFO reference number to write to.
→	theSource	The location of the data to be written.
→	theCount	The number of bytes to write.
←	FunctionResult	The number of bytes actually written.

DESCRIPTION

The `DSPFIFOWrite` routine writes data to a FIFO. The `theCount` parameter is the number of bytes to be written. The number of bytes actually written are returned in the `FunctionResult` parameter. The data to be written to the FIFO is indicated by `theSource`.

DSPFIFOSwap

The `DSPFIFOSwap` routine swaps new data into the specified FIFO.

```
pascal unsigned long DSPFIFOSwap (
    DSPFIFORefNum     theFIFORefNum,
    unsigned long     theSize,
    Ptr               Logical,
    Ptr               physical,
    Boolean           fifoFull,
    MessageActionProc theInterrupt);
```

Field descriptions

→	`theFIFORefNum`	The FIFO reference number.
→	`theSize`	The size of data buffer allocated.
→	`logical`	The logical address.
→	`physical`	The physical address.
→	`fifoFull`	Flag to tell routine to fill FIFO with data.
→	`theInterrupt`	The procedure for receiving messages from the FIFO.

DESCRIPTION

The `DSPFIFOSwap` routine takes an existing FIFO header and changes the FIFO data. This routine is used with linked FIFOs when a FIFO linked message is received and more data needs to be put into the FIFO. It does not allocate memory and is safe to call at the interrupt level. The FIFO linked message indicates the specific FIFO is empty and can safely be removed from the linked FIFO. If the removed FIFO was allocated by the application, then it must be deallocated by the application.

For more information about `MessageActionProc` see "Sending Messages," earlier in this chapter.

DSPFIFOReset

The `DSPFIFOReset` routine empties the FIFO and clears any pending operations for a FIFO.

```
pascal OSErr DSPFIFOReset (DSPFIFORefNum theFIFORefNum);
```

Field description

→   theFIFORefNum   The FIFO reference number.

DESCRIPTION

The `DSPFIFOReset` routine is used to reset a FIFO after a transaction has finished. This routine disables message passing by calling `DSPFIFOSetMessageMode` with `theFlags` set to `kdspMaskAllMessages`. It then calls `DSPFIFOClearInterrupt` to clear any pending interrupts. Finally, it resets both the read and write indexes to 0, which has the effect of emptying the FIFO.

DSPFIFOClearInterrupt

The `DSPFIFOClearInterrupt` routine clears the interrupt for a FIFO.

```
pascal OSErr DSPFIFOClearInterrupt (DSPFIFORefNum theFIFORefNum);
```

Field description

→   theFIFORefNum   The FIFO reference number.

DESCRIPTION

The `DSPFIFOClearInterrupt` routine clears the interrupt for the given FIFO. Once the DSP has sent a message to a FIFO, it will not send additional messages until this call has been made to clear the current interrupt. Usually this call would be made from the FIFO's `MessageActionProc`.

For more information about `MessageActionProc` see "Sending Messages," earlier in this chapter.

DSPFIFOGetRefCon

The `DSPFIFOGetRefCon` routine returns application-specific information for the specified FIFO.

```
pascal unsigned long DSPFIFOGetRefCon (DSPFIFORefNum
    theFIFORefNum);
```

Field descriptions

→   theFIFORefNum   The FIFO reference number.
←   FunctionResult   The current `RefCon` of the FIFO.

DESCRIPTION

The `DSPFIFOGetRefCon` routine returns the application-specific data for a given DSP FIFO.

DSPFIFOSetRefCon

The `DSPFIFOSetRefCon` routine sets the value of the application-specific data field in a DSP FIFO buffer.

```
pascal OSErr DSPFIFOSetRefCon (
    DSPFIFORefNum  theFIFORefNum,
    unsigned long  theValue);
```

Field descriptions

→   theFIFORefNum   The FIFO reference number.
→   theValue   The desired value of the `RefCon`.

DESCRIPTION

The `RefCon` for the FIFO is an application-specific parameter. When a client application has to pass FIFO parameters to its FIFO `MessageActionProc` it should use this routine to set the parameter. `DSPFIFOGetRefCon` can be used from inside the `MessageActionProc` routine to get the FIFO parameter that was set up by the client.

For more information about `MessageActionProc` see "Sending Messages," earlier in this chapter.

DSPFIFOGetMessageMode

The `DSPFIFOGetMessageMode` routine returns the mode of the specified FIFO.

```
pascal unsigned long DSPFIFOGetMessageMode (DSPFIFORefNum
    theFIFORefNum);
```

Field descriptions

→	theFIFORefNum	The FIFO reference number.
←	FunctionResult	The current value of the FIFO's flags.

DESCRIPTION

The `DSPFIFOGetMessageMode` routine returns the mode set by the `DSPFIFOSetMessageMode` routine.

DSPFIFOSetMessageMode

The `DSPFIFOSetMessageMode` routine sets the message passing mode for a FIFO.

```
pascal OSErr DSPFIFOSetMessageMode (
    DSPFIFORefNum      theFIFORefNum,
    unsigned long      theFlags);
```

Field descriptions

→	theFIFORefNum	The FIFO reference number.
←	theFlags	The desired value of the FIFO's flags.

DESCRIPTION

The `DSPFIFOSetMessageMode` routine is used to enable or disable message passing from the DSP to the FIFO. Setting `theFlags` to `kdspMaskAllMessages` will disable all messaging. Setting `theFlags` to `kdspHalfMessageEnable` will enable the half empty/full messages. Setting `theFlags` to `kdspTerminationMessageEnable` will enable the empty/full message.

To transfer data from the host to the DSP, the typical steps to follow are to set the mode to `kdspHalfMessageEnable` until the host runs out of data, then set the mode to `kdspTerminationMessageEnable`. When the empty message comes in, set the mode to `kdspMaskAllMessages` since the transfer is complete.

Table 4-6 shows the Apple-defined message masks.

For more information about message passing see "Sending Messages," earlier in this chapter.

Table 4-6   Message masks

Message	Comments
kdspFIFOMaskAllMessages	Disable all messages, (p) priority of FIFO messages in descending order
kdspFIFOEnableOverUnderMessage	(4) enable message when FIFO transfer causes an overrun or underrun
kdspFIFOEnableFullEmptyMessage	(3) enable message when FIFO is full or empty
kdspFIFOEnableHighLowMessage	(2) enable message when FIFO is at least half full or half empty
kdspFIFOEnableLinkMessage	(1) enable message when FIFO's link was traversed
kdspFIFOOverUnderTaskInactive	If task accessing FIFO causes either FIFO overrun or underrun then set task inactive
kdspFIFOFullEmptyTaskInactive	If task accessing FIFO causes either FIFO full or empty then set task inactive

DSPFIFOGetMessageActionProc

The DSPFIFOGetMessageActionProc routine returns a pointer to the specified FIFOs message action procedure.

```
pascal OSErr DSPFIFOGetMessageActionProc (
    DSPFIFORefNum        theFIFORefNum,
    MessageActionProc    *theVector);
```

Field descriptions

→	theFIFORefNum	The FIFO reference number.
←	theVector	The FIFO message action procedure.

DESCRIPTION

The DSPFIFOGetMessageActionProc routine provides the location of the specified FIFOs message action procedure. The result is used to get or set the messages associated with the FIFO.

For more information about MessageActionProc see "Sending Messages," earlier in this chapter.

DSPFIFOSetMessageActionProc

The `DSPFIFOSetMessageActionProc` routine sets the location of the message action procedure for the specified FIFO.

```
pascal OSErr DSPFIFOSetMessageActionProc (
    DSPFIFORefNum        theFIFORefNum,
    MessageActionProc    *theVector);
```

Field descriptions

→   `theFIFORefNum`   The FIFO reference number.
→   `theVector`   The FIFO message action procedure.

DESCRIPTION

The `DSPFIFOSetMessageActionProc` routine sets the location of the specified FIFO's message action procedure. The location is used to get or set the messages associated with the FIFO.

For more information about `MessageActionProc` see "Sending Messages," earlier in this chapter.

DSPFIFOSetMessageThreshold

The `DSPFIFOSetMessageThreshold` routine sets the byte level that triggers a FIFO message.

```
pascal OSErr DSPFIFOSetMessageThreshold (
    DSPFIFORefNum     theFIFORefNum,
    unsigned long     theThreshold);
```

Field descriptions

→   `theFIFORefNum`   The FIFO reference number.
→   `theThreshold`   The desired value of the threshold in bytes.
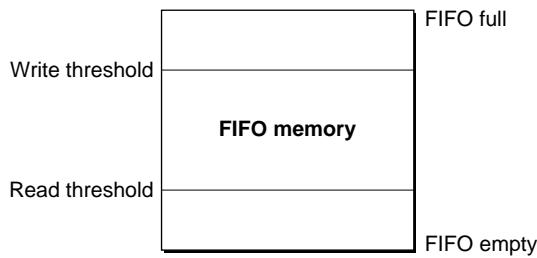
DESCRIPTION

The `DSPFIFOSetMessageThreshold` routine is used by a client application to set the minimum number of bytes, the `theThreshold` value, from the beginning or end of the FIFO memory that are needed to generate the interrupt, as shown in Figure 4-13. The `theThreshold` value is an application-specific parameter subject to these rules:

Reading data: When the FIFO has been filled with the threshold amount of data, then an interrupt will be generated.

Writing data: When the FIFO has been emptied by the amount of the threshold, then an interrupt will be generated.

Maximum `theThreshold` value = `FIFOSize` – 4 bytes

Minimum `theThreshold` value = 4 bytes

Figure 4-13   FIFO threshold

Summary of the Real Time Manager

Constants

All constants used in real-time software are listed in Chapter 5, "DSP Operating System." Only the constants referred to in the present chapter are listed here.

```
//------------------------------------------------------------------
// DSPManager Version Number
//------------------------------------------------------------------
kdspManagerBuildVersion    0x0000000x  // last digit is version number


//------------------------------------------------------------------
// constants used by a client to specify where to insert a task
//------------------------------------------------------------------
kdspHeadInsert          0x00000004     // insert at head of list
kdspTailInsert          0x00000008     // insert at tail of list
kdspBeforeInsert        0x00000010     // insert before reference link
kdspAfterInsert         0x00000020     // insert after reference link
kdspAnyPositionInsert   kdspHeadInsert // insert anywhere in list
```

```
//----------------------------------------------------------------
// constants for messages received by client tasks
//----------------------------------------------------------------
kdspBIOPinChangedState              0x62696f70  // 'biop' (bio pin has changed
                                                // state)
kdspFIFOMessage                     0x66000000  // 'f  ' (prefix used for FIFO
                                                // messages)
kdspFIFOLinkMessage                 0x666c6e6b  // 'flnk' (the FIFO's link was
                                                // traversed)
kdspFIFOOverrunMessage              0x666f7672  // 'fovr' (the FIFO's buffer
                                                // filled before the FIFO write
                                                // completed)
kdspFIFOUnderrunMessage             0x66756e64  // 'fund' (the FIFO's buffer
                                                // emptied before the FIFO read
                                                // completed)
kdspFIFOFullMessage                 0x6666756c  // 'fful' (the FIFO's buffer is
                                                // exactly full)
kdspFIFOEmptyMessage                0x66656d70  // 'femp' (the FIFO's buffer is
                                                // empty)
kdspFIFOHighMessage                 0x66686967  // 'fhig' (the FIFO's buffer is
                                                // at least half full but not
                                                // exactly full)
kdspFIFOLowMessage                  0x666c6f77  // 'flow' (the FIFO's buffer is
                                                // at least half empty but not
                                                // exactly empty)
kdspFIFOPrimeMessage                0x66707269  // 'fpri' (application-specific)
kdspExceptionMessage                0x78000000  // 'x  ' (prefix for dsp
                                                // exception messages)
kdspExceptionReset                  0x78727374  // 'xrst'
kdspExceptionBusError               0x78627573  // 'xbus'
kdspExceptionIllegalOpcode          0x78696c6c  // 'xill'
kdspExceptionReservedOne            0x78727631  // 'xrv1'
kdspExceptionAddressError           0x78616472  // 'xadr'
kdspExceptionDAUOverUnderflow       0x78646175  // 'xdau'
kdspExceptionNotANumber             0x786e616e  // 'xnan'
kdspExceptionReservedTwo            0x78727632  // 'xrv2'
kdspExceptionExternalIntZero        0x78657830  // 'xex0'
kdspExceptionTimer                  0x7874696d  // 'xtim'
kdspExceptionReservedThree          0x78727633  // 'xrv3'
kdspExceptionSIOInputBufFull        0x78736962  // 'xsib'
kdspExceptionSIOOutputBufEmpty      0x78736f62  // 'xsob'
kdspExceptionSIODMAInputFrame       0x78736966  // 'xsif'
kdspExceptionSIODMAOutputFrame      0x78736f66  // 'xsof'
```

```
kdspExceptionExternalIntOne       0x78657831  // 'xex1'
kdspExceptionRuntimeError         0x78657272  // 'xerr'
kdspGPBMessage                    0x67000000  // 'g   ' (prefix used for GPB
                                              // messages)
kdspGPBTaskActive                 0x67616374  // 'gact' (the task is active)
kdspGPBTaskInactive               0x67696e61  // 'gina' (the task is inactive)
kdspGPBFrameOverrun               0x676f7672  // 'govr' (this task was
                                              // involved in a frame overrun
                                              // and is now inactive)
kdspGPBFrameSkip                  0x67736b70  // 'gskp' (this task has skipped
                                              // one or more frames due to a
                                              // frame overrun)
//-------------------------------------------------------------------
// read/write permission constants for clients
//-------------------------------------------------------------------
kdspWritePermission               0x0001
kdspReadPermission                0x0002
kdspReadWritePermission           (kdspWritePermission | kdspReadPermission)


//-------------------------------------------------------------------
// FIFOFlags
//-------------------------------------------------------------------
kdspFIFOMaskAllMessages           0x00000000  // disable all messages (p)
                                              // priority of FIFO messages in
                                              // descending order
kdspFIFOEnableOverUnderMessage    0x00000001  // (4) enable message when FIFO
                                              // transfer causes an overrun
                                              // or underrun
kdspFIFOEnableFullEmptyMessage    0x00000002  // (3) enable message when FIFO
                                              // is full or empty
kdspFIFOEnableHighLowMessage      0x00000004  // (2) enable message when FIFO
                                              // is at least half full or
                                              // half empty
kdspFIFOEnableLinkMessage         0x00000008  // (1) enable message when
                                              // FIFO's link was traversed
kdspFIFOOverUnderTaskInactive     0x00000010  // if task accessing FIFO causes
                                              // either FIFO overrun or
                                              // underrun then set task
                                              // inactive
kdspFIFOFullEmptyTaskInactive     0x00000020  // if task accessing FIFO
                                              // causes either FIFO full or
                                              // empty then set task inactive
```

```
//------------------------------------------------------------------
// GPBFlags
//------------------------------------------------------------------
kdspLumpyModule                     0x00000000  // use bnEstimate
kdspSmoothModule                    0x00000001  // use bnActual


//------------------------------------------------------------------
// SectionFlags
//------------------------------------------------------------------
// Costs the DSP one instruction to use the following flags:
kdspLeaveSection            0x00000000  // do not load or save this section
kdspLoadSection             0x00000001  // load this section
kdspSaveSection             0x00000002  // save this section
kdspClearSection            0x00000004  // fill this section with zeroes
kdspSaveOnContextSwitch     0x00000008  // save this section on context
                                        // switch
kdspExternal                0x00000000  // never loaded on chip
kdspBankA                   0x00000020  // load in Bank A if possible
kdspBankB                   0x00000040  // load in Bank B if possible
kdspAnyBank                 (kdspBankA | kdspBankB)// load anywhere
kdspStaticSection           0x00000080  // this section statically
                                        // allocated before runtime
kdspFIFOSection             0x00000100  // section is a FIFO buffer
kdspReservedSectionFlag0200 0x00000200  // reserved
kdspLoadFIFOSection         0x00000400  // when loading convert from a FIFO
kdspSaveFIFOSection         0x00000800  // when saving convert to a FIFO
kdspHIHOSection             0x00001000  // this is a HIHO Section
kdspReservedForToggleSectionTbl 0x00002000// holds the kdspToggleSectionTable
                                        // flag from the module's flag
kdspLoadHIHOSection         0x00004000  // when loading convert from a HIHO
kdspSaveHIHOSection         0x00008000  // when saving convert to a HIHO

// Costs the DSP two instructions to use the following flags:
kdspNotIOBufferSection      0x00010000  // all cases other than below
kdspInputBuffer             0x00020000  // section is an input buffer
kdspOutputBuffer            0x00040000  // section is an output buffer
kdspITBSection              0x00080000  // section is an InterTask Buffer
kdspScalableSection         0x00100000  // section size can be scaled
kdspSectionAllocated        0x00200000  // reserved for use by the DSP
                                        // Manager
kdspDSPUseOnly              0x00400000  // only DSP should modify this
                                        // memory
```

```
//-------------------------------------------------------------------
// SectionDataTypes
//-------------------------------------------------------------------
kdspNonData                   0x00000000  // data in section is beyond
                                          // description
kdsp3200Float                 0x00000001  // data is in 3200 float format
kdspIEEEFloat                 0x00000002  // data is in IEEE float format
kdspInt32                     0x00000003  // data is 32bit integer
kdspInt1616                   0x00000004  // data is 16bit integer packed
kdspInt8888                   0x00000005  // data is 8bit integer packed
kdspmuLaw                     0x00000006  // data is muLaw format
kdspALaw                      0x00000007  // data is Alaw format
kdspAppSpecificData           0x0000FFFF  // data is application-specific


//-------------------------------------------------------------------
// the processor types 'ptyp'
//-------------------------------------------------------------------
kdsp3210                      '3210'   // DSP3210 hardware compatible
kdsp32C                       '32C '   // DSP32C hardware compatible
```

Data Types

```
DSPPosition:          unsigned long  // type for position parameter
DSPCycles:            unsigned long  // type for processing bandwidth
DSPClientRefNum:      unsigned long  // type for DSPClient reference number
DSPIODeviceRefNum:    unsigned long  // type for DSPIODevice reference number
DSPCPUDeviceRefNum:   unsigned long  // type for DSPCPUDevice reference number
DSPTaskRefNum:        unsigned long  // type for DSPTask reference number
DSPModuleRefNum:      unsigned long  // type for DSPModule reference number
DSPSectionRefNum:     unsigned long  // type for DSPSection reference number
DSPFIFORefNum:        unsigned long  // type for DSPFIFO reference number
DSPLinkedFIFORefNum:  unsigned long  // type for DSPLinkedFIFO ref number
DSPSynchRefNum:       unsigned long  // type for TaskToSynch and SynchTasks
                                     // reference number

pascal void (*MessageActionProc)(struct DSPMessage *theMessage);

enum DSPConnectionType {
   kdspDirectConnection,
   kdspIndirectConnection,
   kdspHIHOConnection
};
```

```
enum DSPTaskPriority {
   kdspRealTime,
   kdspTimeShare
};

enum {
   kdspTaskIsActive,
   kdspTaskIsInactive,
   kdspTaskIsGoingInactive,
   kdspTaskIsGoingActive,
   kdspInvalidTask
} DSPTaskStatus;


DSPCPUDeviceParamBlk     *DSPCPUDeviceParamBlkPtr,
                         **DSPCPUDeviceParamBlkHandle;
DSPIODeviceParamBlk      *DSPIODeviceParamBlkPtr,
                         **DSPIODeviceParamBlkHandle;
DSPBIODeviceParamBlk     *DSPBIODeviceParamBlkPtr,
                         **DSPBIODeviceParamBlkHandle;
DSPSIODeviceParamBlk     *DSPSIODeviceParamBlkPtr,
                         **DSPSIODeviceParamBlkHandle;
DSPClientInfoParamBlk    *DSPClientInfoParamBlkPtr,
                         **DSPClientInfoParamBlkHandle;
DSPTaskInfoParamBlk      *DSPTaskInfoParamBlkPtr,
                         *DSPTaskInfoParamBlkHandle;
DSPModuleInfoParamBlk    *DSPModuleInfoParamBlkPtr,
                         **DSPModuleInfoParamBlkHandle;
DSPSectionInfoParamBlk   *DSPSectionInfoParamBlkPtr,
                         **DSPSectionInfoParamBlkHandle;
DSPGPBModeIndicator      *DSPGPBModeIndicatorPtr,
                         *DSPGPBModeIndicatorHandle;
```

Data Structures

```
//===================================================================
// DSPFIFOAddress
//===================================================================
struct DSPFIFOAddress {
   struct DSPFIFO *l;      // logical pointer
   struct DSPFIFO *p;      // physical pointer
   };
```

```
//====================================================================
// DSPParamBlkHeader
//     This parameter block header is shared between the
//     DSPIODeviceParamblock and the DSPCPUDeviceParamBlk. The
//     reason we declare things this way is so that we can have
//     routines that operate only on the common parts of the
//     structures therefore making the code smaller, easier to
//     write, easier to maintain, etc…
//
//     pbhDeviceIndex           the index for this device
//     pbhClientPermission      read/write permission for this client
//     pbhClientRefNum          the reference number for this client
//     pbhClientName            the name of this client
//     pbhClientICON;           handle to client's icon
//     pbhDeviceName            the device name for the io or cpu device
//     pbhDeviceICON            the device icon for the io or cpu device
//
//====================================================================

#define DSPDeviceParamBlkHeader\
   unsigned short           pbhDeviceIndex;\
   unsigned short           pbhClientPermission;\
   DSPClientRefNum          pbhClientRefNum;\
   Str31                    pbhClientName;\
   Handle                   pbhClientICON;\
   Str31                    pbhDeviceName;\
   Handle                   pbhDeviceICON;\


//====================================================================
// DSPDeviceParamBlk:
//====================================================================


struct DSPDeviceParamBlk {
   DSPDeviceParamBlkHeader
   };


//====================================================================
// DSPCPUDeviceParamBlk:
//     This parameter block is used for controlling a cpu device. It
//     shares the DSPParamBlkHeader with the io device.
//====================================================================
```

```
struct DSPCPUDeviceParamBlk {
   DSPDeviceParamBlkHeader
   unsigned char  cpuSlotNumber;       // the slot number the card is in
   unsigned char  cpuProcessorNumber;  // the processor number, zero based
   OSType         cpuProcessorType;    // the type of the processor
   DSPCycles      cpuMaxCycles;        // max processor execution cycles per
                                       // frame
   DSPCycles      cpuAllocatedCycles;  // num cycles currently allocated
   DSPCycles      cpuCurRealTimeLoading;// num cycles used during the last
                                       // frame
   DSPCycles      cpuTimeShareLoading; // num cycles it took for timeshare
                                       // list
   DSPCycles      cpuTimeShareFreq;    // how often the timeshare list is run
   unsigned long  cpuFrameRate;        // num frames per second for this cpu
                                       // device
   MessageActionProc    cpuClientMessageActionProc;
   };

//====================================================================
// DSPIODeviceParamBlk
//    This parameter block is used for controlling an io device.
//    It shares the DSPParamBlkHeader with the cpu device.
//====================================================================
struct DSPIODeviceParamBlk {
   DSPDeviceParamBlkHeader
   unsigned short iopbIODeviceIndex;   // index of this io device
   unsigned short iopbIODeviceType;    // type of this io device
   unsigned char  iopbReserved[32];    // data specific to the io device
   };

struct DSPBIODeviceParamBlk {
   DSPDeviceParamBlkHeader
   unsigned short iopbIODeviceIndex;   // index of this io device
   unsigned short iopbIODeviceType;    // type of this io device
   struct DSPMessage *bioMessageHandler;// gets called when state of bio pin
                                       // changes
   unsigned char  bioPinState;         // 0 = logic level low, 1 = logic
                                       // level high
   unsigned char  bioPinDirection;     // 0 = input only, 1 = output only
   unsigned char  bioPinIntEnable;     // 0 = interrupt masked, 1 = interrupt
                                       // enabled
   unsigned char  bioReserved[25];     // reserved space
   };
```

```
struct DSPSIODeviceParamBlk {
   DSPDeviceParamBlkHeader
   unsigned short iopbIODeviceIndex;// index of this io device
   unsigned short iopbIODeviceType; // type of this io device
   unsigned long  sioSampleRate;    // sample rate for this io device, both
                                    // input and output
   unsigned long  sioSampleFormat;  // sample size for this io device, both
                                    // input and output
   DSPFIFOAddress sioInputFIFO;     // pass the physical ptr to your dsp code
   DSPFIFOAddress sioOutputFIFO;    // pass the physical ptr to your dsp code
   unsigned char  sioDMAEnable;     // 0 = off, 1 = on, enables both input
                                    // and output
   unsigned char  sioFeatures;      // indicates if the io device can be used
                                    // for standard sound etc…
   unsigned char  sioReserved[6];   // reserved space
   };


//======================================================================
// DSPBandwidth: is used to represent guaranteed processing
// bandwidth information.
//======================================================================
struct DSPBandwidth {          // bn = bandwidth
   DSPCycles     bnEstimate; // worst-case pre-runtime
   DSPCycles     bnActual;   // worst-case runtime
   unsigned long bnFlags;    // control flags
   };


//======================================================================
// DSPMessage: used for passing messages back and forth between the
// kernel and the driver.
//======================================================================
struct DSPMessage {
   MessageActionProc   msVector;     // vector of routine
   unsigned long       msData[3];    // application-specific data
   };


//======================================================================
// DSPGPBModeIndicator: used for requesting more (or less) GPB when
// a module is running.
//======================================================================
struct DSPGPBModeIndicator {
   unsigned short miCurrentMode; // mode in which the module is operating
                                 // (only the DSP code should change this)
```

```
    unsigned short miNextMode;     // (only the Real Time Manager should
                                   // change this)
    };


//----------------------------------------------------------------------
// Get Info Parameter Blocks
//----------------------------------------------------------------------
// DSPClientInfoParamBlk
//     this parameter block is used for getting information about a
//     client that is signed into the DSP Manager.
//----------------------------------------------------------------------
struct DSPClientInfoParamBlk {
    DSPClientRefNum    ciClientRefNum;      // returned ref num of the client
    unsigned short     ciClientPermission;  // returned read/write permission
                                            // of the client
    Handle             ciClientICON;        // returned icon of the client
    Str31              ciClientName;        // returned name of the client
    };
//----------------------------------------------------------------------
// DSPTaskInfoParamBlk
//     this parameter block is used for getting information about a
//     task that is installed on the DSP.
//----------------------------------------------------------------------
struct DSPTaskInfoParamBlk {
    DSPTaskRefNum     tiRefNum;   // returned reference number for this task
    unsigned long     tiRefCon;   // returned application-specific info
    MessageActionProc tiVector;   // returned vector of task action proc
    unsigned long     tiFlags;    // returned flags for DSP task control
    Str31             tiName;     // returned name of this DSP task
    };
//----------------------------------------------------------------------
// DSPModuleInfoParamBlk
//     this parameter block is used for getting information about a
//     module that is installed on the DSP.
//----------------------------------------------------------------------
struct DSPModuleInfoParamBlk {
    DSPModuleRefNum      miRefNum;       // returned reference number of this
                                         // module
    struct DSPBandwidth  miGPB;          // returned guaranteed processing
                                         // bandwidth
    unsigned long        miFlags;        // returned module flags
    unsigned long        miNumSections;  // returned number of sections in
                                         // module
```

```
   Str31                 miName;        // returned name of DSP module
   unsigned long         miExecutions;  // returned number of executions
   unsigned long         miSkipCount;   // returned number of modules to skip
   };
//-------------------------------------------------------------------
// DSPSectionInfoParamBlk
//    this parameter block is used for getting information about a
//    section that is installed on the DSP.
//-------------------------------------------------------------------
struct DSPSectionInfoParamBlk {
   DSPSectionRefNum  siRefNum;       // returned reference number for this
                                     // section
   unsigned long     siSize;         // returned size of data in actual
                                     // section
   unsigned long     siFlags;        // returned section flags
   unsigned long     siType;         // returned for section connection type
                                     // checking
   Str31             siName;         // returned name of this section
   Ptr               siPrimary;      // returned location of section data
   Ptr               siSecondary;    // returned optional section data storage
                                     // location
   DSPSectionRefNum  siPrevSection;  // returned previous connection
   DSPSectionRefNum  siNextSection;  // returned next connection
   DSPFIFORefNum     siFIFORefNum;   // returned reference number of the FIFO
   };
```

Trap Macros and Routine Selectors

```
//===================================================================
// misc dispatcher constants & macros
//===================================================================
#define  _DSPDispatch          0xABF5
#define  MOVEL                 0x303C
#define  DSPDispatch(select)   {MOVEL,select,_DSPDispatch};


//===================================================================
// dispatch selectors for external routines
//===================================================================
#define kdspGetTrapAddress              0
#define kdspSetTrapAddress              1
#define kdspGetIndexedCPUDevice         2
#define kdspOpenCPUDevice               3
```

```
#define kdspCloseCPUDevice              4
#define kdspGetIndexedCPUDeviceOption   5
#define kdspSetIndexedCPUDeviceOption   6
#define kdspGetIndexedIODevice          7
#define kdspOpenIODevice                8
#define kdspCloseIODevice               9
#define kdspGetIndexedIODeviceOption    10
#define kdspSetIndexedIODeviceOption    11
#define kdspSignInCPUDevice             12
#define kdspSignOutCPUDevice            13
#define kdspSignInIODevice              14
#define kdspSignOutIODevice             15
#define kdspLoadModule                  16
#define kdspUnloadModule                17
#define kdspConnectSections             18
#define kdspNewInterTaskBuffer          19
#define kdspNewTask                     20
#define kdspDisposeTask                 21
#define kdspInsertTask                  22
#define kdspManagerVersion              23
#define kdspRemoveTask                  24
#define kdspTaskToSynchronize           25
#define kdspAddressInZone               26
#define kdspSynchronizeTasks            27
#define kdspSetTaskActive               28
#define kdspSetTaskInactive             29
#define kdspCreateZone                  30
#define kdspNewAddress                  31
#define kdspDisposeAddress              32
#define kdspGetAddressSize              33
#define kdspGetZoneSize                 34
#define kdspGetAddress                  35
#define kdspGetZone                     36
#define kdspInsertDouble                37
#define kdspRemoveDouble                38
#define kdspInsertSingle                39
#define kdspRemoveSingle                40
#define kdspGetSection                  41
#define kdspNewFIFO                     42
#define kdspDisposeFIFO                 43
#define kdspGetSectionData              44
#define kdspFIFORead                    45
#define kdspFIFOWrite                   46
```

```
#define kdspFIFOGetReadCount            47
#define kdspFIFOGetWriteCount           48
#define kdspFIFOGetSize                 49
#define kdspAllocateModule              51
#define kdspDisposeModule               52
#define kdspRemoveModule                53
#define kdspGetClientZone               54
#define kdspInitializeStorage           55
#define kdspAllocateStorage             56
#define kdspUnreserveStorage            57
#define kdspDeallocateStorage           58
#define kdspDisposeStorage              59
#define kdspProcessMessages             60
#define kdspFIFOClearInterrupt          61
#define kdspFIFOGetRefCon               62
#define kdspFIFOSetRefCon               63
#define kdspFIFOGetMessageMode          64
#define kdspFIFOSetMessageMode          65
#define kdspFIFOReset                   66
#define kdspSetTaskRefCon               67
#define kdspGetTaskRefCon               68
#define kdspGetIndexedClient            69
#define kdspGetIndexedTask              70
#define kdspGetOwnerClient              71
#define kdspGetIndexedModule            72
#define kdspGetOwnerTask                73
#define kdspGetIndexedSection           74
#define kdspGetOwnerModule              75
#define kdspGetClientInfo               76
#define kdspGetTaskInfo                 77
#define kdspGetModuleInfo               78
#define kdspGetSectionInfo              79
#define kdspConnectITBSections          80
#define kdspConnectTwoFIFOSections      81
#define kdspConnectFIFOToAIAOSections   82
#define kdspConnectFIFOSections         83
#define kdspConnectRegularSections      84
#define kdspAllocate                    85
#define kdspUpdateCPUDeviceInfo         86
#define kdspDisposeOneFIFO              87
#define kdspFIFOSwap                    88
#define kdspCountModule                 89
#define kdspDontCountModule             90
```

```
#define kdspSetSkipCount              91
#define kdspGetTaskStatus             92
#define kdspSetCPUDeviceBondage       93
#define kdspFIFOGetMessageDispatch    94
#define kdspUpdateGPBPrefFile         95
#define kdspFIFOSetMessageThreshold   96
#define kdspSetGPBMode                97
#define kdspDontUpdateGPBPrefs        98
#define kdspUpdateGPBPreferenceFile   99
#define kdspConnectHIHOSections       100
#define kdspDisposeZone               101
#define kdspClientMessageDispatch     102
#define kdspFIFOGetMessageActionProc  103
#define kdspFIFOSetMessageActionProc  104
#define kdspSetSystemTask             105
#define kdspGetAvailableOnChipMemory  106
#define kdspSetSectionSize            107


//====================================================================
// Real Time Manager dispatched traps (external only)
//====================================================================
pascal OSErr DSPGetIndexedCPUDevice
      (DSPCPUDeviceParamBlkPtr theCPUParamBlk)
      = DSPDispatch(kdspGetIndexedCPUDevice)
pascal OSErr DSPOpenCPUDevice
      (DSPCPUDeviceParamBlkPtr theCPUParamBlk)
      = DSPDispatch(kdspOpenCPUDevice)
pascal OSErr DSPCloseCPUDevice
      (DSPCPUDeviceParamBlkPtr theCPUParamBlk)
      = DSPDispatch(kdspCloseCPUDevice)
pascal OSErr DSPUpdateCPUDeviceInfo
      (DSPCPUDeviceParamBlkPtr theCPUParamBlk)
      = DSPDispatch(kdspUpdateCPUDeviceInfo)
pascal OSErr DSPGetIndexedCPUDeviceOption
      (DSPCPUDeviceParamBlkPtr theCPUParamBlk,
      unsigned short theOptionIndex, OSType theOptionType,
      unsigned long *theOption)
      = DSPDispatch(kdspGetIndexedCPUDeviceOption)
pascal OSErr DSPSetIndexedCPUDeviceOption
      (DSPCPUDeviceParamBlkPtr theCPUParamBlk,
      unsigned short theOptionIndex, OSType theOptionType)
      = DSPDispatch(kdspSetIndexedCPUDeviceOption)
```

```
pascal OSErr DSPGetIndexedIODevice
      (DSPIODeviceParamBlkPtr theIOParamBlk)
      = DSPDispatch(kdspGetIndexedIODevice)
pascal OSErr DSPOpenIODevice (DSPIODeviceParamBlkPtr theIOParamBlk)
      = DSPDispatch(kdspOpenIODevice)
pascal OSErr DSPCloseIODevice (DSPIODeviceParamBlkPtr theIOParamBlk)
      = DSPDispatch(kdspCloseIODevice)
pascal OSErr DSPGetIndexedIODeviceOption
      (DSPIODeviceParamBlkPtr theIOParamBlk,
      unsigned short theOptionIndex, OSType theOptionType,
      unsigned long *theOption)
      = DSPDispatch(kdspGetIndexedIODeviceOption)
pascal OSErr DSPSetIndexedIODeviceOption
      (DSPIODeviceParamBlkPtr theIOParamBlk,
      unsigned short theOptionIndex, OSType theOptionType)
      = DSPDispatch(kdspSetIndexedIODeviceOption)
pascal OSErr DSPLoadModule (StringPtr theName,
      DSPTaskRefNum theTaskRefNum, DSPPosition thePosition,
      DSPModuleRefNum theReferenceModuleRefNum,
      DSPModuleRefNum *theNewModuleRefNum, long theScale )
      = DSPDispatch(kdspLoadModule)
pascal OSErr DSPUnloadModule (DSPTaskRefNum theTaskRefNum,
      DSPModuleRefNum theModuleRefNum )
      = DSPDispatch(kdspUnloadModule)
pascal OSErr DSPConnectSections (DSPSectionRefNum sectionOneRefNum,
      DSPSectionRefNum sectionTwoRefNum,
      DSPConnectionType theConnectionType )
      = DSPDispatch(kdspConnectSections)
pascal OSErr DSPNewInterTaskBuffer (DSPTaskRefNum theTaskRefNum,
      unsigned long theSize, unsigned short theDataType,
      Str31 theInterTaskBufferName, short bankPreference,
      DSPSectionRefNum *theReturnedSection )
      = DSPDispatch(kdspNewInterTaskBuffer)
pascal OSErr DSPNewTask (DSPCPUDeviceParamBlkPtr theCPUDevice,
      MessageActionProc theVector, StringPtr theTaskName,
      DSPTaskRefNum *theRefNum )
      = DSPDispatch(kdspNewTask)
pascal OSErr DSPDisposeTask (DSPTaskRefNum theTaskRefNum )
      = DSPDispatch(kdspDisposeTask)
pascal OSErr DSPInsertTask (DSPCPUDeviceParamBlkPtr theCPUDevice,
      DSPTaskRefNum theNewTaskRefNum, DSPPosition thePosition,
      DSPTaskPriority interruptLevel,
      DSPTaskRefNum theRelativeTaskRefNum )
      = DSPDispatch(kdspInsertTask)
```

```
pascal OSErr DSPRemoveTask (DSPTaskRefNum theTaskRefNum )
      = DSPDispatch(kdspRemoveTask)
pascal OSErr DSPTaskToSynchronize (DSPTaskRefNum theRefNum,
      unsigned long frameDelay, DSPSynchRefNum *synchRefNum)
      = DSPDispatch(kdspTaskToSynchronize)
pascal OSErr DSPSynchronizeTasks (DSPSynchRefNum *synchRefNum )
      = DSPDispatch(kdspSynchronizeTasks)
pascal OSErr DSPSetTaskActive (DSPTaskRefNum theRefNum)
      = DSPDispatch(kdspSetTaskActive)
pascal OSErr DSPSetTaskInactive (DSPTaskRefNum theRefNum)
      = DSPDispatch(kdspSetTaskInactive)
pascal OSErr DSPGetSection (DSPModuleRefNum theModuleRefNum,
      StringPtr theSectionName, DSPSectionRefNum *theSectionRefNum )
      = DSPDispatch(kdspGetSection)
pascal OSErr DSPNewFIFO (DSPSectionRefNum theSectionRefNum,
      DSPFIFORefNum *theFIFORefNum, unsigned long theSize,
      Ptr logical, Ptr physical, Boolean fifoFull,
      MessageActionProc theInterrupt)
      = DSPDispatch(kdspNewFIFO)
pascal OSErr DSPDisposeFIFO (DSPSectionRefNum theSectionRefNum)
      = DSPDispatch(kdspDisposeFIFO)
pascal OSErr DSPFIFOFill (DSPFIFORefNum theFIFORefNum,
      unsigned long theSize, Ptr logical, Ptr physical,
      Boolean fifoFull, MessageActionProc theInterrupt)
      = DSPDispatch(kdspFIFOSwap)
pascal OSErr DSPGetSectionData (DSPSectionRefNum theSectionRefNum,
      Ptr *theData)
      = DSPDispatch(kdspGetSectionData)
pascal unsigned long DSPFIFORead (DSPFIFORefNum theFIFORefNum,
      Ptr theDestination,
      unsigned long theCount)
      = DSPDispatch(kdspFIFORead)
pascal unsigned long DSPFIFOWrite (DSPFIFORefNum theFIFORefNum,
      Ptr theSource,
      unsigned long theCount)
      = DSPDispatch(kdspFIFOWrite)
pascal unsigned long DSPFIFOGetReadCount
      (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOGetReadCount)
pascal unsigned long DSPFIFOGetWriteCount
      (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOGetWriteCount)
```

```
pascal unsigned long DSPFIFOGetSize (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOGetSize)
pascal OSErr DSPFIFOClearInterrupt (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOClearInterrupt)
pascal unsigned long DSPFIFOGetRefCon (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOGetRefCon)
pascal OSErr DSPFIFOSetRefCon (DSPFIFORefNum theFIFORefNum,
      unsigned long theValue)
      = DSPDispatch(kdspFIFOSetRefCon)
pascal unsigned long DSPFIFOGetMessageMode
      (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOGetMessageMode)
pascal OSErr DSPFIFOSetMessageMode (DSPFIFORefNum theFIFORefNum,
      unsigned long theFlags)
      = DSPDispatch(kdspFIFOSetMessageMode)
pascal unsigned long DSPManagerVersion (void)
      = DSPDispatch(kdspManagerVersion)
pascal OSErr DSPFIFOReset (DSPFIFORefNum theFIFORefNum)
      = DSPDispatch(kdspFIFOReset)
pascal OSErr DSPSetTaskRefCon (DSPTaskRefNum theRefNum,
      unsigned long theData)
      = DSPDispatch(kdspSetTaskRefCon)
pascal unsigned long DSPGetTaskRefCon (DSPTaskRefNum theRefNum)
      = DSPDispatch(kdspGetTaskRefCon)
pascal OSErr DSPGetIndexedClient (unsigned long theIndex,
      DSPCPUDeviceParamBlkPtr theCPUDevice,
      DSPClientRefNum *theClientRefNum)
      = DSPDispatch(kdspGetIndexedClient)
pascal OSErr DSPGetIndexedTask (unsigned long theIndex,
      DSPTaskPriority interruptLevel,
      DSPCPUDeviceParamBlkPtr theCPUDevice,
      DSPTaskRefNum*theTaskRefNum)
      = DSPDispatch(kdspGetIndexedTask)
pascal OSErr DSPGetOwnerClient (DSPTaskRefNum theTaskRefNum,
      DSPClientRefNum *theClientRefNum)
      = DSPDispatch(kdspGetOwnerClient)
pascal OSErr DSPGetIndexedModule (unsigned long theIndex,
      DSPTaskRefNum theTaskRefNum, DSPModuleRefNum *theModuleRefNum)
      = DSPDispatch(kdspGetIndexedModule)
pascal OSErr DSPGetOwnerTask (DSPModuleRefNum theModuleRefNum,
      DSPTaskRefNum *theTaskRefNum)
      = DSPDispatch(kdspGetOwnerTask)
```

```
pascal OSErr DSPGetIndexedSection (unsigned long theIndex,
      DSPModuleRefNum theModuleRefNum,
      DSPSectionRefNum *theSectionRefNum)
      = DSPDispatch(kdspGetIndexedSection)
pascal OSErr DSPGetOwnerModule (DSPSectionRefNum theSectionRefNum,
      DSPModuleRefNum  *theModuleRefNum)
      = DSPDispatch(kdspGetOwnerModule)
pascal OSErr DSPGetClientInfo
      (DSPClientInfoParamBlkHandle theClientParamBlkHdl)
      = DSPDispatch(kdspGetClientInfo)
pascal OSErr DSPGetTaskInfo
      (DSPTaskInfoParamBlkHandle theTaskParamBlkHdl)
      = DSPDispatch(kdspGetTaskInfo)
pascal OSErr DSPGetModuleInfo
      (DSPModuleInfoParamBlkHandle theModuleParamBlkHdl)
      = DSPDispatch(kdspGetModuleInfo)
pascal OSErr DSPGetSectionInfo
      (DSPSectionInfoParamBlkHandle theSectionParamBlkHdl)
      = DSPDispatch(kdspGetSectionInfo)
pascal OSErr DSPCountModule (DSPModuleRefNum theModuleRefNum)
      = DSPDispatch(kdspCountModule)
pascal OSErr DSPDontCountModule (DSPModuleRefNum theModuleRefNum)
      = DSPDispatch(kdspDontCountModule)
pascal OSErr DSPSetSkipCount (DSPModuleRefNum theModuleRefNum,
      unsigned long theCount)
      = DSPDispatch(kdspSetSkipCount)
pascal DSPTaskStatus DSPGetTaskStatus (DSPTaskRefNum theRefNum)
      = DSPDispatch(kdspGetTaskStatus)
pascal OSErr DSPSetCPUDeviceBondage
      (DSPCPUDeviceParamBlkPtr cpuDevice1,
      DSPCPUDeviceParamBlkPtr cpuDevice2 )
      = DSPDispatch(kdspSetCPUDeviceBondage)
pascal OSErr DSPFIFOSetMessageThreshold
      (DSPFIFORefNum theFIFORefNum,
      unsigned long theThreshold)
      = DSPDispatch(kdspFIFOSetMessageThreshold)
pascal OSErr DSPFIFOSwap (DSPFIFORefNum theFIFORefNum,
      unsigned long theSize, Ptr logical, Ptr physical,
      Boolean fifoFull, MessageActionProc theInterrupt)
      = DSPDispatch(kdspFIFOSwap)
pascal OSErr DSPSetGPBMode (DSPModuleRefNum theModuleRefNum,
      unsigned short desiredMode)
      = DSPDispatch(kdspSetGPBMode)
```

```
pascal OSErr DSPDontUpdateGPBPrefs (DSPModuleRefNum theModuleRefNum)
      = DSPDispatch(kdspDontUpdateGPBPrefs)
pascal OSErr DSPUpdateGPBPreferenceFile
      (DSPModuleRefNum theModuleRefNum)
      = DSPDispatch(kdspUpdateGPBPreferenceFile)
pascal OSErr DSPFIFOGetMessageActionProc
      (DSPFIFORefNum theFIFORefNum,
      MessageActionProc *theVector)
      = DSPDispatch(kdspFIFOGetMessageActionProc)
pascal OSErr DSPFIFOSetMessageActionProc
      (DSPFIFORefNum theFIFORefNum,
      MessageActionProc theVector)
      = DSPDispatch(kdspFIFOSetMessageActionProc)
pascal OSErr DSPGetAvailableOnChipMemory
      (DSPModuleRefNum theModuleRefNum,
      unsigned long *theSize)
      = DSPDispatch(kdspGetAvailableOnChipMemory)
pascal OSErr DSPSetSectionSize (DSPSectionRefNum theSectionRefNum,
      unsigned long theSize)
      = DSPDispatch(kdspSetSectionSize)
```

DSP Operating System

This chapter describes the software routines for building, accessing, and using the digital signal processor (DSP) subsystem of the Macintosh Quadra 840AV and Macintosh Centris 660AV computers. The DSP subsystem provides real-time processing for applications that require a guaranteed throughput. It also provides processing for applications that perform timeshare processing.

Before you read this chapter you should already be familiar with

■ basic Macintosh programming concepts

■ DSP3210 programming

■ creating and using resource files

■ the concepts of digital signal processing given in Chapter 3, "Introduction to Real-Time Data Processing"

This chapter starts with a brief overview of the DSP chip and the DSP3210 register requirements. "DSP Operating System Reference" begins with two main sections: "Creating a Module" and "Building a Section." This chapter does not discuss what the module will do, nor does it explain how to write DSP3210 code.

The remaining parts of the reference section explain the macros used within the DSP operating system environment to manipulate its various components, determine run-time variables, and communicate between the DSP module and the host application.

For information about installing and debugging DSP programs in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Appendix A, "DSP d Commands for MacsBug," Appendix B, "BugLite User's Guide," and Appendix C, "Snoopy User's Guide."

About DSP Modules

The DSP provides both real-time and timeshare processing capabilities for Macintosh applications. Apple provides a complete set of programming macros to help DSP programmers write DSP modules compatible with the DSP operating system. These macros isolate the DSP3210 code contained within the module from the specific implementation of the DSP operating system. This guarantees that the module will be compatible with all DSP3210 implementations on Macintosh computers.

The relation between the DSP programming macros and the DSP operating system and DSP code is diagrammed in Figure 5-1.
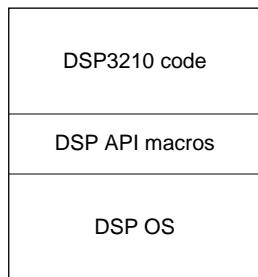
Figure 5-1 DSP programming model

DSP3210 Register Model

Table 5-1 shows how the DSP3210 registers are used.

Table 5-1 DSP3210 register assignments

Register	Usage	Description
r1–r4 r15–r17 a0–a1	scratch	The contents of these registers are not saved or restored. The contents of these registers may be destroyed by DSP API routines.
r5–r14 a2–a3	protected	The contents of these registers must always be saved and restored when they are used by the programmer. The contents of these registers are always saved before and restored after they are used by the DSP API routines.
r18 (RV)	return	The DSP operating system always calls the first instruction in the entry section of each module. Register r18 contains the return vector to get back to the DSP operating system when the module is finished executing. Before jumping to the return vector, all protected registers must be restored to the same values they contained upon initial entry to the module.
r19	reserved	This register is reserved by Apple. Do not alter its contents.
r20	reserved	This register is reserved by Apple. Do not alter its contents.
r21	stack	This register is the common stack pointer register shared by the programmer and the DSP operating system. Register r21 always points to the next available stack location. Therefore, r21 is pre-decremented for pops and post-incremented for pushes.
r22	reserved	This register is reserved by Apple. Do not alter its contents.

32-Bit Data Transfers

Because of the implementation of the DSP3210 floating-point instruction set it is necessary to use a specific instruction when moving 32-bit data between memory locations. The DSP3210 hardware checks for a valid floating-point value in `dau` instructions. If a floating-point number whose least significant byte is 0 is moved into an accumulator register, the whole accumulator register is zeroed. For example, the correct way to move 32-bit data from memory pointed to by `r3` into memory pointed to by `r2` is the following:

```
a0 = (*r2++ = *r3++) *a0
```

The following command is *incorrect,* because it will not work for some floating-point numbers:

```
*r2++ = a0 = *r3++
```

DSP Program Information for the Macintosh Programmer

Using DSP modules effectively isolates DSP programming from Macintosh programming. However, since DSP modules are installed by Macintosh applications, the DSP programmer must document some basic information about each module. This information is used by the Macintosh application programmer to create applications that install and use these modules. The information should include:

■ the names of all input and output sections, their section types, and their section data types

■ the names of all parameter sections and their formats

■ the GPB scaling vectors supported by the module

■ the module's grouping assumptions

■ the module's run-time environment

These items of information for each module are discussed below.

Input and Output Sections

The Macintosh programmer needs to know the names of all sections in the DSP module that are input or output sections. The type of each section and the type of the data contained in the section must also be specified. This information is used by Macintosh applications to make connections between DSP sections.

Parameter Sections

The names of all sections in the DSP module that are parameter sections must be documented by the DSP programmer. The format for the data in each parameter section must also be specified. This information is used by Macintosh applications to use and manipulate data in DSP parameter sections.

GPB Scaling Vectors

The DSP programmer must document what GPB scaling vectors are declared in the header of the DSP module. These vectors specify the frame rates, buffer scaling vectors, and GPB estimates for the module. This information is used by the application to determine the appropriate mode selection.

Grouping Assumptions

The Macintosh programmer needs to know if the DSP module makes assumptions about group ordering. Modules that depend upon other modules, change their skip count, or set their owner task inactive, are examples of modules that make assumptions about group ordering. The information is used by the application to determine the order of module installation, module grouping, and intermodule dependencies.

Run-Time Environment

The DSP programmer must tell the Macintosh programmer whether to install each DSP module into the real-time or timeshare DSP run-time environment. The requirements and expectations for running the module in each environment must also be specified. For example, if the module runs in real time what are its data input requirements? If the module runs as a timeshare processor, how is it expected to be used? This information helps the Macintosh programmer determine how the module can be used in the application.

DSP Operating System Reference

This reference section is divided into three parts. The first part covers creating a new module. This lays the foundation for the DSP program by creating the structure for the resource that will be loaded on the DSP chip.

The second part covers the process of building a module by creating sections to hold all the components needed to perform the desired functions. Creating sections is described in the following order:

■ code and variables

■ data input

■ data output
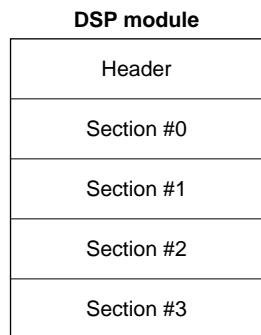
The third part explains all the DSP operating system macros you can use, from low-level system macros to host-level communication routines. This part is organized from the bottom up—the lowest level routines used for system level manipulation are described first. The section, module, task, and FIFO manipulation macros are explained in that order. They are followed by the guaranteed processing bandwidth (GPB), semaphore, and message manipulation macros.

Creating a Module

DSP modules are the basic units that are stored on disk and loaded by Macintosh applications. Each module is composed of DSP sections created by the DSP programmer. Figure 5-2 shows how a module is constructed. For information on building sections refer to "Building a Section," later in this chapter.

Figure 5-2   DSP module structure

DSP module

Header
Section #0
Section #1
Section #2
Section #3

NewModule

The `NewModule` macro creates a new DSP module.

```
NewModule (Name, GPBFlags, ModuleFlags, EntryName)
```

`Name`	The name of this module.
`GPBFlags`	The `GPBFlags` argument to `NewModule` contains information about how the module's GPB is used in the GPB calculation.
	Valid `GPBFlags` are:

`kdspLumpyModule`	Use `bnEstimate`
`kdspSmoothModule`	Use `bnActual`

ModuleFlags

The `ModuleFlags` argument to `NewModule` contains information about the module's run-time environment. These flags determine if the module is automatically cached by the DSP operating system (`AutoCache`) or explicitly cached by the DSP programmer (`DemandCache`). You must use `OR` to combine all flags that apply.

Valid `ModuleFlags` are:

`kdspAutoCache`	Select auto cache model
`kdspDemandCache`	Select demand cache model
`kdspOnChipSectionTable`	Put section table on-chip
`kdspOnChipStack`	A stack of the specified size is created on-chip
`kdspOffChipStack`	A stack of the specified size is created off-chip

EntryName

Entry point into the module. This is the name of the code that the DSP operating system will call when the module is executed.

DESCRIPTION

When the `NewSection` macro uses the name specified by `ModuleName`, the new section is included in the specified module. The `NewModule` macro declares `EntryName` as the entry point to the first section. The programmer must also specify the location of the stack, either on-chip or off-chip, for this module.

The programmer must declare a set of GPB scaling vectors immediately after the `NewModule` macro. At least one scaling vector is required; additional vectors are optional. Each of the vectors consists of three 32-bit integers: the frame rate, the scaling factor used to multiply the size of the scalable section, and the GPB estimate for this scale. For example:

```
long  <min demand cache stack size> // Demand Cache only
long  <stack size>                  // Demand Cache only
long  100,240,3000      // scaling vector for 100 frames per
                        //   second, 240 samples per frame
long  100,320,4500      // scaling vector for 100 frames per
                        //   second, 320 samples per frame
```

DSP modules should not assume

■ the frame rate associated with the data being accessed

■ the sample rate associated with the data being accessed

▲ **WARNING**
The GPB estimate cannot be 0. You should specify a positive number as close to the actual processing time as you can estimate. ▲

Building a Section

A section is the minimum DSP program unit that can be cached on the DSP chip. The following macros are used in your source code to create code and variable sections or add to them. A section remains active until a new section is declared by a routine such as `AppendSection`.

NewSection

The `NewSection` macro creates a new section that remains active until the next section is declared.

`NewSection (Name, SectionFlags, SectionDataType, ModuleName)`

`Name`	The name of this section.
`SectionFlags`	The flags for controlling run-time operation.
`SectionDataType`	The type of data that is in this section.
`ModuleName`	The name of the module this section is attached to.

DESCRIPTION

The `Name` field contains the section name. This section is included in the module specified by `ModuleName`. Each section requires a set of flags to assist the operating system in handling the section correctly.

Although the `NewSection` routine can be used to create all types of code, variable, and data sections there are a set of standard section macros available to make the most commonly used sections. This routine should only be used if no other routine listed in this chapter can accomplish the desired functionality. A complete understanding of both the DSP operating system and the section control flags is needed to use this routine properly.

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip; the `PopSection` macro (page 227) is used to move a section off-chip. ◆

SECTION CONTROL FLAGS

The `SectionFlags` argument to `NewSection` controls the caching operation of the new section. Remember to use `OR` to combine the parts that apply. All unused bits are reserved for use by Apple and should be left as zeros.

If the module can run in timeshare then it is important to know the sections that need to be saved back into main memory before the real-time modules execute. This saves the DSP operating system from having to page out the entire on-chip memory for every

context switch. For example, code sections would generally not need the Save flag set, but variable sections would. Remember that a context switch can occur at any time during a timeshare task. Section control flags are listed in Table 5-2.

Table 5-2   Section flags

DSP constant	Comment
kdspLeaveSection	Do not load or save this section
kdspLoadSection	Load this section on-chip
kdspSaveSection	Save this section off-chip
kdspClearSection	Clear this section before it is used
kdspSaveOnContextSwitch	Save this section to main memory if a context switch is done

Each section can be cached into either memory bank A or B on the DSP3210. Generally Bank A is used for program space, variables, tables, state, and coefficients, and Bank B is used for buffers. Bank preference is controlled by the programmer, using the constants listed in Table 5-3, not by the DSP operating system.

Table 5-3   DSP3210 bank preferences section flags

DSP constant	Comment
kdspExternal	Never load on-chip
kdspBankA	Load in Bank A
kdspBankB	Load in Bank B
kdspAnyBank	Load in either bank, use system preference
kdspStaticSection	This section statically allocated before run time

The type of data contained in a section is specified by the `SectionType` flags. A section can have any of the Apple-defined section types shown in Table 5-4.

Table 5-4   Buffer type section flags

DSP constant	Comments
kdspFIFOSection	Section is a FIFO buffer
kdspNotIOBufferSection	All cases except those below, or code section
kdspInputBuffer	Section is an input buffer
kdspOutputBuffer	Section is an output buffer
kdspScalableSection	Section size is scalable
kdspDSPUseOnly	Only DSP should modify this memory

The `SectionDataType` argument to `NewSection` defines what type of data is in the section. Data types have been predefined by Apple. They are used when connecting buffers to ensure that the buffers contain compatible data as expected by the module. Data types are shown in Table 5-5; only one can be used at a time.

Table 5-5   Data type flags

DSP constant	Comments
kdspNonData	Data in this section is not describable, or is code
kdsp3200Float	Data is 3200 float format
kdspIEEEFloat	Data is IEEE float format
kdspInt32	Data is 32-bit integer format
kdspInt1616	Data is 16-bit integer packed format
kdspInt8888	Data is 8-bit integer packed format
kdspmuLaw	Data is `muLaw` format
kdspALaw	Data is `ALaw` format
kdspAppSpecificData	Data is application specific

Code and Variables

The following macros create new DSP code and variables.

NewCachedProgramSection

The `NewCachedProgramSection` macro defines a code section that is loaded on the DSP chip.

```
NewCachedProgramSection (Name, ModuleName)
```

| Name | The name of this section. |
| ModuleName | The name of the module this section is attached to. |

DESCRIPTION

The `NewCachedProgramSection` macro does not put any code into the section. This macro only creates the header; code is then added using the `AppendSection` macro.

Note
In a `DemandCache` module use the call
`PushSection(theSectionName)` to move
the section on-chip. ◆

AppendSection

The `AppendSection` macro is used to add contents to an already defined section.

```
AppendSection (Name)
```

Name   The name of the section this code is appended to.

DESCRIPTION

The primary application of the `AppendSection` macro is to add code to the first section in the module. Anything following this macro will be added to the section specified by `Name`.

▲ **WARNING**
The `AppendSection` macro cannot be used to extend an AIAO or a FIFO section. Attempts to extend these section types will not work and will result in a loss of data. ▲

NewParameterSection

The `NewParameterSection` macro creates a parameter section. This section is not loaded or saved.

```
NewParameterSection (Name, SectionDataType, ModuleName)
```

Name   The name of this section.
SectionDataType   The type of data that is in this section.
ModuleName   The name of the module this section is attached to.

DESCRIPTION

The `NewParameterSection` macro defines a parameter section for passing control or status between the module and another module or between the module and a Macintosh application. Parameter sections that are shared between a module and an application should not be saved back to main memory. This would cause changes made by the application to the copy of the buffer in main memory to be overwritten. Locked read-modify-write cycles should be used for any data in a parameter buffer that is being jointly updated by both the host application and the module. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip; the `PopSection` macro (page 227) is used to move a section off-chip. ◆

NewTableSection

The `NewTableSection` macro creates a table section. This section is loaded in the DSP, but is not saved.

```
NewTableSection (Name, SectionDataType, ModuleName)
```

Name	The name of this section.
SectionDataType	The type of data that is in this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewTableSection` macro defines a table buffer for providing a fixed table of data for use within a module. Code is added to this section using the `AppendSection` macro. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Note

When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip; the `PopSection` macro (page 227) is used to move a section off-chip. ◆

NewStateVariableSection

The `NewStateVariableSection` macro creates a table section that is both loaded and saved.

```
NewStateVariableSection (Name, SectionDataType, ModuleName)
```

Name	The name of this section.
SectionDataType	The type of data that is in this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewStateVariableSection` macro defines a state variable buffer for providing a variable table of data for use within a module. If any data in the parameter buffer is being jointly updated by both the host and the DSP, then locked read-modify-write cycles should be used. Code is added to this section using the `AppendSection` macro. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Note

When used in a `DemandCache` module, the `PushSection` macro
(page 228) is used to move a section on-chip; the `PopSection`
macro (page 227) is used to move a section off-chip. ◆

NewTempVariableSection

The `NewTempVariableSection` macro creates a temporary on-chip buffer. Nothing is
loaded into this section and it is not saved except during a timeshare context switch.

`NewTempVariableSection (Name, SectionDataType, ModuleName)`

`Name`	The name of this section.
`SectionDataType`	The type of data that is in this section.
`ModuleName`	The name of the module this section is attached to.

DESCRIPTION

The `NewTempVariableSection` macro defines a scratch buffer for use by the module
code section while the module is executing. The `SectionDataType` field uses the same
selection of values as the `SectionDataType` field in the `NewSection` routine
(page 210).

Note

When used in a `DemandCache` module, the call
`PushSection (theSectionName)` is used to move a
section on-chip; the call `PopSection (theSectionName)`
is used to move a section off-chip. ◆

NewTempScalableAIAOSection

The `NewTempScalableAIAOSection` macro creates a temporary on-chip buffer.
Nothing is loaded into this section and it is not saved except during a timeshare
context switch.

`NewTempScalableAIAOSection (Name, AIAOScale, SectionDataType,`
   `ModuleName)`

`Name`	The name of this section.
`AIAOScale`	The scale factor for controlling AIAO size.
`SectionDataType`	The type of data that is in this section.
`ModuleName`	The name of the module this section is attached to.

DESCRIPTION

The `NewTempScalableAIAOSection` macro defines a temporary scalable AIAO section. Typically used as a scratch area that must be sized in relationship to the input data for the module. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip; the `PopSection` macro (page 227) is used to move a section off-chip. ◆

NewExternalProgramSection

The `NewExternalProgramSection` macro creates a code section that is not loaded or saved.

```
NewExternalProgramSection (Name, ModuleName)
```

Name	The name of this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewExternalProgramSection` macro defines a code section which is never cached on the DSP chip. It is not saved during a context switch. Code is added to this section using the `AppendSection` macro (page 213).

Data Input

The macros described in this section create DSP input buffers.

NewInputAIAOSection

The `NewInputAIAOSection` macro creates an AIAO section that is loaded, but saved only during a context switch.

```
NewInputAIAOSection (Name, AIAOSize, SectionDataType, ModuleName)
```

Name	The name of this section.
AIAOSize	The AIAO size.
SectionDataType	The type of data that is in this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewInputAIAOSection` macro defines an AIAO section that connects to another AIAO section. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

▲ **WARNING**
You cannot append data to an AIAO section using the `AppendSection` macro. Attempting to do so will result in a loss of data. ▲

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewScalableInputAIAOSection

The `NewScalableInputAIAOSection` macro creates a scalable AIAO section that is loaded, but saved only during a context switch.

```
NewScalableInputAIAOSection (Name, AIAOScale, SectionDataType,
    ModuleName)
```

Name	The name of this section.
AIAOScale	Scale factor for controlling AIAO size.
SectionDataType	The type of data that is in this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewScalableInputAIAOSection` macro defines a scalable AIAO section that connects to another AIAO section. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

▲ **WARNING**
Data cannot be appended to an AIAO section using the `AppendSection` macro. Any data that is added in this way will result in a loss of data. ▲

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewInputFIFOAndBufferSection

The `NewInputFIFOAndBufferSection` macro creates an input FIFO section that is saved only during a context switch.

```
NewInputFIFOAndBufferSection (Name, BufferSize, SectionDataType,
    ModuleName)
```

Name   The name of this section.

BufferSize   The buffer size.

SectionDataType   The type of data that is in this section.

ModuleName   The name of the module this section is attached to.

DESCRIPTION

The `NewInputFIFOAndBufferSection` macro names a FIFO buffer and defines a temporary FIFO section to hold data read from the FIFO. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Data is moved from the FIFO buffer to the FIFO section using the `FIFORead`, `FIFOReadN`, and `FIFOReadNBuffer` macros. The `GetSectionAddress`, `GetSectionLabel`, and `GetSectionSize` macros return information about the FIFO section.

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewInputFIFOAndScalableBufferSection

The `NewInputFIFOAndScalableBufferSection` macro creates a scalable input FIFO section that is saved only during a context switch.

```
NewInputFIFOAndScalableBufferSection (Name, BufferScale,
    SectionDataType, ModuleName)
```

Name   The name of this section.

BufferScale   The scale factor for controlling buffer size.

SectionDataType   The type of data that is in this section.

ModuleName   The name of the module this section is attached to.

DESCRIPTION

The `NewInputFIFOAndScalableBufferSection` macro names a FIFO buffer and defines a scalable temporary FIFO section to hold data read from the FIFO. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Data is moved from the FIFO buffer to the FIFO section using the `FIFORead`, `FIFOReadN`, and `FIFOReadNBuffer` macros. The `GetSectionAddress`, `GetSectionLabel`, and `GetSectionSize` macros return information about the FIFO section.

Note

When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

Data Output

The macros described in this section create DSP output buffers.

NewOutputFIFOAndBufferSection

The `NewOutputFIFOAndBufferSection` macro creates an output FIFO section that is saved only during a context switch.

```
NewOutputFIFOAndBufferSection (Name, BufferSize, SectionDataType,
    ModuleName)
```

Name	The name of this section.
BufferSize	The buffer size.
SectionDataType	The type of data that is in this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewOutputFIFOAndBufferSection` macro names a FIFO buffer and defines a temporary FIFO section to hold data before it's written to the FIFO. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Data is moved from the FIFO section to the FIFO buffer using the `FIFOWrite`, `FIFOWriteN`, and `FIFOWriteNBuffer` macros. The `GetSectionAddress`, `GetSectionLabel`, and `GetSectionSize` macros return information about the FIFO section.

Note

When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewOutputFIFOAndScalableBufferSection

The `NewOutputFIFOAndScalableBufferSection` macro creates a scalable output FIFO section that is saved only during a context switch.

```
NewOutputFIFOAndScalableBufferSection (Name, BufferScale,
    SectionDataType, ModuleName)
```

Name   The name of this section.

BufferScale   The scale factor for controlling buffer size.

SectionDataType   The type of data that is in this section.

ModuleName   The name of the module this section is attached to.

DESCRIPTION

The `NewOutputFIFOAndScalableBufferSection` macro names a FIFO buffer and defines a scalable temporary FIFO section to hold data before it's written to the FIFO. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

Data is moved from the FIFO section to the FIFO buffer using the `FIFOWrite`, `FIFOWriteN`, and `FIFOWriteNBuffer` macros. The `GetSectionAddress`, `GetSectionLabel`, and `GetSectionSize` macros return information about the FIFO section.

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewOutputCRBSection

The `NewOutputCRBSection` macro creates a complete result buffer (CRB) section that is saved off-chip and saved during a context switch.

```
NewOutputCRBSection (Name, AIAOSize, SectionDataType, ModuleName)
```

Name   The name of this section.

AIAOSize   The AIAO size.

SectionDataType   The type of data that is in this section.

ModuleName   The name of the module this section is attached to.

DESCRIPTION

The `NewOutputCRBSection` macro defines an AIAO section that connects to another AIAO section. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

▲ **WARNING**
You cannot append data to an AIAO section using the `AppendSection` macro. Attempting to do so will result in a loss of data. ▲

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewScalableOutputCRBSection

The `NewScalableOutputCRBSection` macro creates a scalable complete result buffer (CRB) section that is saved off-chip and saved during a context switch.

```
NewScalableOutputCRBSection (Name, AIAOScale, SectionDataType,
    ModuleName)
```

Name	The name of this section.
AIAOScale	The scale factor for controlling AIAO size.
SectionDataType	The type of data that is in this section.
ModuleName	The name of the module this section is attached to.

DESCRIPTION

The `NewScalableOutputCRBSection` macro defines a scalable AIAO section that connects to another AIAO section. The CRB section is initially cleared to 0. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

▲ **WARNING**
You cannot append data to an AIAO section using the `AppendSection` macro. Attempting to do so will result in a loss of data. ▲

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewOutputPRBSection

The `NewOutputPRBSection` macro creates a partial result buffer (PRB) section that is cleared when created on-chip and saved off-chip. It is also saved during a context switch.

`NewOutputPRBSection (Name, AIAOSize, SectionDataType, ModuleName)`

Name   The name of this section.

AIAOSize   The AIAO size.

SectionDataType   The type of data that is in this section.

ModuleName   The name of the module this section is attached to.

DESCRIPTION

The `NewOutputPRBSection` macro defines an AIAO section that connects to another AIAO section. Data input to this section is summed with any data already in the AIAO section. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

▲ **WARNING**
You cannot append data to an AIAO section using the `AppendSection` macro. Attempting to do so will result in a loss of data. ▲

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

NewScalableOutputPRBSection

The `NewScalableOutputPRBSection` macro creates a scalable partial result buffer (PRB) section that is cleared when created on-chip and saved off-chip. It is also saved during a context switch.

`NewScalableOutputPRBSection (Name, AIAOScale, SectionDataType, ModuleName)`

Name   The name of this section.

AIAOScale   The scale factor for controlling AIAO size.

SectionDataType   The type of data that is in this section.

ModuleName   The name of the module this section is attached to.

The `NewScalableOutputPRBSection` macro defines a scalable AIAO section that connects to another AIAO section. Data input to this section is summed with any data already in the AIAO section. The `SectionDataType` field uses the same selection of values as the `SectionDataType` field in the `NewSection` routine (page 210).

▲ **W A R N I N G**
You cannot append data to an AIAO section using the `AppendSection` macro. Attempting to do so will result in a loss of data. ▲

Note
When used in a `DemandCache` module, the `PushSection` macro (page 228) is used to move a section on-chip. ◆

DSP Operating System Macros

Apple provides a set of DSP operating system macros to assist DSP programmers in manipulating DSP modules and sections. All DSP operating system macros are register-based and use the scratch registers `r1` through `r4` for passing parameters and returning results. In addition, the contents of the scratch registers `r15` through `r18`, `a0`, and `a1` may be destroyed by the macros.

General Manipulation Macros

The macros described in this section perform general tasks in DSP programming.

BlockMove

The `BlockMove` macro moves blocks of data from a source to a destination.

```
BlockMove(theSrcPtr, theDestPtr, theCount)
```

`theSrcPtr`	Must be a `cau` register `r1-r14`.
`theDestPtr`	Must be a `cau` register `r1-r14` or a constant.
`theCount`	Value (in bytes) is divided by four and only `int()` is used. (Example: 17 bytes / 4 = 4). It may be any `cau` register `r1-r17` or a constant.

REGISTER USAGE

The `BlockMove` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `BlockMove` macro copies `theCount` bytes from `theSrcPtr` to `theDestPtr`. The move is accomplished using the instruction `a0 = (*r2++ = *r1++) * a0`. The value of `theCount` must be exactly divisible by four. If it is not, only the integer result will be used. (For example, 18 bytes / 4 = 4.) The values of `theSrcPtr` and `theDestPtr` must be addresses on a longword boundary. If it is not, the address is rounded down to the nearest longword boundary. (Example: $00000802 → $00000800.) The values of `theSrcPtr` and `theDestPtr` cannot be 0.

PcLabel

The `PcLabel` macro locates the specified section on-chip and returns its offset..

```
register = PcLabel(theSectionLabel)
```

theSectionLabel   Label used within this section.

REGISTER USAGE

The `PcLabel` macro does not alter the contents of any register except `register`.

DESCRIPTION

The `PcLabel` macro calculates the offset from the current `pc` location to the `theSectionLabel` address and places the offset into `register`, which may be any `cau` register `r1-r18`.

Pop

The `Pop` macro pops the stack.

```
Pop (theRegister)
```

theRegister   Return value, may be from any `cau` register.

REGISTER USAGE

The `Pop` macro does not alter the contents of any register except `register` and the stack pointer (`SP`).

DESCRIPTION

The `Pop` macro pre-decrements `SP` and then reads a longword into `theRegister`, which may be any `cau` register `r1-r18`.

Push

The `Push` macro pushes the stack.

```
Push (theRegister)
```

theRegister   Any `cau` register.

REGISTER USAGE

The `Push` macro does not alter the contents of any register except `SP`. This macro does not affect the `cau` flags.

DESCRIPTION

The `Push` macro writes the longword from `cau` register `theRegister` to the top of the stack and then increments `SP` by four.

Section Manipulation Macros

The following macros help you work with DSP sections.

CallSection

The `CallSection` macro branches to continue execution at the specified section.

```
CallSection (theSectionName)
```

theSectionName   The name of the section to continue operation at.

REGISTER USAGE

The `CallSection` macro does not alter the contents of any registers except `RV`.

DESCRIPTION

The `CallSection` macro takes one argument, the section name of a DSP section to branch to and continue execution. The section whose name is `theSectionName` must be a DSP section located within the same DSP module as the current section.

The `CallSection` macro places the return address in `RV`.

GetSectionAddress

The `GetSectionAddress` macro returns the physical address of the specified section.

`GetSectionAddress (theSectionPtr,theSectionName)`

theSectionPtr   Returns a value, physical location of section.

theSectionName   The name of section to locate.

REGISTER USAGE

The `GetSectionAddress` macro does not alter the contents of any registers except `theSectionPtr`.

DESCRIPTION

The `GetSectionAddress` macro calculates the physical address of `theSectionName` and copies the address into the `theSectionPtr` register, which may be any `cau` register `r1-r18`.

GetSectionLabel

The `GetSectionLabel` macro returns a physical pointer to a label in the specified section.

`GetSectionLabel (theSectionLabelPtr,theSectionLabel)`

theSectionLabelPtr   Returns a pointer, physical location of section.

theSectionLabel   Label used within the section.

REGISTER USAGE

The `GetSectionLabel` macro does not alter the contents of any register except `theSectionLabelPtr`.

DESCRIPTION

The `GetSectionLabel` macro returns a physical pointer to a label designated by `theSectionLabel`. The pointer is returned in `theSectionLabelPtr`, which may be any `cau` register `r1-r18`.

GetSectionSize

The `GetSectionSize` macro returns the size of the specified section.

`GetSectionSize (theSectionSize,theSectionName)`

theSectionSize   The size of the section.
theSectionName   The section name.

REGISTER USAGE

The `GetSectionSize` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GetSectionSize` macro calculates the size of `theSectionName` and copies it into the `theSectionSize` register, which may be any `cau` register `r1-r18`.

PopSection

The `PopSection` macro caches the specified section off-chip.

`PopSection (theSectionName)`

theSectionName   The section name.

REGISTER USAGE

The `PopSection` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `PopSection` macro caches `theSectionName`. The actual caching operation performed depends upon the section's caching flags.

For static sections, `PopSection` caches the section data from its primary container to its secondary container. For non-static sections, `PopSection` caches the section data from the top of the demand cache stack to its primary container.

Note

The Save flag must be set (caching flags) for the specified section if data
is to be moved. The memory space is automatically reclaimed by the
DSP operating system. ◆

▲ **WARNING**

Sections must use `PopSection` in the reverse order that they use
`PushSection`. ▲

PushSection

The `PushSection` macro loads the specified section on-chip.

`PushSection (theSectionName)`

`theSectionName`   The section name.

REGISTER USAGE

The `PushSection` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`,
and `a0-a1`.

DESCRIPTION

The `PushSection` macro caches `theSectionName`. The actual caching operation
performed depends upon the section's caching flags.

For static sections `PushSection` caches the section data from its secondary container to
its primary container. For non-static sections, `PushSection` caches the section data
from its primary container to the top of a demand cache stack.

Note

You must set the Load flag (caching flags) for the specified section if data
is to be moved. The Clear flag must be set if the section is to be cleared.
Either the Bank A or Bank B flag should also be set. If no Bank flag or the
Don't Care flag is selected the DSP operating system will use Bank A. ◆

Module Manipulation Macro

The `SetSkipCount` macro helps you program DSP modules.

SetSkipCount

The `SetSkipCount` macro sets the skip count (number of modules to be jumped over).

```
SetSkipCount (theSkipCount)
```

theSkipCount   The number of modules to skip over.

REGISTER USAGE

The `SetSkipCount` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SetSkipCount` macro sets the skip count for the currently executing module. The current module continues its execution. When the module finishes its execution, the new skip count takes effect.

The `theSkipCount` parameter is a 32-bit constant or any `cau` register in the range `r1` through `r17`.

Task Manipulation Macros

The macros described in this section help you work with tasks.

GetNumRealTimeFrames

The `GetNumRealTimeFrames` macro returns the number of real-time frames that have been executed.

```
GetNumRealTimeFrames (numFrames)
```

numFrames   The number of frames executed.

REGISTER USAGE

The `GetNumRealTimeFrames` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `GetNumRealTimeFrames` macro is used to get the number of real-time frames that have been executed since the DSP was started or reset.

SetTaskInactive

The `SetTaskInactive` macro turns off the task associated with the section that is using it.

```
SetTaskInactive ()
```

REGISTER USAGE

The `SetTaskInactive` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SetTaskInactive` macro sets the owner task for the currently executing module inactive. Setting the task inactive does not take effect until the next frame. The task's modules complete their execution for the current frame.

FIFO Manipulation Macros

The macros described in this section help you work with FIFO buffers.

▲ **WARNING**
Although FIFO manipulations deal with byte counts, all operations
must be done in longword (4 bytes) increments only. Use of the FIFO
calls with non-longword counts will cause unpredictable results. ▲

FIFOGetReadCount

The `FIFOGetReadCount` macro returns the available number of data bytes in the FIFO.

```
FIFOGetReadCount (theFIFOName)
```

theFIFOName   The FIFO name.

REGISTER USAGE

The `FIFOGetReadCount` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOGetReadCount` macro returns, in `r2`, the current number of bytes available in the FIFO that can be read. A value of 0 indicates an empty FIFO.

FIFORead

The `FIFORead` macro copies FIFO data into the specified section.

`FIFORead (theSectionName)`

`theSectionName`   The section name.

The `FIFORead` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFORead` macro takes one argument, the section name of an AIAO FIFO section. The AIAO FIFO section must be located within the same DSP module as the current section.

The `FIFORead` macro copies data to the AIAO FIFO section from the FIFO that's connected to it.

The size of AIAO is used as the number of bytes to read from the FIFO. If the FIFO empties during the read, only the actual number available will be read. The remaining bytes in the section are cleared to 0.

In the event that an underrun occurs (the FIFO does not contain enough data to fill the AIAO), a `kdspFIFOUnderrunMessage` message is sent to the FIFO's message handler if the FIFO's `kdspEnableOverUnderMessage` flag is set. Also, if the FIFO's `kdspOverUnderTaskInactive` flag is set, the owner task of the currently executing module is set inactive.

Note
Reads and writes to the buffers must occur on longword boundaries. ◆

FIFOReadN

The `FIFOReadN` macro copies the requested number of bytes of FIFO data into the specified section.

`FIFOReadN (theFIFOName, theCount)`

`theFIFOName`   The FIFO name.
`theCount`   The number of bytes to copy.

REGISTER USAGE

The `FIFOReadN` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOReadN` macro reads the specified number of bytes in `theCount` from the named FIFO to the section. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO empties during the read, only the actual number of bytes available are read. The remaining bytes in the section are cleared to 0.

FIFOReadNBuffer

The `FIFOReadNBuffer` macro copies the requested number of bytes of FIFO data into the specified section.

`FIFOReadNBuffer (theFIFOName, theCount, theBufferPtr)`

`theFIFOName`	The FIFO name.
`theCount`	The number of bytes to copy.
`theBufferPtr`	The section data is being copied to.

REGISTER USAGE

The `FIFOReadNBuffer` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOReadNBuffer` macro reads the specified number of bytes in `theCount` from the named FIFO to the section pointed to by `theBufferPtr`. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO empties during the read process, only the actual number of bytes available will be read. The remaining bytes in the section are cleared to 0.

FIFOGetWriteCount

The `FIFOGetWriteCount` macro returns the number of empty bytes available in the FIFO.

```
FIFOGetWriteCount (theFIFOName)
```

theFIFOName   The FIFO name.

REGISTER USAGE

The `FIFOGetWriteCount` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOGetWriteCount` macro returns in `r2` the current number of bytes available in the FIFO that can be written—in other words, how much empty space is available. A value of 0 indicates a full FIFO.

FIFOWrite

The `FIFOWrite` macro copies section data into the specified FIFO.

```
FIFOWrite (theSectionName)
```

theSectionName   The section name.

REGISTER USAGE

The `FIFOWrite` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `FIFOWrite` macro writes from the AIAO section to the named FIFO. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO fills up, without overrunning, the maximum number of bytes possible will be transferred.

The size of the AIAO section is used as the number of bytes to write to the FIFO.

In the event that an overrun occurs (the FIFO does not contain enough space to hold the AIAO's data), a kdspFIFOOverrunMessage message is sent to the FIFO's message handler if the FIFO's kdspEnableOverUnderMessage flag is set. Also, if the FIFO's kdspOverUnderTaskInactive flag is set, the owner task of the currently executing module is set inactive.

Note
Reads and writes to the FIFO and the buffer may occur on longword boundaries only. ◆

FIFOWriteN

The FIFOWriteN macro copies the specified number of bytes of section data into the specified FIFO.

FIFOWriteN (theFIFOName, theCount)

theFIFOName   The FIFO name.
theCount   The number of bytes to copy.

REGISTER USAGE

The FIFOWriteN macro destroys the contents of cau registers r1-r4, r15-r18, and a0-a1.

DESCRIPTION

The FIFOWriteN macro writes the specified number of bytes in theCount to the named FIFO from the section. The programmer should check r2 to make sure that the requested number of bytes was transferred. If the FIFO fills up, without overrunning, the maximum number of bytes possible will be transferred.

FIFOWriteNBuffer

The FIFOWriteNBuffer macro copies the specified number of bytes of section data into the specified FIFO.

FIFOWriteNBuffer (theFIFOName, theCount, theBufferPtr)

theFIFOName   The FIFO name.
theCount   The number of bytes to copy.
theBufferPtr   The section data is being copied from.

The `FIFOWriteNBuffer` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

The `FIFOWriteNBuffer` macro writes the specified number of bytes in `theCount` to the named FIFO from the buffer pointed to by `theBufferPtr`. The programmer should check `r2` to make sure that the requested number of bytes was transferred. If the FIFO fills up, without overrunning, the maximum number of bytes possible will be transferred.

Note
Reads and writes to the FIFO and the buffer are on longword boundaries only. ◆

GPB Manipulation Macros

The macros described in this section help you manage the GPB for a module. GPB is discussed in "Guaranteed Processing Bandwidth," in Chapter 3.

GPBElapsedCycles

The `GPBElapsedCycles` macro returns the number of DSP cycles used by this module up to the point it is called.

```
GPBElapsedCycles (theCycles)
```

theCycles   Elapsed cycles since start or reset.

The `GPBElapsedCycles` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

The `GPBElapsedCycles` macro returns the number of DSP instruction cycles that have elapsed since this module started execution. By comparing this value with the expected value returned from the `GPBExpectedCycles()` macro, a dumb lumpy algorithm can determine if it should cease processing. Dumb lumpy algorithms are discussed in "Smooth and Lumpy Algorithms," in Chapter 3.

GPBExpectedCycles

The `GPBExpectedCycles` macro returns the computed number of DSP cycles this
module is expected to need based on the supplied GPB estimate.

```
GPBExpectedCycles (theCycles)
```

theCycles   Expected cycles for this module.

REGISTER USAGE

The `GPBExpectedCycles` macro destroys the contents of `cau` registers `r1-r4`,
`r15-r18`, and `a0-a1`.

DESCRIPTION

The `GPBExpectedCycles` macro returns the expected number of DSP instruction cycles
to complete this module. This is used in conjunction with the `GPBElapsedCycles`
macro (page 235) to control the execution of a dumb lumpy algorithm.

GPBSetUseActual

The `GPBSetUseActual` macro tells the DSP operating system to use the actual GPB
required instead of the estimated value.

```
GPBSetUseActual ()
```

REGISTER USAGE

The `GPBSetUseActual` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`,
and `a0-a1`.

DESCRIPTION

The `GPBSetUseActual` macro sets the `UseActualGPB` flag for the module. This flag is
set immediately, so this routine should not be called until the module is in its worst-case
GPB usage.

Semaphore Manipulation Macros

The macros described in this section help you work with semaphores.

SemaphoreClear

The `SemaphoreClear` macro clears the specified semaphore in a locked environment.

`SemaphoreClear (theSemaphorePtr, theMask, theOldSemaphoreValue)`

theSemaphorePtr   Pointer to the semaphore.
theMask   Mask of new semaphore value.
theOldSemaphoreValue Returns the value of the old semaphore.

REGISTER USAGE

The `SemaphoreClear` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SemaphoreClear` macro locks the system bus and performs the following operation:

```
[lock the bus]
*theSemaphorePtr = ((theOldSemaphoreValue = *theSemaphorePtr) &
                         ~theMask)
[unlock the bus]
```

The value of `theSemaphorePtr` must be a `cau` register in the range `r1` through `r17` containing a physical pointer.

The value of `theMask` may be any register in the range `r1` through `r17`, or a constant.

The `SemaphoreClear` macro performs `dolock` on the bus to prevent host access and then reads the semaphore location. The old semaphore value is `AND`-combined with `NOT` of the mask and this new value is written back to the semaphore location.

RETURN VALUE

The value of `theOldSemaphoreValue` is the value of the semaphore before it was `AND`-combined with the one's-complement of the value of `theMask`.

SemaphoreSet

The `SemaphoreSet` macro sets the specified semaphore in a locked environment.

`SemaphoreSet (theSemaphorePtr, theMask, theOldSemaphoreValue)`

theSemaphorePtr   Pointer to the semaphore.

theMask   Mask of new semaphore value.

theOldSemaphoreValue Returns the value of the old semaphore.

REGISTER USAGE

The `SemaphoreSet` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SemaphoreSet` macro locks the system bus and performs the following operation:

```
[lock the bus]
*theSemaphorePtr = ((theOldSemaphoreValue = *theSemaphorePtr) |
                          theMask)
[unlock the bus]
```

The value of `theSemaphorePtr` must be a `cau` register `r1-r17` containing a physical pointer.

The `SemaphoreSet` macro performs `DoLock` on the bus to prevent host access and then reads the semaphore location. The old semaphore value is `OR`-combined with the mask and this new value is written back to the semaphore location.

RETURN VALUE

The value of `theOldSemaphoreValue` is the value of the semaphore before it was `OR`-combined with `theMask`.

Message Manipulation Macro

The `SendMessageToHost` macro helps you work with DSP messages.

SendMessageToHost

The `SendMessageToHost` macro sends a message from the module to the host using the interrupt handler.

SendMessageToHost (theDSPMessagePtr)

theDSPMessagePtr   Pointer to the message vector.

REGISTER USAGE

The `SendMessageToHost` macro destroys the contents of `cau` registers `r1-r4`, `r15-r18`, and `a0-a1`.

DESCRIPTION

The `SendMessageToHost` macro calls the `msVector` (interrupt handler) in the Real Time Manager structure that then passes the message to the interrupt handler. When used by a module to send a message to the client application the `msData[0]` through `msData[2]` fields are not defined when using this macro.

The value of `theDSPMessagePtr` must be a `cau` register `r1-r17` containing a physical pointer to a DSP message.

Note
The `msVector` field of the message must be initialized to a valid interrupt handler. Fields `msData[0]` through `msData[2]` can be used by the programmer as needed. ◆

When the Real Time Manager uses this routine to send a message to the client application `msData[0]` contains `theErrorMessage` constant. The message is sent to the interrupt vector of the owner task for the currently executing module. The owner task is then set inactive.

The `theErrorMessage` constant is a DSP message constant or a register containing a DSP message constant. The Apple-defined DSP message constants are defined in the next section.
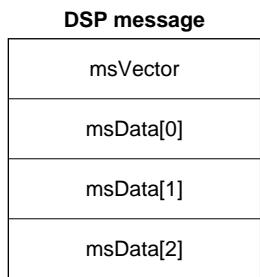
When the host interrupt vector for the task is called, a complete `DSPMessage` structure is passed on the stack containing the following information:

The owner task's interrupt vector $\rightarrow$ `msVector`

`theErrorMessage` $\rightarrow$ `msData[0]`

The task's reference number $\rightarrow$ `msData[1]`

The current module's reference number $\rightarrow$ `msData[2]`

The DSP message structure is diagrammed in Figure 5-3.

Figure 5-3   DSP message structure

DSP message

msVector
msData[0]
msData[1]
msData[2]

The corresponding routine in the Macintosh API is the `MessageActionProc` routine.

Summary of the DSP Operating System

Constants

```
====================================================================
//                  DSP MODULE/SECTION DEFINITIONS
====================================================================


--------------------------------------------------------------------
// FIFOFlags
--------------------------------------------------------------------
#define  kdspFIFOMaskAllMessages         0x00000000  // disable all
          //messages (p) priority of FIFO messages in descending order


#define  kdspFIFOEnableOverUnderMessage  0x00000001  // (3) enable
          // message when FIFO transfer causes an overrun or underrun
```

```
#define  kdspFIFOEnableFullEmptyMessage   0x00000002  // (2) enable
          // message when FIFO goes full or empty


#define  kdspFIFOEnableHighLowMessage     0x00000004  // (1) enable
          // message when FIFO goes at least half full or half empty


#define  kdspFIFOEnableLinkMessage        0x00000008  // (4) enable
          // message when FIFO's link is traversed


#define  kdspFIFOOverUnderTaskInactive    0x00000010  // if task
          // accessing FIFO causes either FIFO overrun or underrun then
          // set task inactive


#define  kdspFIFOFullEmptyTaskInactive    0x00000020  // if task
          // accessing FIFO causes either FIFO full or FIFO empty then set
          // task inactive


//----------------------------------------------------------------------
// ModuleFlags
//----------------------------------------------------------------------
#define  kdspAutoCache            0x00000000  // select auto cache model
#define  kdspDemandCache          0x00000001  // select demand cache model
#define  kdspOnChipSectionTable   0x00000004  // put section table on-chip
#define  kdspOnChipStack          0x00000020  // a stack of the specified
                                              // size will be created on-chip
#define  kdspOffChipStack         0x00000040  // a stack of the specified
                                              // size will be created off-chip


//----------------------------------------------------------------------
// GPBFlags (see DSPConstantsPrivate.h for the complete list of flags)
//----------------------------------------------------------------------
#define  kdspLumpyModule          0x00000000  // use bnEstimate
#define  kdspSmoothModule         0x00000001  // see DSPConstantsPrivate.h


//----------------------------------------------------------------------
// SectionFlags
//----------------------------------------------------------------------
// Costs the DSP one instruction to use the following flags:
#define  kdspLeaveSection         0x00000000  // do not load or save this
                                              // section
#define  kdspLoadSection          0x00000001  // load this section
#define  kdspSaveSection          0x00000002  // save this section
#define  kdspClearSection         0x00000004  // fill this section with zeroes
```

```
#define   kdspSaveOnContextSwitch 0x00000008  // save this section on context
                                              // switch


#define   kdspExternal            0x00000000  // never loaded on-chip
#define   kdspBankA               0x00000020  // load in Bank A if possible
#define   kdspBankB               0x00000040  // load in Bank B if possible
#define   kdspAnyBank             (kdspBankA | kdspBankB) // load anywhere
#define   kdspStaticSection       0x00000080  // section statically allocated
                                              // before runtime
#define   kdspFIFOSection         0x00000100  // section is a FIFO buffer
#define   kdspReservedSectionFlag0200   0x00000200  // reserved
#define   kdspLoadFIFOSection     0x00000400  // when loading convert from a
                                              // FIFO
#define   kdspSaveFIFOSection     0x00000800  // when saving convert to a FIFO


#define   kdspHIHOSection         0x00001000  // this is a HIHO section
#define   kdspReservedForToggleSectionTbl  0x00002000  // this flag holds the
                                              // kdspToggleSectionTable flag
                                              // from the module's flag


#define   kdspLoadHIHOSection     0x00004000  // when loading convert from a
                                              // HIHO
#define   kdspSaveHIHOSection     0x00008000  // when saving convert to a HIHO


// Costs the DSP two instructions to use the following flags:
#define   kdspNotIOBufferSection  0x00010000  // all cases other than below
#define   kdspInputBuffer         0x00020000  // section is an input buffer
#define   kdspOutputBuffer        0x00040000  // section is an output buffer
#define   kdspITBSection          0x00080000  // section is an intertask
                                              // buffer
#define   kdspScalableSection     0x00100000  // section size can be scaled
#define   kdspSectionAllocated    0x00200000  // reserved for use by the DSP
                                              // Manager
#define   kdspDSPUseOnly          0x00400000  // only DSP should modify this
                                              // memory


//----------------------------------------------------------------
// SectionDataTypes
//----------------------------------------------------------------
#define   kdspNonData          0x00000000  // data in section is beyond
                                           // description
#define   kdsp3200Float        0x00000001  // 3200 float
#define   kdspIEEEFloat        0x00000002  // IEEE float format
```

```
#define  kdspInt32            0x00000003  // 32bit integer
#define  kdspInt1616          0x00000004  // 16bit integer packed
#define  kdspInt8888          0x00000005  // 8bit integer packed
#define  kdspmuLaw            0x00000006  // muLaw format
#define  kdspALaw             0x00000007  // Alaw format
#define  kdspAppSpecificData  0x0000FFFF  // application-specific


//===================================================================
//                 DSP CLIENT DEFINITIONS
//===================================================================
//-------------------------------------------------------------------
// constants used by a client to specify where to insert a task
//-------------------------------------------------------------------
                                          // insert at list:
#define  kdspHeadInsert          0x00000004   // head
#define  kdspTailInsert          0x00000008   // tail
#define  kdspBeforeInsert        0x00000010   // before reference link
#define  kdspAfterInsert         0x00000020   // after reference link
#define  kdspAnyPositionInsert   kdspHeadInsert // anywhere


//-------------------------------------------------------------------
// constants for messages received by client tasks
//-------------------------------------------------------------------
#define  kdspBIOPinChangedState  0x62696f70  //'biop' (bio pin has changed
                                             // state)


// constants used for FIFO:
#define  kdspFIFOMessage         0x66000000  // 'f   ' (messages)
#define  kdspFIFOLinkMessage     0x666c6e6b  // 'lnk ' (link was traversed)
#define  kdspFIFOOverrunMessage  0x666f7672  // 'fovr' (buffer filled before
                                             // FIFO write completed)
#define  kdspFIFOUnderrunMessage 0x66756e64  //'fund' (buffer emptied before
                                             // FIFO read completed)


// constants used for FIFO buffer:
#define  kdspFIFOFullMessage     0x6666756c  // 'fful' (exactly full)
#define  kdspFIFOEmptyMessage    0x66656d70  // 'femp' (exactly empty)
#define  kdspFIFOHighMessage     0x66686967  // 'fhig' (at least half full
                                             //  but not exactly full)
#define  kdspFIFOLowMessage      0x666c6f77  // 'flow' (at least half empty
                                             //  but not exactly empty)
#define  kdspFIFOPrimeMessage    0x66707269  // 'fpri' (application-specific)
```

```
// constants used for dsp exception messages
#define   kdspExceptionMessage              0x78000000  // 'x   '
#define   kdspExceptionReset                0x78727374  // 'xrst'
#define   kdspExceptionBusError             0x78627573  // 'xbus'
#define   kdspExceptionIllegalOpcode        0x78696c6c  // 'xill'
#define   kdspExceptionReservedOne          0x78727631  // 'xrv1'
#define   kdspExceptionAddressError         0x78616472  // 'xadr'
#define   kdspExceptionDAUOverUnderflow     0x78646175  // 'xdau'
#define   kdspExceptionNotANumber           0x786e616e  // 'xnan'
#define   kdspExceptionReservedTwo          0x78727632  // 'xrv2'
#define   kdspExceptionExternalIntZero      0x78657830  // 'xex0'
#define   kdspExceptionTimer                0x7874696d  // 'xtim'
#define   kdspExceptionReservedThree        0x78727633  // 'xrv3'
#define   kdspExceptionSIOInputBufFull      0x78736962  // 'xsib'
#define   kdspExceptionSIOOutputBufEmpty    0x78736f62  // 'xsob'
#define   kdspExceptionSIODMAInputFrame     0x78736966  // 'xsif'
#define   kdspExceptionSIODMAOutputFrame    0x78736f66  // 'xsof'
#define   kdspExceptionExternalIntOne       0x78657831  // 'xex1'
#define   kdspExceptionRuntimeError         0x78657272  // 'xerr'


#define   kdspGPBMessage                    0x67000000  // 'g   ' (prefix used
                                                        // for GPB messages)
#define   kdspGPBTaskActive                 0x67616374  // 'gact' (task is
                                                        // active)
#define   kdspGPBTaskInactive               0x67696e61  // 'gina' (task is
                                                        // inactive)
#define   kdspGPBFrameOverrun               0x676f7672  // 'govr' (task was
                                                        // involved in a frame
                                                        // overrun and is now
                                                        // inactive)


#define   kdspGPBFrameSkip                  0x67736b70  // 'gskp' (task has
                                                        // skipped one or more
                                                        // frames due to a
                                                        // frame overrun)


//-----------------------------------------------------------------
// read/write permission constants for clients
//-----------------------------------------------------------------
#define   kdspWritePermission     0x0001
#define   kdspReadPermission      0x0002
#define   kdspReadWritePermission (kdspWritePermission | kdspReadPermission)
```

```
//--------------------------------------------------------------------
// constants for indexed devices
//--------------------------------------------------------------------
// CPU processor types
#define  kdsp3210         '3210'
#define  kdsp32C          '32C '


//--------------------------------------------------------------------
// constants for DSP API functions
//--------------------------------------------------------------------
#define  kevtMessageToHost      (17)
#define  kevtCacheSection       (22)
#define  kevtCopyFIFO           (23)
#define  kevtGetSectionSize     (43)
#define  kevtGPBSetUseActual    (44)
#define  kevtGPBExpectedCycles  (45)
#define  kevtGPBElapsedCycles   (46)
#define  kevtSemaphoreSet       (47)
#define  kevtSemaphoreClear     (48)
#define  kevtSetSkipCount       (50)
#define  kevtSetTaskInactive    (51)
#define  kevtBlockMove          (53)
#define  kevtNumreal-timeFrames (113)


//===================================================================
// constants for errors returned by Macintosh DSP API
//===================================================================


//--------------------------------------------------------------------
// misc errors
// the next available error code number is -733
// if you add an error, also add it to the DSPErrorStrings.r file
//--------------------------------------------------------------------
#define  kdspUnimplemented     (-692)   // feature is not implemented
#define  kdspParamErr          (-704)   // bad parameter


//--------------------------------------------------------------------
// DSPFIFO errors
//--------------------------------------------------------------------
#define  kdspNotAFIFOSection      (-700)   // not a FIFO section
#define  kdspNoMessageInterrupt   (-702)   // no message passing without a
                                           // vector
```

```
#define   kdspFIFOInUseByDSP        (-719)   // this FIFO is currently being
                                             // accessed by the DSP
#define   kdspTaskMustBeInActive  (-720)   // can only dipose of inactive
                                             // structures

#define   kdspNotFirstFIFO          (-721)   // the FIFO must be the first FIFO
                                             // in the link to wrap it


//------------------------------------------------------------------
// DSPList errors
//------------------------------------------------------------------
#define   kdspPositionIllegalErr    (-666)   // illegal DSPPosition type
#define   kdspPositionBusyErr       (-667)   // DSPPosition already occupied
#define   kdspInvalidReferenceErr   (-668)   // illegal insertion request
#define   kdspNonExistantReferenceErr(-669)  // reference element does not
                                             // exist
#define   kdspNonExistantElementErr (-670)   // deletion element not found


//------------------------------------------------------------------
// DSPMemory errors
//------------------------------------------------------------------
#define   kdspMemFullErr          (-671)   // heap full,allocation failed
#define   kdspAddressNotInZone    (-672)   // address is not in a zone
#define   kdspNilAddress          (-683)   // trying to dispose of nil
#define   kdspContainingNilAddress(-684)   // trying to dispose of (nil, nil)
#define   kdspInvalidZoneSize     (-685)   // heap size must be factor of four
#define   kdspInvalidZoneBase     (-686)   // heap base must be longword
                                             // aligned


//------------------------------------------------------------------
// DSPClient errors
//------------------------------------------------------------------
#define   kdspDeviceNotFound        (-673)   // no device matching given name
#define   kdspInvalidIndexErr       (-674)   // no device (or whatever)
                                             // matching index given
#define   kdspDeviceHasActiveClients (-675)  // can't sign out device with
                                             // clients
#define   kdspInvalidPermission     (-688)   // invalid permission for
                                             // operation
#define   kdspWritePermissionDenied (-689)   // client already exists with
                                             // write permission
#define   kdspClientNameInvalid     (-690)   // client name must be [1……31]
                                             // bytes
```

```
#define   kdspInvalidOptionSelector  (-691)  // options selector not
                                              // recognized
#define   kdspInvalidIODeviceType    (-707)  // invalid io device type,
                                              // index out of range
#define   kdspInvalidClientICON      (-717)  // an invalid ICON was passed
#define   kdspDeviceCantBeSlave      (-728)  // specified cpu device cannot
                                              // be a slave


//--------------------------------------------------------------------
// resource loader errors
//--------------------------------------------------------------------
#define   kdspModuleNotFound              (-676)   // module does not exist
#define   kdspModuleUncompatibleRate      (-677)   // incompatible frame or
                                                    // sample rate
#define   kdspUnknownDSPFResourceVersion  (-679)   // DSPF resource not
                                                    // recognized
#define   kdspUnknownDSPSectionTag        (-680)   // DSPF resource not
                                                    // recognized
#define   kdspZeroGPB                     (-714)   // module has GPB set to
                                                    // zero
#define   kdspTwoStacks                   (-731)   // cannot have both an
                                                    // on-chip and an
                                                    // off-chip stack


//--------------------------------------------------------------------
// DSPStorage errors
//--------------------------------------------------------------------
#define   kdspStorageNotFound       (-695)   // the amount and location do
                                              // not exist
#define   kdspNotEnoughOnChipMemory (-696)   // not enough on-chip memory to
                                              // allocate


//--------------------------------------------------------------------
// DSPAllocation errors
//--------------------------------------------------------------------
#define   kdspCouldNotAllocate         (-678)   // could not allocate the
                                                 // module
#define   kdspMoreThanOneModule        (-687)   // can allocate only one
                                                 // module for now
#define   kdspSectionAlreadyConnected  (-693)   // one of the sections has
                                                 // already been
                                                 // connected (i.e. FIFO
                                                 // sections)
```

```
#define  kdspSectionsDoNotMatch        (-694)    // the sections which are
                                                 // being connected either do
                                                 // not have the same size
                                                 // or the same type or are
                                                 // both input or both output
#define  kdspSectionsNotInSameModule   (-706)    // the sections that are
                                                 // being connected are not
                                                 // in the same module
#define  kdspSectionNotFound           (-697)    // could not find the
                                                 // specified section
#define  kdspBothFIFOsAllocated        (-698)    // both FIFO sections have
                                                 // already been attached to
                                                 // FIFOs
#define  kdspHadToUseOffChipMemory     (-699)    // section which was
                                                 // supposed to be on-chip
                                                 // was set up off-chip; the
                                                 // module will still run,
                                                 // but not as quickly
#define  kdspAlreadyAllocated          (-701)    // you cannot make a new ITB
                                                 // or connect sections if
                                                 // task has already been
                                                 // allocated
#define  kdspTooManyITBs               (-703)    // you cannot have more than
                                                 // MAX_MAP_SECTIONS ITBs
#define  kdspInvalidModuleAddress      (-712)    // passed in a nil module
                                                 // address
#define  kdspAIAOMustLoadOrSave        (-715)    // when connecting a FIFO to
                                                 // an AIAO,the AIAO must
                                                 // move data or the
                                                 // connection will not work
#define  kdspFIFOsNotConnected         (-716)    // you cannot insert a task
                                                 // if all the FIFOs are not
                                                 // connected to other FIFOs
#define  kdspNotAllocated              (-718)    // you must insert the task
                                                 // before you can call
                                                 // DSPGetSectionData
#define  kdspTaskNotInstalled          (-732)    // you cannot get the
                                                 // available on-chip
                                                 // memory until after the
                                                 // task is installed
```

```
//-------------------------------------------------------------------
// DSPTask errors
//-------------------------------------------------------------------
#define  kdspTaskRefNumAlreadyAllocated(-681)  // trying to reuse used
                                               // DSPTaskRefNum
#define  kdspNilMessageActionProc     (-705)   // passed in nil where
                                               // MessageActionProc
                                               // required

#define  kdspInvalidCPUDevicePtr      (-708)   // passed in nil for
                                               // DSPCPUDeviceParamBlkPtr
#define  kdspInvalidTaskRefNumPtr     (-709)   // passed in nil for the
                                               // DSPTaskRefNumPtr
#define  kdspInvalidTaskAddress       (-710)   // passed in nil for the
                                               // DSPTaskAddressPtr
#define  kdspInvalidTaskRefNum        (-711)   // passed in nil for the
                                               // DSPTaskRefNum
#define  kdspInvalidTaskName          (-713)   // length of name must
                                               // be > 0 and < 31

#define  kdspNoMasterSlaveRelationship (-722)  // tasks to be synchronized
                                               // must be on one
                                               // DSP or on DSPs that have
                                               // master-slave relationship
#define  kdspAllTasksMustBeRealTime   (-723)   // tasks to be synchronized
                                               // that are on different
                                               // DSPs must all be in
                                               // the real-time task list
#define  kdspNotEnoughTime            (-724)   // didn't have enough time
                                               // to successfully
                                               // synchronize all the tasks
#define  kdspChangingState            (-725)   // task is in the process of
                                               // going (in)active
#define  kdspAlreadyActive            (-726)   // task is already active
#define  kdspAlreadyInactive          (-727)   // task is already inactive

//-------------------------------------------------------------------
// DSPGPB errors
//-------------------------------------------------------------------
#define  kdspNotEnoughGPB  (-682)   // not enough real time for allocation
```

```
//------------------------------------------------------------------
// DSP address fixup errors
//------------------------------------------------------------------
#define  kdspOnChipPatchup       (-729)   // auto-init using address already
                                          // on-chip
#define  kdspBadRelocationType   (-730)   // internal assert - unrecognized
                                          // relocation type from linker


//==================================================================
//                    DSP REGISTER ASSIGNMENTS
//==================================================================
//
// DSP3210 Register Model
//
// REGISTER      USAGE        DESCRIPTION
// r1-r4         scratch      The contents of these registers are not
// r15-r17                    saved or restored. The contents of these
// a0-a1                      registers may be destroyed by DSP API calls.
//
// r5-r14        protected    The contents of these registers must
// a2-a3                      always be saved and restored when they
//                            are used by the programmer.
//
//                            The contents of these registers are always
//                            saved before and restored after they are
//                            used by the DSP API calls.
//
// r18           return       The DSP operating system always calls the first
//                            instruction in the entry section of
//                            each module. r18 contains the return
//                            vector to get back to the DSP operating system
//                            when the module has finished executing.
//
//                            Before jumping to the return vector, all
//                            protected registers must be restored to the
//                            same values they contained upon initial
//                            entry to the module.
//
// r19           reserved     This register is reserved by Apple. Do not alter
//                            its contents.
//
// r20           reserved     This register is reserved by Apple. Do not alter
//                            its contents.
//
```

```
// r21         stack       This register is the common stack pointer
//                         register shared by the programmer and the
//                         DSP operating system. r21 always points to the
//                         next available stack location. Therefore, r21
//                         is pre-decremented for pops and
//                         post-incremented for pushes.

// r22         reserved    This register is reserved by Apple. Do not alter
//                         its contents.
//
//====================================================================

#define  TMP   r17   // temporary register
#define  RV     r18   // return vector
#define  MXPB  r19   // reserved
#define  ERRT  r20   // reserved
#define  SP     r21   // stack pointer
#define  EVTP  r22   // reserved


//====================================================================
//                         DSP API CONSTANTS
//====================================================================

#define kLongWordSize      (4)
#define kStr31Size         (32)


// DSPMessage
#define  msVector          (0)
#define  msData            (msVector + kLongWordSize)
#define  DSPMessageSize    (msData + 3*kLongWordSize)

#define  kEVTPad           (512)
#define  Find(theSelector) EVTP + kEVTPad + theSelector*kLongWordSize

#define  OSCall(theSelector)\
     RV  = (long)  Find(theSelector);\
     RV  = (long) *RV;\
     nop;\
     call RV (RV);\
     nop
```

```
// Selectors/Options for DSP API macros
#define  kdspSelectPush            0x00000000  // selector for direction
#define  kdspSelectPop             0x00000001  // selector for direction
#define  kdspOptionSpecifyBuffer   0x00000002  // option for specifying
                                               // buffer
#define  kdspOptionSpecifyCount    0x00000004  // option for specifying
                                               // count
#define  kdspOptionNoCopyJustCount 0x00000008  // option for just counting
```

Routines

```
//===================================================================
//                  DSP PROGRAM MACROS
//===================================================================

#define  NewModule (Name, GPBFlags, ModuleFlags, EntryName)\
   @lowmod (Name, GPBFlags, ModuleFlags, EntryName)


#define  NewSection (Name, SectionFlags, SectionDataType, ModuleName)\
   @lowseg (Name, SectionFlags, 0, SectionDataType, ModuleName)
#define  AppendSection (Name)\
   .rsect"Name", TEXT


#define  NewInputFIFOAndBufferSection (Name, BufferSize, SectionDataType,
   ModuleName)\ NewSection (Name,\
                 kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                 | kdspInputBuffer | kdspFIFOSection,\
                 SectionDataType,\
                 ModuleName)\
                 BufferSize * long 0


#define NewOutputFIFOAndBufferSection (Name, BufferSize, SectionDataType,
   ModuleName)\NewSection (Name,\
                 kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                 |kdspOutputBuffer | kdspFIFOSection,\
                 SectionDataType,\
                 ModuleName)\
                 BufferSize * long 0
```

```
#define NewInputFIFOAndScalableBufferSection (Name, BufferScale,
    SectionDataType, ModuleName)\NewSection (Name,\
                    kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspInputBuffer | kdspFIFOSection
                    | kdspScalableSection,\
                    SectionDataType,\
                    ModuleName)\
                    BufferScale * long 0

#define NewOutputFIFOAndScalableBufferSection (Name, BufferScale,
    SectionDataType, ModuleName)\NewSection (Name,\
                    kdspBankB | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspOutputBuffer | kdspFIFOSection
                    | kdspScalableSection,\
                    SectionDataType,\
                    ModuleName)\
                    BufferScale * long 0

#define NewInputAIAOSection (Name, AIAOSize, SectionDataType, ModuleName)\
    NewSection (Name,\
                    kdspLoadSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspInputBuffer | kdspStaticSection,\
                    SectionDataType,\
                    ModuleName)\
                    AIAOSize * long 0

#define NewOutputPRBSection (Name, AIAOSize, SectionDataType, ModuleName)\
    NewSection (Name,\
                    kdspClearSection | kdspSaveSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspInputBuffer | kdspOutputBuffer
                    | kdspStaticSection,\
                    SectionDataType,\
                    ModuleName)\
                    AIAOSize * long 0

#define NewOutputCRBSection (Name, AIAOSize, SectionDataType, ModuleName)\
    NewSection (Name,\
                    kdspSaveSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspOutputBuffer | kdspStaticSection,\
                    SectionDataType,\
                    ModuleName)\
                    AIAOSize * long 0
```

```
#define NewScalableInputAIAOSection (Name, AIAOScale, SectionDataType,
    ModuleName)\NewSection (Name,\
                    kdspLoadSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspInputBuffer | kdspStaticSection
                    | kdspScalableSection,\
                    SectionDataType,\
                    ModuleName)
                    AIAOScale * long 0

#define NewTempScalableAIAOSection (Name, AIAOScale, SectionDataType,
    ModuleName)\NewSection (Name,\
                    kdspLeaveSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspScalableSection | kdspClearSection,\
                    SectionDataType,\
                    ModuleName)\
                    AIAOScale * long 0

#define NewScalableOutputPRBSection (Name, AIAOScale, SectionDataType,
    ModuleName)\NewSection (Name,\
                    kdspClearSection | kdspSaveSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspInputBuffer | kdspOutputBuffer
                    | kdspStaticSection | kdspScalableSection,\
                    SectionDataType,\
                    ModuleName)\
                    AIAOScale * long 0

#define NewScalableOutputCRBSection (Name, AIAOScale, SectionDataType,
    ModuleName)\NewSection (Name,\
                    kdspSaveSection | kdspBankB
                    | kdspSaveOnContextSwitch | kdspDSPUseOnly
                    | kdspOutputBuffer | kdspStaticSection
                    | kdspScalableSection,\
                    SectionDataType,\
                    ModuleName)\
                    AIAOScale * long 0

#define NewCachedProgramSection (Name, ModuleName)\
    NewSection (Name,\
                    kdspLoadSection | kdspBankA | kdspDSPUseOnly
                    | kdspNotIOBufferSection,\
                    kdspNonData,\
                    ModuleName)
```

```
#define NewExternalProgramSection (Name, ModuleName)\
   NewSection (Name,\
                   kdspLeaveSection | kdspNotIOBufferSection,\
                   kdspNonData,\
                   ModuleName)


#define NewParameterSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                   kdspExternal | kdspNotIOBufferSection,\
                   SectionDataType,\
                   ModuleName)


#define NewTableSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                   kdspLoadSection | kdspBankA | kdspDSPUseOnly
                   | kdspNotIOBufferSection,\
                   SectionDataType,\
                   ModuleName)


#define NewStateVariableSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                   kdspLoadSection | kdspSaveSection | kdspBankB
                   | kdspDSPUseOnly | kdspSaveOnContextSwitch
                   | kdspNotIOBufferSection,\
                   SectionDataType,\
                   ModuleName)


#define NewTempVariableSection (Name, SectionDataType, ModuleName)\
   NewSection (Name,\
                   kdspLeaveSection | kdspBankB
                   | kdspSaveOnContextSwitch | kdspDSPUseOnly
                   | kdspNotIOBufferSection,\
                   SectionDataType,\
                   ModuleName)
```

```
//=====================================================================
//                         DSP API MACROS
//=====================================================================
//=====================================================================
//                           GENERAL
//=====================================================================

#define  BlockMove(theSrcPtr,theDestPtr,theCount)\
     r1  = (long)  theSrcPtr;\
     r2  = (long)  theDestPtr;\
     r3  = (long)  theCount;\
     RV  = (long)  Find(kevtBlockMove);\
     RV  = (long) *RV;\
     r15 = (ushort24) 0x0004;\
     call RV (RV);\
     r16 = (ushort24) 0x0004

#define  PcLabel(theSectionLabel) \
     pc + ((theSectionLabel)-(.+8))

#define  Pop(theRegister)\
     SP = (long) SP--;\
     theRegister = (long) *SP;\
     nop

#define  Push(theRegister) *SP++ = (long) theRegister


//=====================================================================
//                      SECTION MANIPULATION
//=====================================================================

#define  CallSection (theSectionName)\
     RV  = (long) MXPB + sectn theSectionName;\
     RV  = (long) *RV;\
     nop
     call RV (RV);\
     nop

#define  GetSectionAddress(theSectionPtr,theSectionName)\
     theSectionPtr = (long) MXPB + sectn theSectionName;\
     theSectionPtr = (long) *theSectionPtr;\
     nop
```

```
#define  GetSectionSize(theSectionSize,theSectionName)\
     RV  = (long)  Find(kevtGetSectionSize);\
     RV  = (long) *RV;\
     r1  = (short) sectn theSectionName;\
     call RV (RV);\
     nop;\
     theSectionSize = (long) r2


#define  GetSectionLabel(theSectionLabelPtr,theSectionLabel)\
     theSectionLabelPtr = (long) MXPB + sectn theSectionLabel;\
     theSectionLabelPtr = (long) *theSectionLabelPtr;\
     nop;\
     theSectionLabelPtr = (long) theSectionLabelPtr + offset theSectionLabel


#define  PopSection (theSectionName)\
     RV  = (long)  Find(kevtCacheSection);\
     RV  = (long) *RV;\
     r1  = (short) sectn theSectionName;\
     call RV (RV);\
     r2  = (ushort24) kdspSelectPop


#define  PushSection (theSectionName)\
     RV  = (long)  Find(kevtCacheSection);\
     RV  = (long) *RV;\
     r1  = (short) sectn theSectionName;\
     call RV (RV);\
     r2  = (ushort24) kdspSelectPush


//===================================================================
//                       MODULE MANIPULATION
//===================================================================


#define  SetSkipCount(theSkipCount)\
     RV  = (long)  Find(kevtSetSkipCount);\
     RV  = (long) *RV;\
     r1  = (long) theSkipCount;\
     call RV (RV);\
     nop
```

```
//====================================================================
//                        TASK MANIPULATION
//====================================================================
#define  GetNumRealTimeFrames(numFrames)\
     numFrames = (long)  Find(kevtNumRealTimeFrames);\
     numFrames = (long) *numFrames;\
     nop


#define  SetTaskInactive()    OSCall(kevtSetTaskInactive)


//====================================================================
//                        FIFO MANIPULATION
//====================================================================


#define  FIFOGetReadCount(theFIFOName)\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV ;\
     r1  = (short) sectn theFIFOName;\
     call RV (RV);\
     r2  = (ushort24) (kdspOptionNoCopyJustCount | kdspSelectPush)


#define  FIFOGetWriteCount(theFIFOName)\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV;\
     r1  = (short) sectn theFIFOName;\
     call RV (RV);\
     r2  = (ushort24) (kdspOptionNoCopyJustCount | kdspSelectPop)


#define  FIFORead(theSectionName)\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV;\
     r1  = (short) sectn theSectionName;\
     call RV (RV);\
     r2  = (ushort24) (kdspSelectPush)


#define  FIFOReadN(theFIFOName,theCount)\
     r4  = (long)  theCount;\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV;\
     r1  = (short) sectn theFIFOName;\
     call RV (RV);\
     r2  = (ushort24) (kdspSelectPush | kdspOptionSpecifyCount)
```

```
#define  FIFOReadNBuffer(theFIFOName,theCount,theBufferPtr)\
     r3  = (long)  theBufferPtr;\
     r4  = (long)  theCount;\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV;\
     r1  = (short) sectn theFIFOName;\
     call RV (RV);\
     r2  = (ushort24) (kdspSelectPush | kdspOptionSpecifyCount
           | kdspOptionSpecifyBuffer)

#define  FIFOWrite(theSectionName)\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV;\
     r1  = (short) sectn theSectionName;\
     call RV (RV);\
     r2  = (ushort24) (kdspSelectPop)

#define  FIFOWriteN(theFIFOName,theCount)\
     r4  = (long)  theCount;\
     RV = (long)  Find(kevtCopyFIFO);\
     RV = (long) *RV;\
     r1  = (short) sectn theFIFOName;\
     call RV (RV);\
     r2  = (ushort24) (kdspSelectPop | kdspOptionSpecifyCount)

#define  FIFOWriteNBuffer(theFIFOName,theCount,theBufferPtr)\
     r3  = (long)  theBufferPtr;\
     r4  = (long)  theCount;\
     RV  = (long)  Find(kevtCopyFIFO);\
     RV  = (long) *RV;\
     r1  = (short) sectn theFIFOName;\
     call RV (RV);\
     r2  = (ushort24) (kdspSelectPop | kdspOptionSpecifyCount
           | kdspOptionSpecifyBuffer)


//===================================================================
//                      GPB MANIPULATION
//===================================================================

#define  GPBElapsedCycles(theCycles)\
     OSCall(kevtGPBElapsedCycles);\
     theCycles = (long) r2
```

```
#define  GPBExpectedCycles(theCycles)\
     OSCall((kevtGPBExpectedCycles);\
     theCycles = (long) r2


#define  GPBSetUseActual()    OSCall(kevtGPBSetUseActual)


//===================================================================
//                     MESSAGE MANIPULATION
//===================================================================


#define  SendMessageToHost(theDSPMessagePtr)\
     RV  = (long)  Find(kevtMessageToHost);\
     RV  = (long) *RV;\
     r1  = (long) theDSPMessagePtr;\
     call RV (RV);\
     nop


//===================================================================
//                     SEMAPHORE MANIPULATION
//===================================================================


#define  SemaphoreClear (theSemaphorePtr, theMask,
                         theOldSemaphoreValue)\
     RV  = (long)  Find(kevtSemaphoreClear);\
     RV  = (long) *RV;\
     r1  = (long) theSemaphorePtr;\
     r2  = (long) theMask;\
     call RV (RV);\
     nop;\
     theOldSemaphoreValue = (long) r3


#define  SemaphoreSet (theSemaphorePtr, theMask,
                       theOldSemaphoreValue)\
     RV  = (long)  Find(kevtSemaphoreSet);\
     RV  = (long) *RV;\
     r1  = (long) theSemaphorePtr;\
     r2  = (long) theMask;\
     call RV (RV);\
     nop;\
     theOldSemaphoreValue = (long) r3
```

Speech Synthesis and Recognition

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* explains the facilities in the Macintosh Quadra 840AV and Macintosh Centris 660AV system software for generating and understanding human speech. It contains three chapters:

- Chapter 6, "Speech Manager," describes a new Macintosh system software manager that provides a standardized way for applications to generate synthesized speech. The Speech Manager also lets an application control one or more speech synthesizers, which generate spoken sound in specific languages, intonations, and speaking styles.

- Chapter 7, "Introduction to Speech Recognition," contains a basic tutorial for the Speech Setup control panel. This control panel provides commands for controlling the speech recognition functions of the Macintosh Quadra 840AV and Macintosh Centris 660AV computers.

- Chapter 8, "Speech Rules," describes the speech rules that are built into the Macintosh Quadra 840AV and Macintosh Centris 660AV system software.

Speech Manager
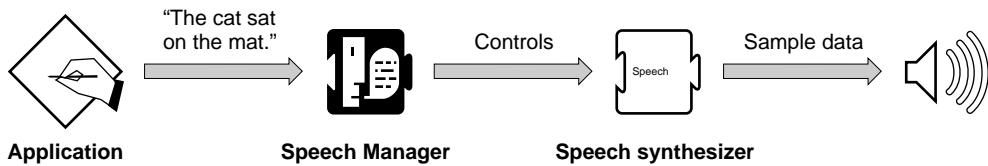
This chapter describes Apple's Speech Manager, which provides a standardized method for Macintosh applications to generate synthesized speech.

This chapter provides an overview of the Speech Manager followed by general information about generating speech from text. The necessary information and calls needed by all text-to-speech applications are given next, followed by a simple example of speech generation. More advanced calls and special-purpose routines are described last.

Speech Manager Overview

A complete system for speech synthesis consists of the elements shown in Figure 6-1.

Figure 6-1 Speech synthesis components

An application calls routines in the Speech Manager to convert character strings into speech and to adjust various parameters that affect the quality or character of the spoken output. The Speech Manager is responsible for dispatching these requests to a speech synthesizer. The speech synthesizer converts the text into sound and creates the actual audio output. Hardware support for speech generation in the Macintosh Quadra 840AV and Macintosh Centris 660AV is described in "Sound I/O," in Chapter 2.

The Apple-supplied voices, pronunciation dictionaries, and speech synthesizer may reside in a single file or in separate files. These files are clearly identifiable as Speech Manager–related files and are installed and removed by being dragged into or out of the System Folder. Additional voices can be provided by bundling the resources in the resource forks of specific applications. These resources are considered private to that particular application. It is up to the individual developers to decide whether the voice resources they provide are usable on a systemwide basis or only from within their applications.

In the first release of the Speech Manager, pronunciation dictionaries are managed entirely by the application. The application is free to store dictionaries in either the resource or the data fork of a file. The application is responsible for loading the individual dictionaries into RAM and then passing a handle to the dictionary data to the Speech Manager.

Applications that use the Speech Manager must provide their own human interface for selecting voices and/or controlling other speech characteristics. If voices are provided in separate files, the speech synthesizer developer is responsible for providing a method for

installing these resources into the System Folder or Extensions folder. The computer must be rebooted after speech synthesizers are added to or removed from the System Folder for the desired changes to be recognized.

Speech Manager Concepts

On a simple level, speech synthesis from text input is a two-stage process. First, plain-language English text is converted into **phonemic** representations for the individual words. Phonemes stand for specific sounds; for a complete explanation, see "Summary of Phonemes and Prosodic Controls," later in this chapter. The resulting sequence of phonemes is converted into audible sounds by mapping of the individual phonemes to a series of waveforms, which are sent to the sound hardware to be played.

In reality, each stage is more complicated than this description suggests. For example, during the text-to-phoneme conversion stage, number strings, abbreviations, and special symbols must be detected and converted into appropriate words before being converted into phonemes. When a sentence such as "He earned over $2,000,000 in 1990" is spoken, it would normally be preferable to say "He earned over two million dollars in nineteen-ninety" rather than "He earned over dollar-sign, two, comma, zero, zero, zero, comma, zero, zero, zero, in one, nine, nine, zero." To produce the desired spoken output automatically, knowledge of these sorts of constructions is built into the synthesizer.

The phoneme-to-sound conversion stage is also complex. Phonemes by themselves are often not sufficient to describe the way a word should be pronounced. For example, the word "object" is pronounced differently depending on whether it is used as a noun or a verb. (When it is used as a noun, the stress is placed on the first syllable. When it is used as a verb, the stress is placed on the second syllable.) In addition to stress information, phonemes must often be augmented with pitch, duration, and other information to produce intelligible, natural-sounding speech.

The speech synthesizer has many built-in rules for automatically converting text into the complex phonemic representation described above. However, there will always be words and phrases that are not pronounced the way you want. The Speech Manager allows you to provide raw phonemic information directly in order to enable very precise control over the spoken output.

By default, speech synthesizers expect input in normal language text. However, using the input mode controls of the Speech Manager, you can tell the synthesizer to process input text in raw phonemic form. By using the embedded commands described in the next section, you can even mix normal language text with phonemic text within a single string or text buffer.

See "Summary of Phonemes and Prosodic Controls," later in this chapter, for a listing of the phonemic character set and each character's interpretation.

Using the Speech Manager

This section describes the routines used to add speech synthesis features to an application. It is organized into three sections: "Getting Started" (easy), "Essential Calls—Simple and Useful" (intermediate), and "Advanced Routines."

Getting Started

If you're just getting started with text-to-speech conversion using the Speech Manager, the following routines will get you up and running with minimal effort. If you're developing an application that does not need to choose voices, use more than one channel of speech, or exercise real-time control over the synthesized speech, these may be the only routines you need.

Determining If the Speech Manager Is Available

You can find out if the Speech Manager is available with a single call to the Gestalt Manager.

Use the `Gestalt` toolbox routine and the selector `gestaltSpeechAttr` to determine whether or not the Speech Manager is available, as shown in Listing 6-1. If `Gestalt` returns `noErr`, then the parameter argument will contain a 32-bit value indicating one or more attributes of the installed Speech Manager. If the Speech Manager exists, the bit specified by `gestaltSpeechMgrPresent` is set.

Listing 6-1   Determining if the Speech Manager is available

```
Boolean SpeechAvailable (void) {
    OSErr    err;
    long     result;
    err = Gestalt(gestaltSpeechAttr, &result);
    if ((err != noErr) || !(result &
        (1 << gestaltSpeechMgrPresent)))
      return FALSE;
    else
      return TRUE;
}
```

Determining Which Version of the Speech Manager Is Running

Once you have determined that the Speech Manager is installed, you can see which version of the Speech Manager is running by calling `SpeechManagerVersion`.

SpeechManagerVersion

Returns the version of the Speech Manager installed in the system.

```
pascal NumVersion SpeechManagerVersion (void);
```

DESCRIPTION

`SpeechManagerVersion` returns the version of the Speech Manager installed in the system. This call should be used to determine the compatibility of your program with the currently installed Speech Manager.

RESULT CODES

None

Making Some Noise

The most basic operation of the Speech Manager is accomplished by using the `SpeakString` call. This call passes a specific text string to be spoken to the Speech Manager.

SpeakString

The `SpeakString` function passes a specific text string to be spoken to the Speech Manager.

```
pascal OSErr SpeakString (StringPtr s);
```

s   Text string to be spoken.

DESCRIPTION

`SpeakString` attempts to speak the Pascal-style text string contained in `myString`. Speech is produced asynchronously using the default system voice. When an application calls this function, the Speech Manager makes a copy of the passed string and creates any structures required to speak it. As soon as speaking has begun, control is returned to the application. The synthesized speech is generated transparently to the application

so that normal processing can continue while the text is being spoken. No further interaction with the Speech Manager is required at this point, and the application is free to release or purge or trash the original string.

If `SpeakString` is called while a prior string is still being spoken, the audio currently being synthesized is interrupted immediately. Conversion of the new text into speech is then initiated. If an empty (zero length) string or a null string pointer is passed to `SpeakString`, it stops the synthesis of any prior string but does not generate any additional speech.

As with all Speech Manager routines that expect text arguments, the text may contain embedded speech control commands.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough memory to speak
synthOpenFailed	−241	Could not open another speech synthesizer channel

Determining If Speaking Is Complete

Once an application starts a speech process with `SpeakString`, the next thing it will probably need to know is whether the string has been completely spoken. It can use `SpeechBusy` to determine whether or not the system is still speaking.

SpeechBusy

The `SpeechBusy` routine is useful when you want to ensure that an earlier speech request has been completed before having the system speak again.

```pascal
pascal short SpeechBusy (void);
```

DESCRIPTION

`SpeechBusy` returns the number of channels of speech that are currently synthesizing speech in the application. If you use just `SpeakString` to initiate speech, `SpeechBusy` will always return 1 as long as speech is being produced. When `SpeechBusy` returns 0, all initiated speech has finished.

RESULT CODES

None

A Simple Example

The example shown in Listing 6-2 demonstrates how to use the routines introduced in this section. It first makes sure the Speech Manager is available. Then it starts speaking a string (hard-coded in this example, but more commonly loaded from a resource) and loops, doing some screen drawing, until the string is completely spoken. This example uses the `SpeechAvailable` routine shown in Listing 6-1.

Listing 6-2   Elementary Speech Manager calls

```
OSErr err;
if (SpeechAvailable()) {
   err = SpeakString("\pThe cat sat on the mat.");
   if (err == noErr)
      while (SpeechBusy() > 0)
         CoolAnimationRoutine();
   else
      NotSoCoolAlertRoutine(err);
}
```

Essential Calls—Simple and Useful

While the routines presented in the last section are simple to use, their applicability is limited to a few basic speech scenarios. This section describes additional routines that let you work with different voices and adjust some basic characteristics of the synthesized speech.

Working With Voices

When describing a person's voice, we talk about the particular set of characteristics that help us to distinguish that person's voice from another. For example, the rate at which one speaks (slow or fast) and the average pitch (high or low) characterize a particular speaker on a crude level. In the context of the Speech Manager, a voice is the set of parameters that specify a particular quality of synthesized speech. This portion of the Speech Manager is used to determine which voices are available and to select particular voices.

Every specific voice has a unique ID associated with it, which is the primary way an application refers to it. Every voice is also associated with a `VoiceSpec` structure that is set up by the `MakeVoiceSpec` routine.

The Speech Manager provides two routines to count and step through the list of currently available voices. `CountVoices` is used to compute how many voices are available with the current system. `GetIndVoice` uses an index, starting at 1, to return information about all currently installed voices.

Use the `GetIndVoice` routine to step through the list of available voices. It will fill a `VoiceSpec` record that can be used to obtain descriptive information about the voice or to speak using that voice.

Any application that wishes to use multiple voices will probably need additional information about the available voices beyond the `VoiceSpec` structure, such as the name of the voice and perhaps what script and language each voice supports. This information might be presented to the user in a "voice picker" dialog box or voice menu, or it might be used internally by an application trying to find a voice that meets certain criteria. Applications can use the `GetVoiceDescription` routine for these purposes.

MakeVoiceSpec

To maximize compatibility with future versions of the Speech Manager, you should always use `MakeVoiceSpec` instead of setting the fields of the `VoiceSpec` structure directly.

```
pascal OSErr MakeVoiceSpec (OSType creator, OSType id, VoiceSpec
*voice);

typedef struct VoiceSpec {
   OSType   creator; // determines which synthesizer is required
   OSType   id;      // voice ID on the specified synth
} VoiceSpec;
```

Field descriptions

`creator`	The synthesizer required by your application.
`id`	Identification number for this voice.
`*voice`	Pointer to the `VoiceSpec` structure.

DESCRIPTION

Most voice management routines expect to be passed a pointer to a `VoiceSpec` structure. `MakeVoiceSpec` is a utility routine provided to facilitate the creation of `VoiceSpec` records. On return, the passed `VoiceSpec` structure contains the appropriate values.

Voices are stored in resources of type `'ttsv'` in the resource fork of Macintosh files. The Speech Manager uses the same search method as the Resource Manager, looking for voice resources in three different locations when attempting to resolve `VoiceSpec` references. It first looks in the application's resource file chain. If the specified voice is not found in an open file, it then looks in the System Folder and the Extensions folder (or in just the System Folder under System 6) for files of type `'ttsv'` (single-voice files) or `'ttsb'` (multivoice files) and in text-to-speech synthesizer component files (file type `'INIT'` or `'thng'`). Voices stored in the System Folder or Extensions folder are normally available to all applications. Voices stored in the resource fork of an application files are private to the application.

RESULT CODE

noErr   0   No error

While the determination of specific voice ID values is mostly left to synthesizer developers, the voice `creator` values are specified by Apple (they would ordinarily correspond to a developer's currently assigned creator ID). For both the `creator` and `id` fields Apple further reserves the set of `OSType` values specified entirely by space characters and lowercase letters. Apple is establishing a standard suite of voice ID values that developers can count upon being available with all speech synthesizers.

CountVoices

The `CountVoices` routine returns the number of voices available.

```
pascal OSErr CountVoices (short *voiceCount);
```

voiceCount Number of voices available to the application.

DESCRIPTION

Each time `CountVoices` is called, the Speech Manager searches for new voices. This algorithm supports dynamic installation of voices by applications or users. On return, the `voiceCount` parameter contains the number of voices available.

RESULT CODE

noErr   0   No error

GetIndVoice

The `GetIndVoice` routine returns information about a specific voice.

```
pascal OSErr GetIndVoice (short index, VoiceSpec *voice);
```

index   Index value for a specific voice.
*voice   Pointer to the `VoiceSpec` structure.

DESCRIPTION

As with all other index-based routines in the Macintosh Toolbox, an index value of 1 causes `GetIndVoice` to return information for the first voice. The order that voices are returned is not presently defined and should not be assumed. Speech Manager behavior when voice files or resources are added, removed, or modified is also presently

undefined. However, calling `CountVoices` or `GetIndVoice` with an index of 1 will force the Speech Manager to update its list of available voices. `GetIndVoice` will return a `voiceNotFound` error if the passed index value exceeds the number of available voices.

RESULT CODES

noErr	0	No error
voiceNotFound	–244	Voice resource not found

GetVoiceDescription

The `GetVoiceDescription` routine returns information about a voice beyond that provided by `GetIndVoice`.

```
pascal OSErr GetVoiceDescription (VoiceSpec *voice,
        VoiceDescription *info, long infoLength);
```

```
enum {kNeuter = 0, kMale, kFemale};// returned in gender field below
```

```
typedef struct VoiceDescription {
  long        length;        // size of structure
  VoiceSpec   voice;         // synth and ID info for voice
  long        version;       // version code for voice
  Str63       name;          // name of voice
  Str255      comment;       // additional text info about voice
  short       gender;        // neuter, male, or female
  short       age;           // approximate age in years
  short       script;        // script code of text voice can process
  short       language;      // language code of voice output speech
  short       region;        // region code of voice output speech
  long        reserved[4];   // always zero - reserved
} VoiceDescription;
```

Field descriptions

*voice	Pointer to the `VoiceSpec` structure.
*info	Pointer to structure containing parameters for the specified voice.
infoLength	Length in bytes of `info` structure.

DESCRIPTION

The Speech Manager fills out the passed `VoiceDescription` fields with the correct information for the specified voice. If a null `VoiceSpec` pointer is passed, the Speech Manager returns information for the system default voice. If the `VoiceDescription`

pointer is null, the Speech Manager simply verifies that the specified `VoiceSpec` refers to an available voice. If `VoiceSpec` does not refer to a known voice, `GetVoiceDescription` returns a `voiceNotFound` error, as shown in Listing 6-3.

To maximize compatibility with future versions of the Speech Manager, the application must pass the size of the `VoiceDescription` structure. Having the application do this ensures that the Speech Manager will never write more data into the passed structure than will fit even if additional information fields are defined in the future. On returning from `GetVoiceDescription`, the length field is set to reflect the length of data actually written by this routine.

Listing 6-3   Getting information about a voice

```
OSErr GetVoiceGender (VoiceSpec *voicePtr, short *gender) {
    OSErr       err;
    VoiceDescriptionvd;
    err = GetVoiceDescription
        (voicePtr,&vd,sizeof(VoiceDescription));
    if (err == noErr) {
        if (vd.length > offsetof(VoiceDescription,gender))
            *gender = vd.gender;
        else
            err = badStructLen;
    }
    return err;
}
```

RESULT CODES

noErr	0	No error
paramErr	−50	Parameter error
memFullErr	−108	Not enough memory to load voice into memory
voiceNotFound	−244	Voice resource not found

Managing Connections to Speech Synthesizers

Using the routines described earlier in this chapter, an application can select the voice with which to speak. The next step is to associate the selected voice with the proper speech synthesizer. This is accomplished by creating a new speech channel with the `NewSpeechChannel` routine. A speech channel is a private communication connection to the speech synthesizer, much as a file reference number is a communication channel to an open file in the Macintosh file system.

The `DisposeSpeechChannel` routine closes a speech channel when the application is finished with it and releases any resources that have been allocated to support the speech synthesizer and are no longer needed.

NewSpeechChannel

The NewSpeechChannel routine creates a new speech channel.

```
pascal OSErr NewSpeechChannel (VoiceSpec *voice,
    SpeechChannel *chan);
```

*voice   Pointer to the VoiceSpec structure.
*chan   Pointer to the new channel.

DESCRIPTION

The Speech Manager automatically locates and opens a connection to the proper synthesizer for a specified voice and sets up a channel at the location pointed to by *chan so that it is ready to speak with that voice. If a null VoiceSpec pointer is passed to NewSpeechChannel, the Speech Manager uses the current system default voice.

There is no predefined limit to the number of speech channels an application may create. However, system constraints on available RAM, processor loading, and number of available sound channels may limit the number of speech channels actually possible.

RESULT CODES

```
noErr                    0    No error
memFullErr            –108    Not enough memory to open speech channel
synthOpenFailed      –241    Could not open another speech synthesizer channel
voiceNotFound        –244    Voice resource not found
```

DisposeSpeechChannel

The DisposeSpeechChannel routine disposes of an existing speech channel.

```
pascal OSErr DisposeSpeechChannel (SpeechChannel chan);
```

chan   Specific speech channel.

DESCRIPTION

This routine disposes of an existing speech channel. Any speech channels that have not been explicitly disposed of by the application are released automatically by the Speech Manager when the application quits.

RESULT CODES

```
noErr                       0    No error
invalidComponentID      –3000    Invalid SpeechChannel parameter
```

Starting and Stopping Speech

All the remaining routines in this section require a valid speech channel to work properly. Once the application has successfully created a speech channel, it can start to speak. You use the `SpeakText` routine to begin speaking on a speech channel.

At any time during the speaking process, the application can stop the synthesizer's speech. The `StopSpeech` routine will immediately abort any speech being produced on the specified speech channel and force the channel back into an idle state.

SpeakText

The `SpeakText` routine converts designated text into speech.

```
pascal OSErr SpeakText (SpeechChannel chan, Ptr textBuf, long
    textBytes);
```

Field descriptions

chan	Specific speech channel.
textBuf	Buffer of text.
textBytes	Length of `textBuf`.

DESCRIPTION

In addition to a valid speech channel, `SpeakText` expects a pointer to the text to be spoken and the length in bytes of the text buffer. `SpeakText` will convert the given text stream into speech using the voice and control settings for that speech channel. The speech is generated asynchronously. This means that control is returned to your application before the speech has finished (probably even before it has begun). The maximum length of text buffer that can be spoken is limited only by the available RAM. However, it's generally not very friendly to force the user to listen to long uninterrupted text unless the user requests it.

If `SpeakText` is called while it is currently busy speaking the contents of a prior text buffer, it will immediately stop speaking from the prior buffer and will begin speaking from the new text buffer as soon as possible. As with `SpeakString`, described on page 267, if an empty (zero length) string or a null text buffer pointer is passed to `SpeakText`, this will have the effect of stopping the synthesis of any prior text but will not generate any additional speech.

▲ **WARNING**
With `SpeakText`, unlike with `SpeakString`, the text buffer must be locked in memory and must not move during the entire time that it is being converted into speech. This buffer is read at interrupt time, and very undesirable effects will happen if it moves or is purged from memory. ▲

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	Invalid SpeechChannel parameter

StopSpeech

The StopSpeech routine terminates speech delivery on a specified channel.

```pascal
pascal OSErr StopSpeech (SpeechChannel chan);
```

chan   Specific speech channel.

DESCRIPTION

After returning from StopSpeech, the application can safely release any text buffer that the speech synthesizer has been using. The SpeechBusy routine, described on page 268, can be used to determine if the text has been completely spoken. (In an environment with multiple speech channels, you may need to use the more advanced status routine GetSpeechInfo, described on page 286, to determine if a specific channel is still speaking.) StopSpeech can be called for an already idle channel without ill effect.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	Invalid SpeechChannel parameter

Using Basic Speech Controls

The Speech Manager provides several methods of adjusting the variables that can affect the way speech is synthesized. Although most applications probably do not need to use these advanced features, two of the speech variables, speaking rate and speaking pitch, are useful enough that a very simple way of adjusting these parameters on a channel-by-channel basis is provided. Routines are supplied that enable an application to both set and get these parameters. However, the audible effects of changing the rate and pitch of speech vary from synthesizer to synthesizer; you should test the actual results on all synthesizers with which your application may work.

Speaking rates are specified in terms of words per minute (WPM). While this unit of measurement is difficult to define in any precise way, it is generally easy to understand and use. The range of supported rates is not predefined by the Speech Manager. Each speech synthesizer provides its own range of speaking rates. Furthermore, any specific rate value will correspond to slightly different rates with different synthesizers.

Speaking pitches are defined on a musical scale that corresponds to the keys on a standard piano keyboard. By convention, pitches are represented as fixed-point values in the range from 0.000 through 100.000, where 60.000 corresponds to middle C (261.625

Hz) on a conventional piano. Pitches are represented on a logarithmic scale. On this scale, a change of +12 units corresponds to doubling the frequency, while a change of –12 units corresponds to halving the frequency. For a further discussion of pitch values, see "Getting Information About a Speech Channel," later in this chapter.

Typical voice frequencies might range from around 90 Hertz for a low-pitched male voice to perhaps 300 Hertz for a high-pitched child's voice. These frequencies correspond to pitch values of 41.526 and 53.526, respectively.

Changes in speech rate and pitch are effective immediately (as soon as the synthesizer can respond), even if they occur in the middle of a word.

SetSpeechRate

The `SetSpeechRate` routine sets the speaking rate on a designated speech channel.

```
pascal OSErr SetSpeechRate (SpeechChannel chan, Fixed rate);
```

chan   Specific speech channel.
rate   Word output speaking rate.

DESCRIPTION

The `SetSpeechRate` routine is used to adjust the speaking rate on a speech channel. The `rate` parameter is specified as a fixed-point, words-per-minute value. As a general rule of thumb, "normal" speaking rates range from around 150 WPM to around 180 WPM. It is important when working with speaking rates, however, to keep in mind that users will differ greatly in their ability to understand synthesized speech at a particular rate based upon their level of experience listening to the voice and their ability to anticipate the types of utterances they will encounter.

RESULT CODES

noErr   0   No error
invalidComponentID   –3000   Invalid `SpeechChannel` parameter

GetSpeechRate

The `GetSpeechRate` routine returns the speech rate currently active on a designated speech channel.

```
pascal OSErr GetSpeechRate (SpeechChannel chan, Fixed *rate);
```

chan   Specific speech channel.
*rate   Pointer to the current speaking rate.

DESCRIPTION

The GetSpeechRate routine is used to find out the speaking rate currently active on a speech channel.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	Invalid SpeechChannel parameter

SetSpeechPitch

The SetSpeechPitch routine sets the speaking pitch on a designated speech channel.

```
pascal OSErr SetSpeechPitch (SpeechChannel chan, Fixed pitch);
```

chan   Specific speech channel.

pitch   Frequency of voice.

DESCRIPTION

Use the SetSpeechPitch routine to change the current speaking pitch on a speech channel.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	Invalid SpeechChannel parameter

GetSpeechPitch

The GetSpeechPitch routine returns the current speaking pitch on a designated speech channel.

```
pascal OSErr GetSpeechPitch (SpeechChannel chan, Fixed *pitch);
```

Field descriptions

chan   Specific speech channel.

pitch   Frequency of voice.

DESCRIPTION

The GetSpeechPitch routine is used to find out the speaking pitch currently active on a speech channel.

RESULT CODES

```
noErr                    0     No error
invalidComponentID      –3000   Invalid SpeechChannel parameter
```

Putting It All Together

The code fragment in Listing 6-4 illustrates many of the routines introduced in this section. The example steps through the list of available voices to find the first female voice. Then it creates a new speech channel and begins speaking. While the voice is speaking, the pitch of the voice is continually adjusted around the original pitch. If the mouse button is pressed while the voice is speaking, the code halts the speech and exits. This example uses the SpeechAvailable and GetVoiceGender routines shown earlier in Listing 6-1 and Listing 6-3.

Listing 6-4   Putting it all together

```
OSErr               err;
Str255              myStr = "\pThe bat sat on my hat.";
VoiceSpec           voice;
VoiceDescription    vd;
Boolean             gotVoice = FALSE;
short               voiceCount, gender, i;
SpeechChannel       chan;
Fixed               origPitch, newPitch;

if (myStr[0] && SpeechAvailable()) {
   err = CountVoices(&voiceCount);// count the available voices
   i = 1;
   while ((i <= voiceCount) && ((err=GetIndVoice(i++, &voice))
        ==noErr))
     {
     err = GetIndVoice(i++, &voice)) == noErr;
     err = GetVoiceGender(&voice, &gender);
     if ((err == noErr) && (gender == kFemale)) {
        gotVoice = TRUE;
        break;
     }
   }
   if (gotVoice) {
      err = NewSpeechChannel(&voice, &chan);
      if (err == noErr) {
         err = GetSpeechPitch(chan, &origPitch); // cur pitch
         if (err == noErr)
            err = SpeakText(chan, &myStr[1], myStr[0]);
```

```
        i = 0;
        if (err == noErr)
            while (SpeechBusy() > 0) {
                CoolAnimationRoutine();
                newPitch = (i - 4) << 16; // fixed pitch offset
                newPitch += origPitch;
                i = (i + 1) & 7;// steps from 0 to 7 repeatedly
                err = SetSpeechPitch(chan, newPitch);
                if ((err != noErr) || Button()) {
                    err = StopSpeech(chan);
                    break;
                }
            }
        err = DisposeSpeechChannel(chan);
    }
}
if (err != noErr)
    NotSoCoolAlertRoutine(err);
```

Advanced Routines

This section describes several advanced or rarely used Speech Manager routines. You can use them to improve the quality of your application's speech.

Advanced Speech Controls

The `StopSpeech` routine, described on page 276, provides a simple way to interrupt any speech output instantly. In some situations it is preferable to be able to stop speech production at the next natural boundary, such as the next word or the end of the current sentence. `StopSpeechAt` provides that capability.

Similarly, the `PauseSpeechAt` routine causes speech to pause at a specified point in the text being spoken; the `ContinueSpeech` routine resumes speech after it has paused.

In addition to `SpeakString` and `SpeakText`, described earlier in this chapter, the Speech Manager provides a third, more general routine. `SpeakBuffer` is the low-level speech routine upon which the other two are built. `SpeakBuffer` provides greater control through the use of an additional flags parameter.

The `SpeechBusySystemWide` routine tells you if any speech is currently being synthesized in your application or elsewhere on the computer.

StopSpeechAt

The StopSpeechAt routine halts speech at a specific point in the text being spoken.

```
pascal OSErr StopSpeechAt (SpeechChannel chan, long whereToStop);

enum {
    kImmediate        = 0,
    kEndOfWord        = 1,
    kEndOfSentence    = 2
};
```

chan   Specific speech channel.

whereToStop
                Location in text at which speech is to stop.

DESCRIPTION

StopSpeechAt is used to halt the production of speech at a specified point in the text. The whereToStop argument should be set to one of the following constants:

■ The kImmediate constant stops speech output immediately.

■ The kEndOfWord constant lets speech continue until the current word has been spoken.

■ The kEndOfSentence constant lets speech continue until the end of the current sentence has been reached.

This routine returns immediately, although speech output continues until the specified point has been reached.

▲ **WARNING**
You must not release the memory associated with the current text buffer until the channel status indicates that the speech channel output is no longer busy. ▲

If the end of the input text buffer is reached before the specified stopping point, the speech synthesizer will stop at this point. Once the stopping point has been reached, the application is free to release the text buffer. Calling StopSpeechAt with whereToStop equal to kImmediate is equivalent to calling StopSpeech, described on page 276.

Contrast the StopSpeechAt routine with PauseSpeechAt, described next.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	Invalid SpeechChannel parameter

PauseSpeechAt

The `PauseSpeechAt` routine causes speech to pause at a specified point in the text being spoken.

```
pascal OSErr PauseSpeechAt (SpeechChannel chan,
      long whereToPause);

enum {
   kImmediate      = 0,
   kEndOfWord      = 1,
   kEndOfSentence  = 2
};
```

chan   Specific speech channel.

whereToPause   Location in text at which speech is to pause.

DESCRIPTION

`PauseSpeech` makes speech production pause at a specified point in the text. The `whereToPause` parameter should be set to one of these constants:

■ The `kImmediate` constant stops speech output immediately.

■ The `kEndOfWord` constant lets speech continue until the current word has been spoken.

■ The `kEndOfSentence` constant lets speech continue until the end of the current sentence has been reached.

When the specified point is reached, the speech channel enters the paused state, reflected in the channel's status. `PauseSpeechAt` returns immediately, although speech output will continue until the specified point.

If the end of the input text buffer is reached before the specified pause point, speech output pauses at the end of the buffer.

`PauseSpeechAt` differs from `StopSpeech` and `StopSpeechAt` in that a subsequent call to `ContinueSpeech`, described next, causes the contents of the current text buffer to continue being spoken.

▲ **WARNING**
While in a paused state, the last text buffer must remain available at all times and must not move. While paused, the `SpeechChannel` status indicates `outputBusy = true` and `outputPaused = true`. ▲

RESULT CODES

noErr   0   No error
invalidComponentID   –3000   Invalid `SpeechChannel` parameter

ContinueSpeech

The `ContinueSpeech` routine resumes speech after it has been halted by the `PauseSpeechAt` routine.

```
pascal OSErr ContinueSpeech (SpeechChannel chan);
```

chan   Specific speech channel.

DESCRIPTION

At any time after `PauseSpeechAt` is called, `ContinueSpeech` may be called to continue speaking from the point at which speech paused. Calling `ContinueSpeech` on a channel that is not currently in a pause state has no effect; calling it before a pause is effective cancels the pause.

RESULT CODES

```
noErr                        0     No error
invalidComponentID       –3000    Invalid SpeechChannel parameter
```

SpeakBuffer

The `SpeakBuffer` routine causes the contents of a text buffer to be spoken, using certain flags to control speech behavior.

```
pascal OSErr SpeakBuffer (SpeechChannel chan, Ptr textBuf,
                          long textBytes, long controlFlags);

enum {
    kNoEndingProsody    = 1,
    kNoSpeechInterrupt  = 2,
    kPreflightThenPause = 4
};
```

chan   Specific speech channel.

textBuf   Buffer of text.

textBytes   Length of textBuf.

controlFlags   Control flags to control speech behavior.

DESCRIPTION

When the `controlFlags` parameter is set to 0, `SpeakBuffer` behaves identically to `SpeakText`, described on page 275.

The kNoEndingProsody flag bit is used to control whether or not the speech synthesizer automatically applies **ending prosody,** the speech tone and cadence that normally occur at the end of a statement. Under normal circumstances (for example, when the flag bit is not set), ending prosody is applied to the speech when the end of the textBuf data is reached. This default behavior can be disabled by setting the kNoEndingProsody flag bit.

Some synthesizers do not speak until the kNoEndingProsody flag bit is reset, or they encounter a period in the text, or textBuf is full.

The kNoSpeechInterrupt flag bit is used to control the behavior of SpeakBuffer when called on a speech channel that is still busy. When the flag bit is not set, SpeakBuffer behaves similarly to SpeakString and SpeakText, described earlier in this chapter. Any speech currently being produced on the specified speech channel is immediately interrupted and then the new text buffer is spoken. When the kNoSpeechInterrupt flag bit is set, however, a request to speak on a channel that is still busy processing a prior text buffer will result in an error. The new buffer is ignored and the error synthNotReady is returned. If the prior text buffer has been fully processed, the new buffer is spoken normally.

The kPreflightThenPause flag bit is used to minimize the latency experienced when the speech synthesizer is attempting to speak. Ordinarily whenever a call to SpeakString, SpeakText, or SpeakBuffer is made, the speech synthesizer must perform a certain amount of initial processing before speech output is heard. This startup latency can vary from a few milliseconds to several seconds depending upon which speech synthesizer is being used. Recognizing that larger startup delays may be detrimental to certain applications, a mechanism is provided to provide the synthesizer a chance to perform any necessary computations at noncritical times. Once the computations have been completed, the speech is able to start instantly. When the kPreflightThenPause flag bit is set, the speech synthesizer will process the input text as necessary to the point where it is ready to begin producing speech output. At this point, the synthesizer will enter a paused state and return to the caller. When the application is ready to produce speech, it should call the ContinueSpeech routine to begin speaking.

RESULT CODES

noErr	0	No error
synthNotReady	–242	Speech channel is still busy speaking
invalidComponentID	–3000	Invalid SpeechChannel parameter

SpeechBusySystemWide

You can use SpeechBusySystemWide to determine if any speech is currently being synthesized in your application or elsewhere on the computer.

```
pascal short SpeechBusySystemWide (void);
```

DESCRIPTION

This routine is useful when you want to ensure that no speech is currently being produced anywhere on the Macintosh computer. `SpeechBusySystemWide` returns the total number of speech channels currently synthesizing speech on the computer, whether they were initiated by your code or by some other process executing concurrently.

RESULT CODES

None

Converting Text Into Phonemes

In some situations it is desirable to convert a text string into its equivalent phonemic representation. This may be useful during the content development process to fine-tune the pronunciation of particular words or phrases. By first converting the target phrase into phonemes, you can see what the synthesizer will try to speak. Then you need correct only the parts that would not have been spoken the way you want.

TextToPhonemes

The `TextToPhonemes` routine converts a designated text to phoneme codes.

```
pascal OSErr TextToPhonemes (SpeechChannel chan, Ptr textBuf,
                             long textBytes, Handle phonemeBuf,
                             long *phonemeBytes);
```

`chan`	Specific speech channel.
`textBuf`	Buffer of text.
`textBytes`	Length of `textBuf` in bytes.
`phonemeBuf`	Buffer of phonemes.
`*phonemeBytes`	Pointer to length of `phonemeBuf` in bytes.

DESCRIPTION

It may be useful to convert your text into phonemes during application development in order to be able to reduce the amount of memory required to speak. If your application does not require the text-to-phoneme conversion portion of the speech synthesizer, significantly less RAM may be required to speak with some synthesizers. Additionally, you may be able to use a higher quality text-to-phoneme conversion process (even one that does not work in real time) to generate precise phonemic information. This data can then be used with any speech synthesizer to produce better speech.

`TextToPhonemes` accepts a valid `SpeechChannel` parameter, a pointer to the characters to be converted into phonemes, the length of the input text buffer in bytes, an application-supplied handle into which the converted phonemes can be written, and a

length parameter. On return, the `phonemeBytes` argument is set to the number of phoneme character bytes that were written into `phonemeBuf`. The data returned by `TextToPhonemes` will correspond precisely to the phonemes that would be spoken had the input text been sent to `SpeakText` instead. All current mode settings are applied to the converted speech. No callbacks are generated while the `TextToPhonemes` routine is generating its output.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter value is invalid
`nilHandleErr`	–109	Handle argument is `nil`
`siUnknownInfoType`	–231	Feature not implemented on synthesizer
`invalidComponentID`	–3000	Invalid `SpeechChannel` parameter

Getting Information About a Speech Channel

Several additional types of information have been made available for advanced users of the Speech Manager. This information provides more detailed status information for each channel. You can get this information by calling the `GetSpeechInfo` routine. This function accepts selectors that determine the type of information you want to get.

Note

Throughout this document, there are several references to parameter values specified with fixed-point integer values (`pbas`, `pmod`, `rate`, and `volm`). Unless otherwise stated, the full range of values of the `Fixed` data type is valid. However, it is left to the individual speech synthesizer implementation to determine whether or not to use the full resolution and range of the `Fixed` data type. In the event a specified parameter value lies outside the range supported by a particular synthesizer, the synthesizer will substitute the value closest to the specified value that does lie within its performance specifications. ◆

GetSpeechInfo

The `GetSpeechInfo` routine returns information about a designated speech channel.

```
pascal OSErr GetSpeechInfo (SpeechChannel chan, OSType selector,
                            void *speechInfo);
```

```
enum {
soStatus          = 'stat',   // gets speech output status
soErrors          = 'erro',   // gets error status
soInputMode       = 'inpt',   // gets current text/phon mode
soCharacterMode   = 'char',   // gets current character mode
soNumberMode      = 'nmbr',   // gets current number mode
```

```
soRate              = 'rate',   // gets current speaking rate
soPitchBase         = 'pbas',   // gets current baseline pitch
soPitchMod          = 'pmod',   // gets current pitch modulation
soVolume            = 'volm',   // gets current speaking volume
soSynthType         = 'vers',   // gets speech synth version info
soRecentSync        = 'sync',   // gets most recent sync message info
soPhonemeSymbols    = 'phsy',   // gets phoneme symbols & ex. words
soSynthExtension    = 'xtnd'    // gets synthesizer-specific info
};
```

Field descriptions

chan   Specific speech channel.

selector   Used to specify data being requested.

*speechInfo   Pointer to an information structure.

DESCRIPTION

The following list of selectors describes the various types of information that can be obtained from the Speech Manager. The format of the information returned depends on which value is used in the selector field, as follows:

Note
For future code compatibility, use the application programming interface (API) labels instead of literal selector values. ◆

Field descriptions

stat   Gets various items of status information for the specified channel. Indicates whether any speech audio is being generated, whether or not the channel has paused, how many bytes in the input text have yet to be processed, and the phoneme code for the phoneme that is currently being generated. If inputBytesLeft is 0, the input buffer is no longer needed and can be disposed of. The API label for this selector is soStatus.

```
typedef SpeechStatusInfo *speechInfo;
typedef struct SpeechStatusInfo {
   Boolean  outputBusy;    // true = audio playing
   Boolean  outputPaused;  // true = channel paused
   long      inputBytesLeft;// bytes left to process
   short     phonemeCode;   // current phoneme code
} SpeechStatusInfo;
```

erro   Gets saved error information and clears the error registers. This selector lets you poll for various run-time errors that occur during speaking, such as the detection of badly formed embedded commands. Errors returned directly by Speech Manager routines are not reported here. The count field shows how many errors

have occurred since the last check. If `count` is 0 or 1, then `oldest` and `newest` will be the same. Otherwise, `oldest` contains the error code for the oldest unread error and `newest` contains the error code for the most recent error. Both `oldPos` and `newPos` contain the character positions of their respective errors in the original input text buffer. The API label for this selector is `soErrors`.

```
typedef SpeechErrorInfo *speechInfo;
typedef struct SpeechErrorInfo {
    short    count;   // # of errs since last check
    OSErr    oldest;  // oldest unread error
    long     oldPos;  // char position of oldest err
    OSErr    newest;  // most recent error
    long     newPos;  // char position of newest err
} SpeechErrorInfo;
```

inpt   Gets the current value of the text processing mode control. The returned value specifies whether the specified speech channel is currently in text-input mode (TEXT) or phoneme-input mode (PHON). The API label for this selector is `soInputMode`.

```
typedef OSType *speechInfo;   // TEXT or PHON
```

char   Gets the current value of the character processing mode control. The returned value specifies whether the specified speech channel is currently processing input characters in normal mode (NORM) or in literal, letter-by-letter, mode (LTRL). The API label for this selector is `soCharacterMode`.

```
typedef OSType *speechInfo;// NORM or LTRL
```

nmbr   Gets the current value of the number processing mode control. The returned value specifies whether the specified speech channel is currently processing input character digits in normal mode (NORM) or in literal, digit-by-digit, mode (LTRL). The API label for this selector is `soNumberMode`.

```
typedef OSType *speechInfo;// NORM or LTRL
```

rate   Gets the current speaking rate in words per minute on the specified channel. Speaking rates are fixed-point values. The API label for this selector is `soRate`.

```
typedef Fixed *speechInfo;
```

Note

Words per minute is a convenient, if difficult to define, way of representing speaking rate. Although there is no universally accepted definition of words per minute, it does communicate approximate information about speaking rates. Any specific rate may correspond to different rates on different synthesizers, but the two rates should be reasonably close. More importantly, doubling the rate on a particular synthesizer should halve the time needed to speak any particular utterance. ◆

pbas   Gets the current baseline pitch for the specified channel. The pitch value is a fixed-point integer that conforms to the following frequency relationship:

```
Hertz = 440.0 * 2((BasePitch - 69) / 12)
BasePitch of 1.0      ≈   9 Hertz
BasePitch of 39.5     ≈   80 Hertz
BasePitch of 45.8     ≈   115 Hertz
BasePitch of 50.4     ≈   150 Hertz
BasePitch of 100.0    ≈   2637 Hertz
```

BasePitch values are always positive numbers in the range from 1.0 through 100.0. The API label for this selector is soPitchBase.

```
typedef Fixed *speechInfo;
```

pmod   Gets the current pitch modulation range for the speech channel. Modulation values range from 0.0 through 100.0. A value of 0.0 corresponds to no modulation and means the channel will speak in a monotone. The API label for this selector is soPitchMod.

Nonzero modulation values correspond to pitch and frequency deviations according to the following formula:

```
Maximum pitch = BasePitch + PitchMod
Minimum pitch = BasePitch - PitchMod
Maximum Hertz = BaseHertz * 2(+ ModValue / 12)
Minimum Hertz = BaseHertz * 2(- ModValue / 12)
Given:
   BasePitch of 46.0        (≈ 115 Hertz),
   PitchMod of 2.0,
Then:
   Maximum pitch = 48.0    (≈131 Hertz),
   Minimum pitch = 44.0    (≈104 Hertz)
typedef Fixed *speechInfo;
```

volm   Gets the current setting of the volume control on the specified
              channel. Volumes are expressed in fixed-point units ranging from
              0.0 through 1.0. A value of 0.0 corresponds to silence, and a value of
              1.0 corresponds to the maximum possible volume. Volume units lie
              on a scale that is linear with amplitude or voltage. A doubling of
              perceived loudness corresponds to a doubling of the volume. The
              API label for this selector is `soVolume`.

```
typedef Fixed *speechInfo;
```

vers   Gets descriptive information for the type of speech synthesizer
              being used on the specified speech channel. The API label for this
              selector is `soSynthType`.

```
typedef SpeechVersionInfo *speechInfo;
typedef struct SpeechVersionInfo {
    OSType    synthType;        // always 'ttsc'
    OSType    synthSubType;     // synth flavor
    OSType    synthManufacturer;// synth creator
    long      synthFlags;       // reserved
    NumVersion synthVersion;    // synth version
} SpeechVersionInfo;
```

sync   Returns the `sync` message code for the most recently encountered
              embedded `sync` command at the audio output point. If no `sync`
              command has been encountered, 0 is returned. Refer to the section
              "Embedded Speech Commands," later in this chapter, for informa-
              tion about `sync` commands. The API label for this selector is
              `soRecentSync`.

```
typedef OSType *speechInfo;
```

phsy   Returns a list of phoneme symbols and example words defined for
              the current synthesizer. The input parameter is the address of a
              handle variable. On return, the `PhonemeDescriptor` parameter
              contains a handle to the array of phoneme definitions. Make sure to
              dispose of the handle when you are done using it. This information
              is normally used to indicate to the user the approximate sounds
              corresponding to various phonemes—an important feature in
              international speech. The API label for this selector is
              `soPhonemeSymbols`.

```
typedef PhonemeDescriptor ***speechInfo; // VAR
                                              Handle
typedef struct PhonemeInfo {
    short opcode;  // opcode for the phoneme
    Str15 phStr;   // corresponding char string
    Str31 exampleStr;   // word that shows use of
                        // phoneme
```

```
                        short hiliteStart;   // part of example word
                                             // to be hilighted as in
                        short hiliteEnd;     // TextEdit selections
                    } PhonemeInfo;
                    typedef struct PhonemeDescriptor {
                        short        phonemeCount;   // # of elements
                        PhonemeInfo thePhonemes[1]; // element list
                    } PhonemeDescriptor;
```

xtnd   This call supports a general method for extending the functionality
                    of the Speech Manager. It is used to get synthesizer-specific
                    information. The format of the returned data is determined by the
                    specific synthesizer queried. The speechInfo argument should be
                    a pointer to the proper data structure. If a particular
                    synthCreator value is not recognized by the synthesizer, the
                    command is ignored and the siUnknownInfoType code is
                    returned. The API label for this selector is soSynthExtension.

```
                    typedef SpeechXtndData *speechInfo;
                    typedef struct SpeechXtndData {
                        OSType   synthCreator; // synth creator ID
                        Byte     synthData[2]; // data TBD by synth
                    } SpeechXtndData;
```

RESULT CODES

noErr	0	No error
siUnknownInfoType	–231	Feature is not implemented on synthesizer
invalidComponentID	–3000	Invalid SpeechChannel parameter

Advanced Control Routines

The Speech Manager provides numerous control features for sophisticated developers.
These controls enable you to set various speaking parameters programmatically and
provide a rich set of callback routines that can be used to notify applications of various
conditions within the speaking process. They are extended by many speech synthesizers.

These controls are accessed with the SetSpeechInfo routine. All calls to this routine
expect a SpeechChannel parameter, a selector to indicate the desired function, and a
pointer to some data. The format of this data depends on the particular selector and is
documented in the following routine description.

SetSpeechInfo

The `SetSpeechInfo` routine sets information for a designated speech channel.

```
pascal OSErr SetSpeechInfo (SpeechChannel chan, OSType selector,
    void *speechInfo);
```

```
enum {                                  // Sets the parameter:
   soInputMode           = 'inpt',   // current text/phon mode
   soCharacterMode       = 'char',   // current character mode
   soNumberMode          = 'nmbr',   // current number mode
   soRate                = 'rate',   // current speaking rate
   soPitchBase           = 'pbas',   // current baseline pitch
   soPitchMod            = 'pmod',   // current pitch modulation
   soVolume              = 'volm',   // current speaking volume
   soCurrentVoice        = 'cvox',   // current speaking voice
   soCommandDelimiter    = 'dlim',   // command delimiters
   soReset               = 'rset',   // re channel to default state
   soCurrentA5           = 'myA5',   // app's A5 on callbacks
   soRefCon              = 'refc',   // reference constant
   soTextDoneCallBack    = 'tdcb',   // text done callback proc
   soSpeechDoneCallBack  = 'sdcb',   // end-of-speech callback proc
   soSyncCallBack        = 'sycb',   // sync command callback proc
   soErrorCallBack       = 'ercb',   // error callback proc
   soPhonemeCallBack     = 'phcb',   // phoneme callback proc
   soWordCallBack        = 'wdcb',   // word callback proc
   soSynthExtension      = 'xtnd'    // synthesizer-specific info
};
```

chan   Specific speech channel.

selector   Used to specify data being requested.

*speechInfo Pointer to an information structure.

DESCRIPTION

The following list of selectors outlines the controls available with the Speech Manager. The format of the information returned depends on which value is used in the selector field.

Note
The Speech Manager supports several callback features that can provide the sophisticated developer with a tight coupling to the speech synthesis process. However, these callbacks must be used carefully. Each is invoked from interrupt level. This means that you may not perform any operations that might cause memory to be allocated, purged, or moved. Although application global variables are also ordinarily not accessible at interrupt time, the `soCurrentA5 myA5` selector described in the following text can be used to ask the Speech Manager to point register A5 at your application's global variables prior to each callback. This makes it fairly painless to access global variables from your callback handlers. If this information worries you, don't despair. Most information available through callbacks is also available through a `GetSpeechInfo` call. These calls are more friendly and do not come with the constraints imposed upon callback code. The only drawback is that if you do not poll the information you are interested in often enough, you may miss some of the changes in your speech channel's status. ◆

Field descriptions

inpt   Sets the current value of the text processing mode control. The passed value specifies whether the speech channel should be in text-input mode (`TEXT`) or phoneme-input mode (`PHON`). Input mode changes take effect as soon as possible; however, the precise latency is dependent upon the specific speech synthesizer. The API label for this selector is `soInputMode`.

```
typedef OSType *speechInfo; // TEXT or PHON
```

char   Sets the current value of the character processing mode control. The passed value specifies whether the speech channel should be in normal character processing mode (`NORM`) or literal, letter-by-letter, mode (`LTRL`). Character mode changes take effect as soon as possible; however, the precise latency is dependent upon the specific speech synthesizer. The API label for this selector is `soCharacterMode`.

```
typedef OSType *speechInfo; // NORM or LTRL
```

nmbr   Sets the current value of the number processing mode control. The passed value specifies whether the specified speech channel should be in normal number processing mode (`NORM`) or in literal, digit-by-digit, mode (`LTRL`). The number mode changes take effect as soon as possible. However, the precise latency is dependent upon the specific speech synthesizer. The API label for this selector is `soNumberMode`.

```
typedef OSType *speechInfo; // NORM or LTRL
```

rate   Sets the speaking rate in words per minute on the specified channel.
                  Speaking rates are fixed-point values. All values are valid; however,
                  specific synthesizers will not necessarily be able to speak at all
                  possible rates. The API label for this selector is `soRate`.

```
typedef Fixed *speechInfo;
```

pbas   Changes the current baseline pitch for the specified channel. The
                  pitch value is a fixed-point integer that conforms to the following
                  frequency relationship:

```
Hertz = 440.0 * 2((BasePitch - 69) / 12)
BasePitch of 1.0     ≈   9 Hertz
BasePitch of 39.5    ≈   80 Hertz
BasePitch of 45.8    ≈   115 Hertz
BasePitch of 50.4    ≈   150 Hertz
BasePitch of 100.0   ≈   2637 Hertz
```

                  `BasePitch` values are always positive numbers in the range from
                  1.0 through 100.0.

```
typedef Fixed *speechInfo;
```

                  The API label for this selector is `soPitchBase`.

pmod   Changes the current pitch modulation range for the speech channel.
                  Modulation values range from 0.0 through 100.0. A value of 0.0
                  corresponds to no modulation and means the channel will speak in
                  a monotone. Nonzero modulation values correspond to pitch and
                  frequency deviations according to the following formula:

```
Maximum pitch  = BasePitch + PitchMod
Minimum pitch  = BasePitch - PitchMod
Maximum Hertz  = BaseHertz * 2(+ ModValue / 12)
Minimum Hertz  = BaseHertz * 2(- ModValue / 12)
Given:
   BasePitch of 46.0    (≈115 Hertz),
   PitchMod of 2.0,
Then:
   Maximum pitch = 48.0 (≈131 Hertz),
   Minimum pitch = 46.0 (≈104 Hertz)
```

```
typedef Fixed *speechInfo;
```

                  The API label for this selector is `soPitchMod`.

volm   Changes the current speaking volume on the specified channel.
                  Volumes are expressed in fixed-point units ranging from 0.0
                  through 1.0. A value of 0.0 corresponds to silence, and a value of 1.0
                  corresponds to the maximum possible volume. Volume units lie on

a scale that is linear with amplitude or voltage. A doubling of perceived loudness corresponds to a doubling of the volume. The API label for this selector is `soVolume`.

```
typedef Fixed *speechInfo;
```

cvox   Changes the current voice on the current speech channel to the specified voice. Note that this control call will return an `incompatibleVoice` error if the specified voice is incompatible with the speech synthesizer associated with the speech channel. The API label for this selector is `soCurrentVoice`.

```
typedef VoiceSpec *speechInfo;
```

dlim   Sets the delimiter character strings for embedded commands. The start of an embedded command is determined by comparing the input characters to the start-command delimiter string. Likewise, the end of a command is determined by comparing the input characters to the end-command delimiter string. Command delimiter strings are either 1 or 2 bytes in length. If a single byte delimiter is desired, it should be followed by a null (0) byte. Delimiter characters must come from the set of printable characters. If the delimiter strings are empty, this will have the effect of disabling embedded command processing. Care must be taken not to choose delimiter strings that might occur naturally in the text to be spoken. The API label for this selector is `soCommandDelimiter`.

```
typedef  DelimiterInfo *speechInfo;
typedef  struct DelimiterInfo {
Byte     startDelimiter[2];   // defaults to "[["
Byte     endDelimiter[2];     // defaults to "]]"
} DelimiterInfo;
```

rset   Resets the speech channel to its default states. The `speechInfo` parameter should be set to 0. Specific synthesizers may provide other reset capabilities. The API label for this selector is `soReset`.

```
typedef long *speechInfo;
```

myA5   An application uses this selector to request that the speech synthesizer set up an A5 world prior to all callbacks. In order for an application to access any of its global data, it is necessary that register A5 contain the correct value, since all global variables are referenced relative to register A5. If you pass a non-null value in the `speechInfo` parameter, the speech synthesizer will set register A5 to this value just before it calls one of your callback routines. The A5 register is restored to its original value when your callback routine returns. The API label for this selector is `soCurrentA5`.

```
typedef Ptr speechInfo;
```

A typical application would make the call to `SetSpeechInfo` with code like the following:

```
myA5 = SetCurrentA5();
err = SetSpeechInfo (mySpeechChannel, soCurrentA5,
    myA5);
```

refc   Sets the reference constant associated with the specified channel. All callbacks generated for this channel will return this reference constant for use by the application. The application can use this value any way it wants to. The API label for this selector is `soRefCon`.

```
typedef long *speechInfo;
```

tdcb   Enables the callback that signals that text input processing is done. Your callback routine is invoked when the current buffer of input text has been processed and is no longer needed by the speech synthesizer. This callback does not indicate that the synthesizer is finished speaking the text (see the `sdcb` callback description, next); it merely indicates that the input text has been fully processed and is no longer needed by the speech synthesizer. This callback can be disabled by passing a null `ProcPtr` in the `speechInfo` parameter. When your callback routine is invoked, you have two options. If you set the `nextBuf`, `byteLen`, and `controlFlags` variables before returning, you will enable the speech synthesizer to continue speaking without any interruption in the output. If you set the `nextBuf` parameter to null, you are indicating that you have no more text to speak. The `controlFlags` parameter is defined as in `SpeakBuffer`. The API label for this selector is `soTextDoneCallBack`.

```
typedef Ptr speechInfo;
pascal void MyInputDoneCallback (SpeechChannel
    chan, long refCon, Ptr *nextBuf,
    long *byteLen, long *controlFlags);
```

sdcb   Enables an end-of-speech callback. Your callback routine is called whenever an input text stream has been completely processed and spoken. When your callback routine is invoked, you can be certain that the speech channel is now idle and no audio is being generated. This callback can be disabled by passing a null `ProcPtr` in the `speechInfo` parameter. The API label for this selector is `soSpeechDoneCallBack`.

```
typedef Ptr speechInfo;
pascal void MyEndOfSpeechCallback (SpeechChannel
    chan, long refCon);
```

sycb   Enables the `sync` command callback. Your callback routine is
                    invoked when the text following a `sync` embedded command is
                    about to be spoken. This callback can be disabled by passing a null
                    `ProcPtr` in the `speechInfo` parameter. See "Embedded Speech
                    Commands," later in this chapter, for a description of how to use
                    `sync` commands. The API label for this selector is
                    `soSyncCallBack`.

```
typedef Ptr speechInfo;
pascal void MySyncCommandCallback (SpeechChannel
     chan, long refCon, OSType syncMessage);
```

ercb   Enables error callbacks. Your callback routine is called whenever an
                    error occurs during the processing of an input text stream. Errors
                    can result from syntax problems in the input text, insufficient
                    CPU processing speed (such as an audio data underrun), or other
                    conditions that may arise during the speech conversion process.
                    If error callbacks have not been enabled, when an error condition
                    is detected, the Speech Manager will save its value. The error
                    codes can then be read using the `GetSpeechInfo` status selector
                    `soErrors` (`erro`). The error callback can be disabled by passing a
                    null `ProcPtr` in the `speechInfo` parameter. The API label for this
                    selector is `soErrorCallBack`.

```
typedef Ptr speechInfo;
pascal void MyErrorCallback (SpeechChannel chan,
     long refCon, OSErr error, long bytePos);
```

phcb   Enables phoneme callbacks. Your callback routine is invoked for
                    each phoneme generated by the speech synthesizer just before the
                    phoneme is actually spoken. This callback can be disabled by
                    passing a null `ProcPtr` in the `speechInfo` parameter. The API
                    label for this selector is `soPhonemeCallBack`.

```
typedef Ptr speechInfo;
pascal void MyPhonemeCallBack (SpeechChannel chan,
     long refCon, short phonemeOpcode);
```

wdcb   Enables word callbacks. Your callback routine is invoked for each
                    word generated by the speech synthesizer just before the word is
                    actually spoken. This callback can be disabled by passing a `nil`
                    `ProcPtr` in the `speechInfo` parameter. The API label for this
                    selector is `soWordCallBack`.

```
typedef Ptr speechInfo;
pascal void MyWordCallback (SpeechChannel chan,
     long refCon, long wordPos, short wordLen);
```

xtnd   This call supports a general method for extending the functionality of the Speech Manager. It is used to set synthesizer-specific information. The `speechInfo` argument should be a pointer to the appropriate data structure. If a particular `synthCreator` value is not recognized by the synthesizer, the command is ignored and an `siUnknownInfoType` code is returned. The API label for this selector is `soSynthExtension`.

```
typedef SpeechXtndData *speechInfo;
typedef struct SpeechXtndData {
    OSType    synthCreator; // synth creator ID
    Byte      synthData[2]; // data TBD by synth
} SpeechXtndData;
```

RESULT CODES

noErr	0	No error
paramErr	–50	Parameter value is invalid
siUnknownInfoType	–231	Feature is not implemented on synthesizer
incompatibleVoice	–245	Specified voice cannot be used with synthesizer
invalidComponentID	–3000	Invalid `SpeechChannel` parameter

Application-Defined Pronunciation Dictionaries

No matter how sophisticated a speech synthesis system is, there will always be words that it does not automatically pronounce correctly. The clearest instance of words that are often mispronounced is the class of proper names (names of people, place names, and so on).

One way to get around this fundamental limitation is to use a dictionary of pronunciations. Whenever a speech synthesizer needs to determine the proper phonemic representation for a particular word, it first looks for the word in its dictionaries. Pronunciation dictionary entries contain information that enables precise conversion between text and the correct phoneme codes. They also provide stress, intonation, and other information to help speech synthesizers produce more natural speech. If the word in question is found in the dictionary, then the synthesizer uses the information from the dictionary entry rather than relying on its own letter-to-sound rules. The use of phonemes is described in "Summary of Phonemes and Prosodic Controls," later in this chapter.

The Speech Manager word storage format provides high-quality data that is interchangeable between speech synthesizers. The Speech Manager also uses an easily extensible dictionary structure that does not affect the usability of existing dictionaries.

It is assumed that application-defined pronunciation dictionaries will reside in RAM when in use. The run-time structure of dictionary data presumably depends on the specific needs of particular speech synthesizers and will therefore differ from the structure of the dictionaries as stored on disk.

Associating a Dictionary With a Speech Channel

The following routine can be used to associate an application-defined pronunciation dictionary with a particular speech channel.

UseDictionary

The `UseDictionary` routine associates a designated dictionary with a specific speech channel.

```
pascal OSErr UseDictionary (SpeechChannel chan, Handle
        dictionary);
```

chan   Specific speech channel.

dictionary Handle to the specified dictionary.

DESCRIPTION

The speech synthesizer will attempt to use the dictionary data pointed to by the `dictionary` handle argument to augment the built-in pronunciation rules on the specified speech channel. The synthesizer will use whatever elements of the dictionary resource it considers useful to the speech conversion process. After returning from `UseDictionary`, the caller is free to release any storage allocated for the dictionary handle. The search order for application-provided dictionaries is last in, first searched.

All details of how an application-provided dictionary is represented within the speech synthesizer are dependent on the specific synthesizer implementation and are totally private to the synthesizer.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough memory to use new dictionary
badDictFormat	–246	Format problem with pronunciation dictionary
invalidComponentID	–3000	Invalid `SpeechChannel` parameter

Pronunciation Dictionary Data Format

Each application-defined pronunciation dictionary is implemented as a single resource of type `'dict'`. To read the dictionary contents, the system first reads the resource into memory using Resource Manager routines.

An application dictionary contains the following information:

total byte length	(long)	(Length is all-inclusive)
atom type	(long)	
format version	(long)	
script code	(short)	
language code	(short)	
region code	(short)	
date last modified	(long)	(Seconds since January 1, 1904)
reserved (4)	(long)	
entry count	(long)	
list of entries		

The currently defined atom type is

`'dict'`	→	Dictionary

Each entry consists of the following:

entry byte length	(short)	(Length is all-inclusive)
entry type	(short)	
field count	(short)	
list of fields		

The currently defined entry types are the following:

0x00	→	Null entry
0x01 to 0x20	→	Reserved
0x21	→	Pronunciation entry
0x22	→	Abbreviation entry

Each field consists of the following:

field byte length	(short)	(Length is all-inclusive minus padding)
field type	(short)	
field data	(char[])	(Data is padded to word boundary)

The currently defined field types are the following:

0x00	→	Null field
0x01 to 0x20	→	Reserved
0x21	→	Word represented in textual format.

| 0x22 | → | Phonemic pronunciation including a complete set of syllable, lexical stress, word prominence, and prosodic markers represented in textual format |
| 0x23 | → | Part-of-speech code |

Creating and Editing Dictionaries

There is no built-in support for creating and editing speech dictionaries. You can create dictionary resources using any of the available resource editing tools such as the MPW Rez tool or ResEdit. Of course, you can also fairly easily develop routines to edit the dictionary structure from within the application. At the present time, no assumption should be made that the entries in a dictionary are stored in sorted order.

Advanced Voice Information Routine

Ordinarily, an application should need to use only the `GetVoiceDescription` routine to access information about a particular voice. Occasionally, however, it may be necessary to obtain more detailed information by using the `GetVoiceInfo` routine.

GetVoiceInfo

The `GetVoiceInfo` routine returns information about a specified voice channel beyond that obtainable through the `GetVoiceDescription` routine.

```
pascal OSErr GetVoiceInfo (VoiceSpec *voice, OSType selector,
    void *voiceInfo);

    typedef VoiceDescription *voiceInfo;
    typedef VoiceFileInfo *voiceInfo;
    typedef struct VoiceFileInfo {
    FSSpec    fileSpec;    // vol, dir, name info for voice file
    short     resID;       // resource ID of voice in the file
} VoiceFileInfo;
enum {
    soVoiceDescription   = 'info',   // gets basic voice info
    soVoiceFile          = 'fref'    // gets voice file ref info
};
```

`*voice`   Specific speech channel.

`selector`   Used to specify data being requested.

`*voiceInfo` Pointer to an information structure.

DESCRIPTION

This function accepts selectors that determine the type of information you want to get. The format of the information returned depends on which value is used in the selector field, as follows:

Field descriptions

info
Gets basic information for the specified voice. The structure returned is functionally equivalent to the `VoiceDescription` data structure in `GetVoiceDescription`, described on page 272. To maximize compatibility with future versions of the Speech Manager, the application must set the `length` field of the `VoiceDescription` structure to the size of the existing record before calling `GetVoiceInfo`, which then returns the size of the new record.

fref
Gets file reference information for specified voice; normally only used by speech synthesizers to access voice disk files directly.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough memory to load voice into memory
voiceNotFound	–244	Voice resource not found

Embedded Speech Commands

This section describes how you can insert commands directly into the input text to control or modify the spoken output. When processing input text data, speech synthesizers look for special sequences of characters called **delimiters.** These character sequences are usually defined to be unusual pairings of printable characters that would not normally appear in the text. When a begin command delimiter string is encountered in the text, the following characters are assumed to contain one or more commands. The synthesizer will attempt to parse and process these commands until an end command delimiter string is encountered.

Embedded Speech Command Syntax

By default, the begin command and end command delimiters are defined to be [[and]]. The syntax of embedded command blocks is given below, according to these rules:

■ Items enclosed in angle brackets (< and >) represent logical units that are either defined further below or are atomic units that should be self-explanatory.

■ Items enclosed in brackets are optional.

■ Items followed by an ellipsis (...) may be repeated one or more times.

■ For items separated by a vertical bar (|), any one of the listed items may be used.

■ Multiple space characters between tokens may be used if desired.

■ Multiple commands should be separated by semicolons.

All other characters that are not enclosed between angle brackets must be entered literally. There is no limit to the number of commands that can be included in a single command block.

Here is the embedded command syntax structure:

Identifier	Syntax
CommandBlock	*<BeginDelimiter> <CommandList> <EndDelimiter>*
BeginDelimiter	*<String1>* **|** *<String2>*
EndDelimiter	*<String1>* **|** *<String2>*
CommandList	*<Command>* **[; *<Command>*]...**
Command	*<CommandSelector>* **[***parameter***]...**
CommandSelector	*<OSType>*
Parameter	*<OSType>* **|** *<String1>* **|** *<String2>* **|** *<StringN>* **|** *<FixedPointValue>* **|***<32BitValue>* **|** *<16BitValue>* **|** *<8BitValue>*
String1	*<QuoteChar> <Character> <QuoteChar>*
String2	*<QuoteChar> <Character> <Character> <QuoteChar>*
StringN	*<QuoteChar>* **[*<Character>*]...** *<QuoteChar>*
QuoteChar	**" | '**
OSType	*<4 character pattern (for example, RATE, vers, aBcD)>*
Character	*<Any printable character (for example, A, b, *, #, x)>*
FixedPointValue	*<Decimal number: $0.0000 \leq N \leq 65535.9999$>*
32BitValue	*<OSType>* **|** *<LongInt>* **|** *<HexLongInt>*
16BitValue	*<Integer>* **|** *<HexInteger>*
8BitValue	*<Byte>* **|** *<HexByte>*
LongInt	*<Decimal number: $0 \leq N \leq 4294967295$>*
HexLongInt	*<Hex number: $0x00000000 \leq N \leq 0xFFFFFFFF$>*
Integer	*<Decimal number: $0 \leq N \leq 65535$>*
HexInteger	*<Hex number: $0x0000 \leq N \leq 0xFFFF$>*
Byte	*<Decimal number: $0 \leq N \leq 255$>*
HexByte	*<Hex number: $0x00 \leq N \leq 0xFF$>*

Embedded Speech Command Set

Table 6-1 outlines the set of currently defined embedded speech commands.

Table 6-1 Embedded speech commands

Command	Selector	Command syntax and description
Version	`vers`	`vers` *<Version>* *Version*`::=` *<32BitValue>* This command informs the synthesizer of the format version that will be used in subsequent commands. This command is optional but is highly recommended. The current version is 1.
Delimiter	`dlim`	`dlim` *<BeginDelimiter>* *<EndDelimiter>* The delimiter command specifies the character sequences that mark the beginning and end of all subsequent commands. The new delimiters take effect at the end of the current command block. If the delimiter strings are empty, an error is generated. (Contrast this behavior with the `dlim` function of `SetSpeechInfo`.)
Comment	`cmnt`	`cmnt` [*Character*]... This command enables a developer to insert a comment into a text stream for documentation purposes. Note that all characters following the `cmnt` selector up to the *<EndDelimiter>* are part of the comment.
Reset	`rset`	`rset` *<32BitValue>* The reset command will reset the speech channel's settings back to the default values. The parameter should be set to 0.
Baseline pitch	`pbas`	`pbas` [`+`	`-`] *<Pitch>* *Pitch* `::=` *<FixedPointValue>* The baseline pitch command changes the current pitch for the speech channel. The pitch value is a fixed-point number in the range 1.0 through 100.0 that conforms to the frequency relationship $\text{Hertz} = 440.0 * 2^{((\text{Pitch} - 69) / 12)}$ If the pitch number is preceded by a + or − character, the baseline pitch is adjusted relative to its current value. Pitch values are always positive numbers. For further details, see the description of `SetSpeechInfo` on page 292.

continued

Table 6-1   Embedded speech commands (continued)

Command	Selector	Command syntax and description
Pitch modulation	pmod	pmod [+ | –] *<ModulationDepth>* *ModulationDepth* ::= *<FixedPointValue>* The pitch modulation command changes the modulation range for the speech channel. The modulation value is a fixed-point number in the range 0.0 through 100.0 that conforms to the following pitch and frequency relationships: Maximum pitch = BasePitch + PitchMod Minimum pitch = BasePitch – PitchMod Maximum Hertz = BaseHertz * $2^{(+\,\text{ModValue}\,/\,12)}$ Minimum Hertz = BaseHertz * $2^{(-\,\text{ModValue}\,/\,12)}$ A value of 0.0 corresponds to no modulation and will cause the speech channel to speak in a monotone. If the modulation depth number is preceded by a + or – character, the pitch modulation is adjusted relative to its current value. For further details, see the description of SetSpeechInfo on page 292.
Speaking rate	rate	rate [+ | –] *<WordsPerMinute>* *WordsPerMinute* ::= *<FixedPointValue>* The speaking rate command sets the speaking rate in words per minute on the speech channel. If the rate value is preceded by a + or – character, the speaking rate is adjusted relative to its current value.
Volume	volm	volm [+ | –] *<Volume>* *Volume* ::= *<FixedPointValue>* The volume command changes the speaking volume on the speech channel. Volumes are expressed in fixed-point units ranging from 0.0 through 1.0. A value of 0.0 corresponds to silence, and a value of 1.0 corresponds to the maximum possible volume. Volume units lie on a scale that is linear with amplitude or voltage. A doubling of perceived loudness corresponds to a doubling of the volume.
Sync	sync	sync *<SyncMessage>* *SyncMessage* ::= *<32BitValue>* The sync command causes a callback to the application's sync command callback routine. The callback is made when the audio corresponding to the next word begins to sound. The callback routine is passed the *SyncMessage* value from the command. If the callback routine has not been defined, the command is ignored. For further details, see the description of SetSpeechInfo on page 292.

continued

Table 6-1   Embedded speech commands (continued)

Command	Selector	Command syntax and description
Input mode	`inpt`	`inpt TX	TEXT	PH	PHON` This command switches the input processing mode to either normal text mode or raw phoneme mode.
Character mode	`char`	`char NORM	LTRL` The character mode command sets the word speaking mode of the speech synthesizer. When `NORM` mode is selected, the synthesizer attempts to automatically convert words into speech. This is the most basic function of the text-to-speech synthesizer. When `LTRL` mode is selected, the synthesizer speaks every word, number, and symbol letter by letter. Embedded command processing continues to function normally, however.
Number mode	`nmbr`	`nmbr NORM	LTRL` The number mode command sets the number speaking mode of the speech synthesizer. When `NORM` mode is selected, the synthesizer attempts to automatically speak numeric strings as intelligently as possible. When `LTRL` mode is selected, numeric strings are spoken digit by digit.
Silence	`slnc`	`slnc <Milliseconds>` *Milliseconds* ::= *<32BitValue>* The silence command causes the synthesizer to generate silence for the specified amount of time.
Emphasis	`emph`	`emph +	–` The emphasis command causes the next word to be spoken with either greater emphasis or less emphasis than would normally be used. Using + will force added emphasis, while using – will force reduced emphasis.
Synthesizer-Specific	`xtnd`	`xtnd <SynthCreator> [parameter]` *SynthCreator* ::= *<OSType>* The extension command enables synthesizer-specific commands to be embedded in the input text stream. The format of the data following *SynthCreator* is entirely dependent on the synthesizer being used. If a particular *SynthCreator* is not recognized by the synthesizer, the command is ignored but no error is generated.

Synthesizers often support embedded commands that extend the set given in Table 6-1.

Embedded Speech Command Error Reporting

While embedded speech commands are being processed, several types of errors may be detected and reported to your application. If you have set up an error callback handler with the `soErrorCallBack` selector of the `SetSpeechInfo` routine (described on page 292), you will be notified once for every error that is detected. If you have not enabled error callbacks, you can still obtain information about the errors encountered by calling `GetSpeechInfo` with the `soErrors` selector (described on page 286). The following errors are detected during processing of embedded speech commands:

`badParmVal`	–245	Parameter value is invalid
`badCmdText`	–246	Embedded command syntax or parameter problem
`unimplCmd`	–247	Embedded command is not implemented on synthesizer
`unimplMsg`	–248	Unimplemented message
`badVoiceID`	–250	Specified voice has not been preloaded
`badParmCount`	–252	Incorrect number of embedded command arguments

Summary of Phonemes and Prosodic Controls

This section summarizes the phonemes and prosodic controls used by American English speech synthesizers.

Phoneme Set

Table 6-2 summarizes the set of standard phonemes recognized by American English speech synthesizers.

In this description, it is assumed that specific rules and markers apply only to general American English. Other languages and dialects require different phoneme inventories. Phonemes divide into two groups: vowels and consonants. All vowel symbols are uppercase pairs of letters. For consonants, in cases in which the correspondence between the consonant and its symbol is apparent, the symbol is that lowercase consonant; in other cases, the symbol is an uppercase consonant. Within the example words, the individual sounds being exemplified appear in boldface.

Table 6-2   American English phoneme symbols

Symbol	Example	Opcode	Symbol	Example	Opcode
AE	b**a**t	2	b	**b**in	18
EY	b**ai**t	3	C	**ch**in	19
AO	c**au**ght	4	d	**d**in	20
AX	**a**bout	5	D	**th**em	21
IY	b**ee**t	6	f	**f**in	22
EH	b**e**t	7	g	**g**ain	23
IH	b**i**t	8	h	**h**at	24
AY	b**i**te	9	J	**g**in	25
IX	ros**e**s	10	k	**k**in	26
AA	c**o**t	11	l	**l**imb	27
UW	b**oo**t	12	m	**m**at	28
UH	b**oo**k	13	n	**n**at	29
UX	b**u**d	14	N	ta**ng**	30
OW	b**oa**t	15	p	**p**in	31
AW	b**ou**t	16	r	**r**an	32
OY	b**oy**	17	s	**s**in	33
			S	**sh**in	34
			t	**t**in	35
			T	**th**in	36
			v	**v**an	37
			w	**w**et	38
			y	**y**et	39
%	silence	0	z	**z**en	40
@	breath intake	1	Z	**g**enre	41

Note

The "silence" phoneme (%) and the "breath" phoneme (@) may be lengthened or shortened like any other phoneme. ◆

Prosodic Controls

The symbols listed in Table 6-3 are recognized as modifiers to the basic phonemes described in the preceding section. They can be used to more precisely control the quality of speech that is described in terms of raw phonemes.

Table 6-3 Prosodic control symbols

Type	Symbol		Description of effect	
Lexical stress:			*Marks stress within a word*	
Primary stress	1	anticipation	AEnt2IHsIXp1EYSAXn ("anticipation")	
Secondary stress	2	anticipation		
Syllable breaks:			*Marks syllable breaks within a word*	
Syllable mark	=	(equal)	AEn=t2IH=sIX=p1EY=SAXn ("anticipation")	
Word prominence:			*Marks the beginning of a word (required)*	
Unstressed	~	(asciitilde)	Used for words with minimal information content	
Normal stress	_	(underscore)	Used for information-bearing words	
Emphatic stress	+	(plus)	Special emphasis for a word	
Prosodic:			*Placed before the affected phoneme*	
Pitch rise	/	(slash)	Pitch will rise on the following phoneme	
Pitch fall	\	(backslash)	Pitch will fall on the following phoneme	
Lengthen phoneme	>	(greater)	Lengthens the duration of the following phoneme	
Shorten phoneme	<	(less)	Shortens the duration of the following phoneme	
Punctuation:			*Pitch effect*	*Timing effect*
	.	(period)	Sentence final fall	Pause follows
	?	(question)	Sentence final rise	Pause follows
	!	(exclam)	Sentence final sharp fall	Pause follows
	…	(ellipsis)	Clause final level	Pause follows
	,	(comma)	Continuation rise	Short pause follows
	;	(semicolon)	Continuation rise	Short pause follows
	:	(colon)	Clause final level	Short pause follows

continued

Table 6-3   Prosodic control symbols (continued)

Type	Symbol		Description of effect	
Punctuation (continued):			*Pitch effect*	*Timing effect*
	(	(parenleft)	Start reduced range	Short pause precedes
	)	(parenright)	End reduced range	Short pause follows
	" '	(quotedblleft, quotesingleleft)	Varies	Varies
	" '	(quotedblright, quotesingleright)	Varies	Varies
	-	(hyphen)	Clause-final level	Short pause follows
	&	(ampersand)	Forces no addition of silence between phonemes	

Specific pitch contours associated with these punctuation marks may vary according to other considerations in the analysis of the text, such as whether a question is rhetorical or begins with a *wh* question word, so the above effects should be regarded only as guidelines and not absolute. This also applies to the timing effects, which will vary according to the current rate setting.

The prosodic control symbols (/, \, <, and >) may be concatenated to provide more exaggerated, cumulative effects. The specific nature of the effect is dependent on the speech synthesizer. Speech synthesizers also often extend or enhance the controls described in this section.

Summary of the Speech Manager

Constants

```
#define gestaltSpeechAttr   'ttsc'// Gestalt Manager selector for speech
                                   attributes

enum {
 gestaltSpeechMgrPresent   = 0   // Gestalt bit that indicates that Speech
                                   Manager exists
};
```

```
#define kTextToSpeechSynthType          'ttsc'    // text-to-speech
                                                  synthesizer component type
#define kTextToSpeechVoiceType          'ttvd'    // text-to-speech voice
                                                  resource type
#define kTextToSpeechVoiceFileType      'ttvf'    // text-to-speech voice file
                                                  type
#define kTextToSpeechVoiceBundleType    'ttvb'    // text-to-speech voice
                                                  bundle file type

enum {   // Speech Manager error codes (range from 240 - 259)
   noSynthFound          = -240,
   synthOpenFailed       = -241,
   synthNotReady         = -242,
   bufTooSmall           = -243,
   voiceNotFound         = -244,
   incompatibleVoice     = -245,
   badDictFormat         = -246,
   badPhonemeText        = -247
};

enum {          // constants for SpeakBuffer and text done callback
                controlFlags bits
   kNoEndingProsody      = 1,
   kNoSpeechInterrupt    = 2,
   kPreflightThenPause   = 4
};

enum {          // constants for StopSpeechAt and PauseSpeechAt
   kImmediate       = 0,
   kEndOfWord       = 1,
   kEndOfSentence   = 2
};

// GetSpeechInfo & SetSpeechInfo selectors
#define soStatus            'stat'
#define soErrors            'erro'
#define soInputMode         'inpt'
#define soCharacterMode     'char'
#define soNumberMode        'nmbr'
#define soRate              'rate'
#define soPitchBase         'pbas'
#define soPitchMod          'pmod'
#define soVolume            'volm'
#define soSynthType         'vers'
```

```
#define soRecentSync          'sync'
#define soPhonemeSymbols      'phsy'
#define soCurrentVoice        'cvox'
#define soCommandDelimiter    'dlim'
#define soReset               'rset'
#define soCurrentA5           'myA5'
#define soRefCon              'refc'
#define soTextDoneCallBack    'tdcb'
#define soSpeechDoneCallBack  'sdcb'
#define soSyncCallBack        'sycb'
#define soErrorCallBack       'ercb'
#define soPhonemeCallBack     'phcb'
#define soWordCallBack        'wdcb'
#define soSynthExtension      'xtnd'


// speaking mode constants
#define modeText      'TEXT' // input mode constants
#define modeTX        'TX'
#define modePhonemes  'PHON'
#define modePH        'PH'
#define modeNormal    'NORM' // character mode and number mode constants
#define modeLiteral   'LTRL'

enum {                              // GetVoiceInfo selectors
   soVoiceDescription   = 'info',   // gets basic voice info
   soVoiceFile          = 'fref'    // gets voice file ref info
};
enum {kNeuter = 0, kMale, kFemale}; // returned in gender field below
```

Data Types

```
typedef struct SpeechChannelRecord {
   long data[1];
} SpeechChannelRecord;

typedef SpeechChannelRecord *SpeechChannel;

typedef struct VoiceSpec {
   OSType   creator;   // creator ID of required synthesizer
   OSType   id;        // voice ID on the specified synth
} VoiceSpec;
```

```
typedef struct VoiceDescription {
   long        length;        // size of structure - set by application
   VoiceSpec   voice;         // voice creator and ID info
   long        version;       // version code for voice
   Str63       name;          // name of voice
   Str255      comment;       // additional text info about voice
   short       gender;        // neuter, male, or female
   short       age;           // approximate age in years
   short       script;        // script code of text voice can process
   short       language;      // language code of voice output
   short       region;        // region code of voice output
   long        reserved[4];   // reserved for future use
} VoiceDescription;

typedef struct VoiceFileInfo {
   FSSpec   fileSpec;   // volume, dir, & name information for voice file
   short    resID;      // resource ID of voice in the file
} VoiceFileInfo;

typedef struct SpeechStatusInfo {
   Boolean  outputBusy;      // true if audio is playing
   Boolean  outputPaused;    // true if channel is paused
   long     inputBytesLeft;  // bytes left to process
   short    phonemeCode;     // opcode for cur phoneme
} SpeechStatusInfo;

typedef struct SpeechErrorInfo {
   short    count;   // # of errs since last check
   OSErr    oldest;  // oldest unread error
   long     oldPos;  // char position of oldest err
   OSErr    newest;  // most recent error
   long     newPos;  // char position of newest err
} SpeechErrorInfo;

typedef struct SpeechVersionInfo {
   OSType      synthType;          // always 'ttsc'
   OSType      synthSubType;       // synth flavor
   OSType      synthManufacturer;  // synth creator ID
   long        synthFlags;         // synth feature flags
   NumVersion  synthVersion;       // synth version number
} SpeechVersionInfo;
```

```
typedef struct PhonemeInfo {
    short    opcode;         // opcode for the phoneme
    Str15    phStr;          // corresponding char string
    Str31    exampleStr;     // word that shows use of phoneme
    short    hiliteStart;    // segment of example word that
    short    hiliteEnd;      //   hilighted text (ala TextEdit)
} PhonemeInfo;

typedef struct PhonemeDescriptor {
    short         phonemeCount;     // # of elements
    PhonemeInfo   thePhonemes[1];   // element list
} PhonemeDescriptor;

typedef struct SpeechXtndData {
    OSType   synthCreator;  // synth creator ID
    Byte     synthData[2];  // data TBD by synth
} SpeechXtndData;

typedef struct DelimiterInfo {
    Byte     startDelimiter[2];   // defaults to [[
    Byte     endDelimiter[2];     // defaults to ]]
} DelimiterInfo;
```

Speech Manager Routines

Voice Routines

```
pascal OSErr MakeVoiceSpec (OSType creator, OSType id, VoiceSpec *voice);
pascal OSErr CountVoices (short *numVoices);
pascal OSErr GetIndVoice (short index, VoiceSpec *voice);
pascal OSErr GetVoiceDescription (VoiceSpec *voice, VoiceDescription *info,
    long infoLength);
pascal OSErr GetVoiceInfo (VoiceSpec *voice, OSType selector, void
    *voiceInfo);
```

Routines for Managing Speech Channels

```
pascal OSErr NewSpeechChannel (VoiceSpec *voice, SpeechChannel *chan);
pascal OSErr DisposeSpeechChannel (SpeechChannel chan);
```

Speaking Routines

```
pascal OSErr SpeakString (StringPtr s);
pascal OSErr SpeakText (SpeechChannel chan, Ptr textBuf, long textBytes);
pascal OSErr StopSpeech (SpeechChannel chan);
pascal OSErr StopSpeechAt (SpeechChannel chan, long whereToStop);
pascal OSErr PauseSpeechAt (SpeechChannel chan, long whereToPause);
pascal OSErr ContinueSpeech (SpeechChannel chan);
pascal OSErr SpeakBuffer (SpeechChannel chan, Ptr textBuf, long textBytes,
    long controlFlags);
```

Information and Control Routines

```
pascal NumVersion SpeechManagerVersion (void);
pascal short SpeechBusy (void);
pascal OSErr SetSpeechRate (SpeechChannel chan, Fixed rate);
pascal OSErr GetSpeechRate (SpeechChannel chan, Fixed *rate);
pascal OSErr SetSpeechPitch (SpeechChannel chan, Fixed pitch);
pascal OSErr GetSpeechPitch (SpeechChannel chan, Fixed *pitch);
pascal short SpeechBusySystemWide (void);
pascal OSErr GetSpeechInfo (SpeechChannel chan, OSType selector, void
    *speechInfo);
pascal OSErr SetSpeechInfo (SpeechChannel chan, OSType selector, void
    *speechInfo);
```

Text-to-Phoneme Conversion Routine

```
pascal OSErr TextToPhonemes (SpeechChannel chan, Ptr textBuf,
    long textBytes, Handle phonemeBuf, long *phonemeBytes);
```

Dictionary Management Routine

```
pascal OSErr UseDictionary (SpeechChannel chan, Handle dictionary);
```

Callback Prototypes

```
    // text-done callback routine typedef
typedef pascal void (*TextDoneProcPtr) (SpeechChannel, long, Ptr *, long *,
    long *);


    // speech-done callback routine typedef
typedef pascal void (*SpeechDoneProcPtr) (SpeechChannel, long );
```

```
      // sync callback routine typedef
typedef pascal void (*SyncProcPtr) (SpeechChannel, long, OSType);


      // error callback routine typedef
typedef pascal void (*ErrorProcPtr) (SpeechChannel, long, OSErr, long);


      // phoneme callback routine typedef
typedef pascal void (*PhonemeProcPtr) (SpeechChannel, long, short);


      // word callback routine typedef
typedef pascal void (*WordProcPtr) (SpeechChannel, long, long, short);
```

Error Return Codes

noErr	0	No error
paramErr	–50	Parameter error
memFullErr	–108	Not enough memory to speak
nilHandleErr	–109	Handle argument is nil
siUnknownInfoType	–231	Feature not implemented on synthesizer
noSynthFound	–240	Could not find the specified speech synthesizer
synthOpenFailed	–241	Could not open another speech synthesizer channel
synthNotReady	–242	Speech synthesizer is still busy speaking
bufTooSmall	–243	Output buffer is too small to hold result
voiceNotFound	–244	Voice resource not found
incompatibleVoice	–245	Specified voice cannot be used with synthesizer
badDictFormat	–246	Format problem with pronunciation dictionary
badPhonemeText	–247	Raw phoneme text contains invalid characters
unimplMsg	–248	Unimplemented message
badVoiceID	–250	Specified voice has not been preloaded
badParmCount	–252	Incorrect number of embedded command arguments
invalidComponentID	–3000	Invalid SpeechChannel parameter

CHAPTER 7

Introduction to
Speech Recognition

This chapter introduces the speech recognition software in the Macintosh Quadra 840AV and Macintosh Centris 660AV computers and tells you how to use it.

Speech is the most natural and common form of human communication. Having continuous speech recognition available to all users is a new and exciting kind of computer input. However, this is the first public release of speech recognition technology from Apple. The software is still under development, and its features may change. Not all of the guidelines and tools for developing speech-aware applications are included with this release. You may use this document as a guide for trying the software as it exists today. The performance and recognition accuracy of the software will improve as development progresses.

Casper is the code name for an interface for users and developers to Apple's underlying speech recognition and synthesis technology. All the necessary software to run speech recognition should already be loaded on the computer's hard disk or integrated on the software release floppy disks. See "Software Installation," later in this chapter, for the locations of the needed files. For best results with speech recognition, the Macintosh Quadra 840AV or Macintosh Centris 660AV microphone is required. Microphone placement and usage are described in "Using the Microphone," later in this chapter. Hardware support for speech recognition in the Macintosh Quadra 840AV and Macintosh Centris 660AV is described in "Sound I/O," in Chapter 2.

The following is a list of Casper's speech recognition features:

- *Speaker independence.* No training is required for new speakers.

- *Connected speech.* Phrases can be spoken in a natural manner rather than as isolated words separated by pauses.

- *Environmental adaptation.* The system automatically adapts to changes in background noise and room acoustics.

- *Vocabulary independence.* The system automatically synthesizes pronunciations for new words.

- *Standardized English.* The system recognizes speech from any adult speaking North American English.

How Does Casper Work?

The built-in speech recognition software can recognize any enabled menu item and dialog box button, with a few exceptions. The rules are simple: the active application must have used the standard Macintosh Toolbox, and the user must utter a standard English word for the name of the item.

How does Casper do it? The Casper speech recognizer requires a predefined grammar to function. There must be a finite number of possible phrases available; the more numerous the possibilities, the greater the chance of substitution errors. The Speech Setup control panel and a background application, the **Speech Monitor,** are the interface between the spoken word and speech control of the computer.

Macros or rules are defined to cover every available command. Some are built into uneditable files, such as the speech rules that control the menu items and buttons; some are available to user definition, such as macros created with the Speech Macro Editor (SME).

The recognition level is controlled by a slider that ranges from tolerant to strict in the Speech Setup control panel, described in "The Casper User Interface," later in this chapter. The recognition level should be adjusted to get a reasonable result within your environment and using your voice. The noisier the environment, the higher you will want to set this control to prevent Casper from trying to respond to background sounds.

In a noisy environment, voice commands generally work much better if you use a name, which acts as a keyword, before each command. If you have a slight accent, moving the slider toward Tolerant will make it easier for Casper to understand you. The system has been optimized for North American English accents, so non-English accents may reduce the accuracy rate.

The name you give your computer is user-editable; feel free to name it anything you like. Naming is supported in three modes. *Name optional* mode means that the name never has to be used. *Name required* mode means every utterance must be preceded by a name. *Name required 15 seconds after last command* is based on the principle that once a speaker has a listener's attention (in this case the computer), the speaker shouldn't have to address it by name all the time. So, once an utterance is recognized successfully, Casper switches into name optional mode until it hasn't heard anything for a specified number of seconds, set in the Speech Setup control panel. You can tell if the name is required at any time by looking below the animated icons in the feedback window, where the current name required is actively updated.

Software Installation

Speech recognition and synthesis can run on the Macintosh Quadra 840AV and Macintosh Centris 660AV base configurations of 8 MB of RAM. Speech recognition requires about 2.4 MB, however, and speech synthesis can use up to an additional 2.6 MB for a male voice without compression. Thus, when using speech features with applications that require more than than 1.5 MB of RAM, a total RAM capacity of 12 MB is recommended. To use speech with 8 MB of RAM, you can use compressed voices in the feedback options or disable the voice option. This will free a significant amount of memory.

For information about the base equipment configurations shipped with the Macintosh Quadra 840AV and Macintosh Centris 660AV computers, see "Models and Accessories," in Chapter 1. Macintosh system release 7.1 is also required.

The software for speech recognition either is preinstalled on your hard disk or comes on a series of floppy disks. If you install the software yourself, make sure you copy all the files to their proper locations. Once all the files have been copied, you must restart the computer.

You currently need the files listed below for speech recognition and synthesis.

These files belong to the Casper software package:
    Speech Scripting
    Speech Setup
    SR Macros
    SR Monitor
    SR Rules

These files belong to the PlainTalk software package:
    PlainTalk SR
    SR North American English

These files belong to the Gala Tea software package:
    PlainTalk TTS
    TTS Male Voice
    TTS Male Voice Compressed
    TTS Female Voice

These files belong to AppleScript:
    AppleScript
    Apple Event Manager

These files are scripting additions for AppleScript:
    Beep
    Choose Application
    Choose File
    Current Date
    Display Dialog
    File Commands
    Load Script
    Numerics
    Run Script
    Store Script
    String Commands

This file is a tool that helps you edit speech macros:
    Speech Macro Editor

Of the files just listed, the following go in the Extensions folder inside the System Folder:
    Apple Event Manager
    AppleScript
    PlainTalk SR
    PlainTalk TTS
    SR Macros
    SR Monitor
    SR North American English
    SR Rules
    TTS Female Voice
    TTS Male Voice
    TTS Male Voice Compressed

The following files go in a folder named Scripting Additions inside the Extensions folder:
   Beep
   Choose Application
   Choose File
   Current Date
   Display Dialog
   File Commands
   Load Script
   Numerics
   Run Script
   Store Script
   String Commands
   Speech Scripting

The Speech Setup file goes in the Control Panels folder.

The Speakable Items folder goes inside the Apple Menu Items folder.

Using the Microphone

The Macintosh Quadra 840AV and Macintosh Centris 660AV computers are compatible with a new higher-fidelity microphone that is designed specifically for speech recognition. The microphone has a special connector designed to plug into the audio input on the computer. Do not attempt to plug another microphone into the Macintosh Quadra 840AV or the Macintosh Centris 660AV; the software and hardware are designed specifically to work with Apple's microphone. Do not attempt to plug the speech recognition microphone into another device; it uses a custom connector that is not designed to work with other devices. For further information, see "Sound I/O," in Chapter 2.
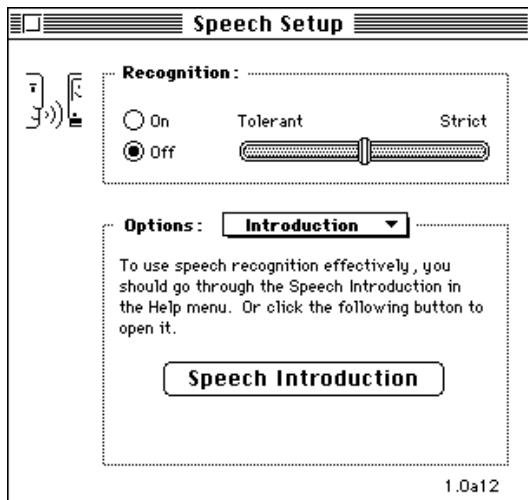
When using the microphone for speech recognition, place it in the top center of the monitor. While speaking, you should maintain a constant distance of approximately 12 to 24 inches from the microphone. The microphone is directionally sensitive. Side-to-side or up-and-down head movement will alter the voice reception and may result in reduced recognition accuracy.

Getting Started

You are now ready to try talking to the computer. The following steps will enable the speech recognition software and allow you to begin commanding your computer by voice.

Go to the Control Panels folder and open Speech Setup. The Speech Setup control panel will open and should look something like Figure 7-1. This control panel turns speech recognition on and off and is the interface between the user and the speech recognition system. The Options pop-up menu selects between four different modes of control panel operation.

Figure 7-1   Speech Setup control panel

When the Options pop-up menu is set to Introduction, as shown in Figure 7-1, the user is advised to read the material in online help. The other Options pop-up menu selections are the following:
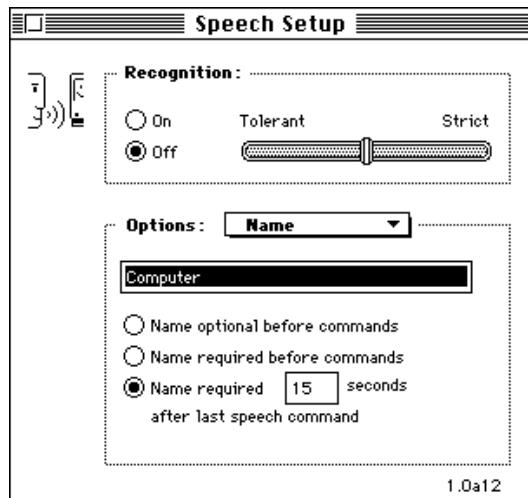
■ Name, which lets the user select a name for addressing the Macintosh computer

■ Feedback, which lets the user select a feedback voice and sounds for replies from the computer

■ Attention Key, which lets the user select a key combination to temporarily disable and enable speech recognition

These three pop-up menu selections are described in more detail in the next three sections.
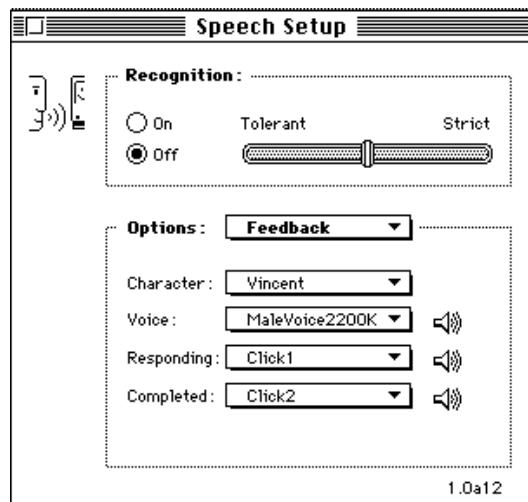
Setting Your Computer's Name

Before you can talk to your computer, you must give it a name. The name should be short but distinctive—for example, "Computer." The naming function of the Speech Setup control panel is shown in Figure 7-2.

Figure 7-2 Setting your computer's name

Choosing Speech Feedback

Feedback sounds are used to indicate positive recognition of a command and completion of its execution. A variety of aural feedback is available for use, several text-to-speech voices are available, and any system sounds can be used. The feedback function of the Speech Setup control panel is shown in Figure 7-3.
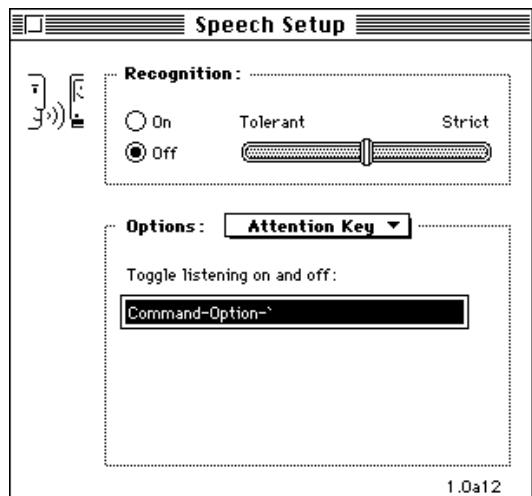
Figure 7-3 Choosing feedback signals

Setting the Attention Key

The attention key helps minimize two inherent problems of speech recognition: the speech recognizer's tendency to tie up the computer's processor trying to respond to noise in the environment, and its occasional interpretation of random sound as a valid speech command. Thus, the attention key is useful in noisy situations in which the speech recognizer must be temporarily disabled and reenabled. Any single key or combination of keys can be used for this purpose except for Command-key combinations and other combinations reserved for system use. Two suggestions are F15 (marked as "pause" on extended keyboards) and the "record" button, marked with a microphone symbol on the new Norsi keyboard. The attention key function of the Speech Setup control panel is shown in Figure 7-3.

Figure 7-4   Setting the attention key

The Casper User Interface

The Speech Setup control panel allows the user to adjust certain operating parameters and the audiovisual feedback that indicates the state of the recognizer. Various Speech Setup control panel modes are shown in the previous section, "Getting Started."

Operational Control

The operational controls in the Speech Setup control panel allow you to make the computer start and stop listening and to change the speech recognition listening parameters. Name Optional should be used only in a quiet environment. Use of a name before a command prevents Casper from attempting to recognize all spoken words and possibly trying to recognize words spoken by someone other than the user.

The following contains a description of each control. Note that the controls may be changed whether or not speech is currently enabled. If it is enabled, the changes will take effect immediately.

On/Off　　　　　This control allows the user to turn speech recognition on and off. When speech recognition is shut off, all memory and CPU and DSP bandwidth resources are reclaimed.

　　　　　　　　　The Speech Setup control panel sets the startup mode for speech recognition. It must be explicitly turned off before shutdown, or speech recognition will be on when the system is powered up.

Tolerant/Strict　　This slider allows the user to adjust the recognition level required for Casper to recognize an utterance. When the slider is set to the far left, Tolerant, more out-of-grammar and out-of-vocabulary utterances are allowed. When the slider is set to Strict, Casper is more likely to reject a valid utterance. The recommended setting is approximately the center of the scale, as shown in Figure 7-1.

Name　　　　　This editable text field allows the user to specify the name that is used before spoken commands. You can verify that the system is working by using the default name Computer before experimenting with other names.

Name optional before commands

　　　　　　　　　This radio button, when selected, causes the system to recognize spoken commands that are not preceded by the name. (However, commands may still be preceded by the name.) See Figure 7-2.

Name required before commands

　　　　　　　　　This radio button, when selected, causes the system to recognize only commands that are preceded by the name. See Figure 7-2.

Name required _ seconds after last speech command

　　　　　　　　　This radio button, when selected, allows the user to enable a timeout value, in seconds, in conjunction with Name required mode. Within the specified number of seconds after the most recent command, it will not be necessary to say the computer's name again before saying a new command.

Feedback Control

This group of controls in the Speech Setup control panel lets the user change how the system provides feedback.

Character　　　　This pop-up menu allows the user to select which character represents the talking computer.

Voice　　　　　This pop-up menu allows the user to select the voice to be used for spoken feedback. The choices available will depend on which synthesizers are installed in the System Folder. If no voices are available, this control is disabled.

Responding　　　This pop-up menu allows the user to select the sound to be used to acknowledge that a command has been understood. Audio feedback is also used for commands that have no text specified for their acknowledgment. The sounds available in this menu are the sounds stored in the System file. Sounds are listed alphabetically.

Completed   This pop-up menu allows the user to select the sound to be used when a spoken command has been carried out. The sounds available are the same as those for the responding sound.

Speech Macro Editor

Speech macros let users initiate actions on the computer by speaking phrases. A speech macro comprises the following pieces:

■ The macro *name* is what users actually speak to trigger the macro.

■ The macro's *context* specifies where users can speak the macro name and have the computer recognize it. For now, the context is either anywhere (the computer will always recognize the name) or a specific application (the computer will recognize the name only when the application is active in the frontmost window).

■ The macro's *scripting system* determines what script engine will execute the commands and hence what commands are used for writing or recording scripts. The script engine must be compatible with the Open Scripting Architecture (OSA) standard. For the Macintosh Quadra 840AV and Macintosh Centris 660AV, such a system is either AppleScript or QuicKeys, although others might appear later.

■ The macro's *script* is a sequence of commands triggered by the macro name. The commands are interpreted by the scripting software currently selected by the Script pop-up menu.

■ Macros can specify whether to signal the user with the *responding* and *completed* sounds by setting the feedback controls in the Speech Setup control panel.

The application **Speech Macro Editor (SME)** lets users create and edit speech macros. An SME document contains a list of the speech macros in the document. Users can edit the attributes of a macro, write or record new macros, and manage the list of macros in the document.

Scripting Tool Requirements

The Speech Macro Editor requires the general AppleScript OSA interpreter for playing and recording scripts. Alternatively, users can compose scripts with QuicKeys. Speech recognition is not needed to write and edit voice macros, just to speak them.

Users will have to deal with the difference between writing scripts and recording them. Whether either is possible depends on both the scripting system and the specific application. The current situation for the Macintosh Quadra 840AV and Macintosh Centris 660AV is described in this section.

AppleScript

AppleScript is good for capturing high-level descriptions of action on objects in an application as opposed to low-level events (for example, moving the icon Untitled to the Trash as opposed to drag from 100,100 to 512,342). The range and quality of these

high-level descriptions depend entirely upon the amount of work a developer puts into factoring the application.

To write scripts, users must have AppleScript-aware applications and must know the scripting commands for those applications.

To record scripts, users must have applications that can convert actions into commands as users perform them. Moreover, they'll probably have to look at the recorded script to determine whether the application recorded all their actions.

As shipped, the Macintosh Quadra 840AV and Macintosh Centris 660AV are not supplied with scriptable or recordable applications. Some third-party applications are currently available. The Apple Scriptable Text Editor is recordable, and Excel and FileMaker® Pro are scriptable. The AppleScript system will become more useful as more and more applications support it.

QuicKeys

QuicKeys is good for recording low-level events and thus for handling simple interactions with most applications. It suffers many of the same problems as the original MacroMaker from Apple:

■ It usually just replays users actions exactly (users see the interface flying by as if they were doing it).

■ Since it's just replaying low-level events, many of its commands break down if the position of the underlying object changes.

■ It lacks the full expressive qualities of AppleScript; it's really its own language, but one lacking sophisticated conditionals, loops, procedure calls, and so on.

To write scripts, users must know the QuicKeys language supported by the OSA component so that they can change volatile commands such as Drag and Click At to more stable commands where possible. The Macintosh Quadra 840AV and Macintosh Centris 660AV system software supports QuicKeys, so users can create new macros. CE Software also provides a set of example macros written with QuicKeys. The QuicKeys scripting language may not include the full power of the "normal" QuicKeys system; for example, QuicKeys extensions, which circumvent the interface and set values directly, may not be supported.

User Requirements

As with AppleScript, users of the Speech Macro Editor will have to be fairly sophisticated to be able to write and edit scripts; the majority of users will have to use prewritten scripts. Recording should allow less experienced users to create voice macros, but recording must be viewed as a shortcut for typing a new script; further editing will probably be required.

Since the SME isn't trying to reproduce the full suite of scripting tools being developed for AppleScript (no debugging, no access to help on scripting commands, and so on), its users will need to know how to find the answers to these questions:

■ Is the application AppleScript aware? Is it also recordable? What scripting commands does it provide?

■ What scripting commands does the script system provide?

The script editor provided with AppleScript has facilities for developing more complicated scripts than are possible with SME and includes complete debugging and error reporting features.

Using the Speech Macro Editor
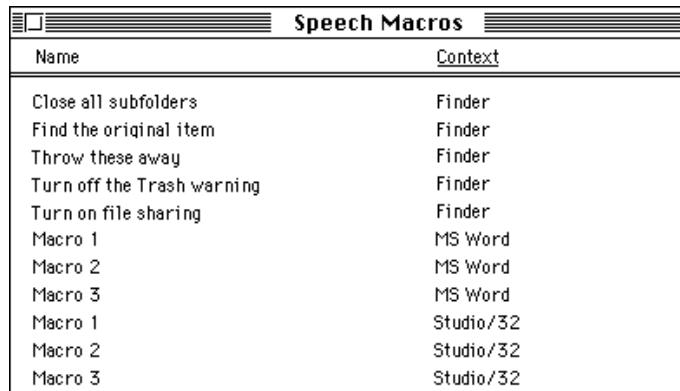
The Speech Macro Editor is an application in the Extras folder on the hard disk. Here's how to start it:

1. **Open the Extras folder.**

2. **Open the Speech Macro Editor by double-clicking its icon.**
   When the Speech Macro Editor starts up, by default it automatically opens the Speech Macros document from the Extensions folder. The document window for Speech Macros lists all the speech macros it contains. Initially, the SME does not select an item in the list. A typical Speech Macros document window is shown in Figure 7-5.

Figure 7-5   Typical Speech Macro document window

IMPORTANT

Speech macros can exist in any SME document. The Speech Macros document is just the default document shipped with the computer. An SME document must be in the Extensions folder or the System Folder for it to become part of the current grammar. ▲

Recording a New Macro

To record a new macro, follow the steps below. As an example, we'll create a speech macro for copying the first item in the Scrapbook.

1. **Choose Create New Macro from the Macro menu.**
   A blank macro window appears onscreen. The insertion point is set in the Name field.

2. **Type the phrase that you want to speak.**
   It's best for the name to be a phrase rather than a single word. Recognition works faster and more accurately if the differences among names are more pronounced. Also note that, unlike most speech recognition technologies, Casper can recognize continuous speech.
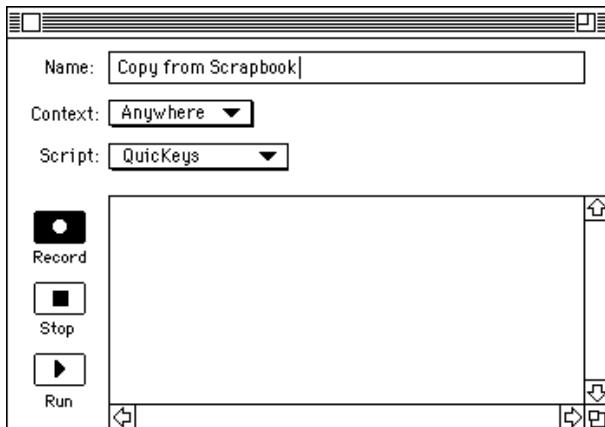
3. **From the Context pop-up menu, choose the context in which you'll be able to speak the new macro.**
   In this case, you want to have this macro available at all times, so change the context to Anywhere.

   Figure 7-6 shows these three steps performed in a New Macro window.

Figure 7-6   Typical New Macro window

Note

If users open the SME on a system that's not running AppleScript, they can only edit scripts. The following actions with the Record, Stop, and Play buttons and the Script pop-up menu will not be available; these items will be dimmed. ◆

4. **Choose a script language for recording the macro.**
   The choice of script system determines what applications (and events in those applications) are recordable. Users need to understand the benefits and limitations of a particular choice here. Since the system and Finder won't support AppleScript, change the script system to QuicKeys.

5. **Click the Record button.**
   The button "locks" in place, and a small recording icon blinks over the Apple menu while the user is in record mode. The icon that appears depends on the script system (AppleScript displays a small cassette, QuicKeys a small microphone).

6. **Switch to the application and perform the desired actions.**
   In this example, pull down the Apple menu, choose Scrapbook, choose Copy, and close the window.

IMPORTANT

Script systems may handle the posting of commands differently. For example, AppleScript sends commands to the SME after each one occurs. QuicKeys returns an entire script after the user stops recording. ▲

7. **Return to the SME by clicking an open SME window or by choosing SME from the Application menu, then click Stop.**
   Wait until the recording icon stops blinking. The script appears in the script area of the window.

8. **Click the close box to save the new macro.**

Renaming a Macro

To change the name of a macro, follow these steps:

1. **Select the macro you want to edit and choose Edit Macro from the Macro menu, or double-click the macro name.**

2. **A macro window appears.**

3. **Type the new name in the text field.**

4. **Click the close box.**

The window disappears, the name and context items in the list of macros change, and the list is sorted.

Saving Macro Changes

At any point, the user can save changes to an SME document by choosing one of the Save commands in the File menu. These commands are available from the main document window or any of the macro windows. If a save command is chosen when a macro window is active, the SME saves the entire document in which that macro resides.

The SME displays the standard Save Changes dialog box if the user closes a document window without having first saved changes.

Loading Macros

Casper loads macros at the following times:

■ When users turn speech on from the Speech Setup control panel, Casper loads rules from any SME document that is in the Extensions folder or the System Folder on the startup disk. Casper also loads any speech rules documents found in either of these two locations.

■ When users make changes to any of the SME documents loaded from the Extensions folder or the System Folder, Casper reloads the changed SME documents when the user saves the document. Casper should acknowledge that it's reloading the macros (so that users know it's happening) by posting a message to the feedback window.

■ Users who place new SME documents or new speech rules documents in the Extensions folder or the System Folder must stop and restart Casper to load the new documents. Casper keeps track only of the documents it loaded when starting up.

■ If an application contains speech rules in its resource fork, they will be loaded when the application is launched. For further information, see "Speech Rules Files," in Chapter 8.

Built-in Speech Rules and Grammar

Speech rules are structures used to define how words can be strung together for speech recognition. They are discussed in detail in Chapter 8, "Speech Rules." Since many commands (such as those that choose menu items) are required in all applications, a standard set of rules is built into Casper to provide a robust set of standard commands. Many menu functions are common across a wide variety of applications, and most applications will also use Finder-type commands to access the Apple menu items.

In English there are grammar rules that define the noun-verb-subject sequence. A similar sequence must be identified explicitly for the speech recognizer. For example:

"Open Chooser"

"Open the Chooser"

"Open menu item Chooser"

could all be used to open the Chooser control panel. All of the acceptable word strings must be defined in order for the Speech Monitor to select the correct command. If the user says "Chooser open," the rules in this example will not recognize that statement as an acceptable command. If the word string "Chooser open" is added to the rules, then Casper will respond with an acceptable command.

In the Macintosh Quadra 840AV and Macintosh Centris 660AV speech recognition software, all menu items and dialog box buttons are controllable by speech. Use the following command forms:

■ "Open *AppleMenuItem*," where *AppleMenuItem* is any item within the Apple menu— for example, "Open Alarm Clock"

■ "Switch to *ProcessMenuItem*," where *ProcessMenuItem* is the name of any process—for example, "Switch to Finder"

A new Speakable Items folder exists in the Apple Menu Items folder in the Macintosh Quadra 840AV and Macintosh Centris 660AV system software. Any item or alias to an item within it will be speakable. Some aliases to standard items are installed automatically, such as Open System Folder. The phrase to speak these items is the same as the name given to the item. AppleScript (or QuicKeys) items can be placed in the

Speakable Items folder as well. This folder is not dynamically updated at present, so speech must be shut down and restarted to load any new items placed in it.

Here are some sample Finder phrases:

- "Hello"
- "What time is it"
- "What day is it"
- "close window"
- "close all windows" (available only when the Finder is frontmost)
- "zoom window"
- "is file sharing on"
- "start file sharing"
- "stop file sharing"
- "shut speech off"

Here are sample printing phrases:

- "Print from n to n"
- "Print from page n to m"
- "Print n copies"
- "Print page n"
- "Print page n to m," where n and m are numbers from 1 through 99. (This works in all applications that use Cmd-P to print.)

Performance

Casper's speech recognition goal is a minimum in-grammar error rate for a typical task in a low-noise environment. *In-grammar error rate* is the number of times the speech recognition software does not respond as intended when a defined command is spoken. All of the variables listed in the next section affect the ability of the system to recognize speech.

Real-Time Response

Response time is a function of several variables:

- *Clear pronunciation.* The search tends to be faster if the utterance is spoken clearly in North American English.
- *Grammar complexity.* The higher the number of possible word phrases in the speech rules, the longer the search and the higher the error rate.

- *Word complexity.* The choice of words can affect the duration of the search; similar-sounding words are harder to distinguish.

- *Extraneous noise.* Additional noise affects the quality of the input and potentially increases the search time as the noise is increased.

- *Room acoustics.* Bad acoustics may degrade system performance, including response time from one acoustic environment to the next.

- *Environmental adaptation.* This algorithm adapts to changing room conditions and background noise—after every five utterances, the environmental adaptation is updated.

Types of Errors

Taken as a group, the rules for a specific application form the grammar for that application. The recognition search returns the best match from the available grammar.

One type of error occurs when the search results are too uncertain, in which case the speech recognizer rejects the sentence as unrecognizable. Another type of error occurs when an in-grammar match is selected to an incorrect sentence and the speech recognizer responds although no command was given.

Apple's naming conventions for speech recognition responses, both correct and erroneous, are shown in Table 7-1.

Table 7-1   Grammatical naming conventions

In-grammar	Out-of-grammar
Correct recognition	Correct rejection
Incorrect recognition	Correct detection of new word
Incorrect rejection (correct words/grammar not identified)	Incorrect recognition (through substitution or insertion)

For the in-grammar case the first item is the nonerror response. For the out-of-grammar case, the first two items are the nonerror responses. There are several reasons why a phrase might not be properly recognized—for example, unclear pronunciation, background noise, or bad room acoustics.

Acceptable Limits or Constraints

The system is constrained to North American adult English used in grammatically simple sentences. Note that this also implies a limited vocabulary.

The system accuracy will typically drop during changing environmental conditions. Adaptation takes approximately five utterances.

The speech recognition software currently understands only clearly spoken English words. The user must speak in well-defined sounds for all words and sentences.

Note

The speech recognizer cannot recognize most nonstandard English words. A nonstandard word could be any word that is formed as a result of concatenating words, using abbreviations, or other shortcuts, which typically result in many ways to say the same word. The current recognizer accepts only one pronunciation for a word, with only small variations from that pronunciation. As an example, a made-up word used as a filename may not be recognizable. Abbreviated forms of words (such as *MPW*) are not typically recognizable as words. ◆

Speech Rules

This chapter describes how the speech recognition software in the Macintosh Quadra 840AV and Macintosh Centris 660AV uses speech rules to interpret and respond to the user's utterances. It also describes the `CompileRules` tool available with the Macintosh Programmer's Workshop (MPW), which compiles the rule source files into resources. Read Chapter 7, "Introduction to Speech Recognition," before reading this chapter.

Overview

At the heart of Apple's speech recognition system is a data structure called a **speech rule.** A speech rule is a word or a sentence that is defined to perform an action within the current computer environment. Each rule performs a unique function depending on the words spoken. An application's grammar is derived from the set of speech rules and the current context.

A rule can include variables used in locations that can be more than a single word. A word within a sentence that can be substituted with another word is called a *category.* A category can be an individual word or another category. When it is a predefined category, the acceptable words are listed in that category. For example, `<number>` can be a number from 1 to 9. A `<ten>` is defined as a number in the tens location, plus a `<number>` or a 0. A `<hundred>` is defined as a number in the hundreds location, plus a `<ten>` or a 0, plus a `<number>` or a 0. This process can be continued to make up any arbitrarily large number. In each case the category is made up of previously defined categories, except for `<number>`, which is a list of individual words.

In its simplest form, a speech rule maps some spoken utterance to a value or an action. When the speech recognition software detects that the user has uttered the phrase, the corresponding value is computed or an action is performed. Here is an example of a simple speech rule:

```
%rule
  bold
%action
  tell application "MyApp"
    set style of selection to bold
  end
%end
```

The effect of this rule is that whenever the user says "bold," the application named MyApp changes the selected text to bold. The `%rule` clause signals the beginning of a new speech rule; the line containing `bold` contains the phrase that should be recognized; the `%action` clause signals the beginning of the action part of the rule; the lines from `tell` to `end` contain the script that should be executed when the rule's phrase is recognized; and the `%end` clause signals the end of the rule.

A speech rule can have any number of phrases—for example:

```
%rule
  bold
  change to bold
  bold this
  make it bold
%action
  tell application "MyApp"
    set style of selection to bold
  end
%end
```

This is a valid speech rule, the effect of which would be to cause MyApp to change the style of the selection when any of the specified phrases is recognized.

Note

Avoid using the same speech string twice. If two speech commands are identical, Casper will use only the first macro it finds. The second macro will be ignored. ◆

One problem with the foregoing rule is that it causes the MyApp application to change styles, no matter what application is currently active. So, if MacWrite® is the active application and the user says"bold," MyApp will change styles (or worse, if MyApp is not running, it will be launched and then it will change styles). One way around this problem is to specify that the rule should be active only when MyApp is the active application:

```
%rule
  bold
%context application "MyApp"
%action
  tell application "MyApp"
    set style of selection to bold
  end
%end
```

In this case, the speech recognizer listens for the phrase *bold* only when the MyApp application is active. Spoken commands that make sense only when a particular application or window is active can be marked in this fashion.

As you begin to build up larger vocabularies for your computer, you will want to avoid having to enumerate every utterance that the system should recognize. Speech rules can be used to construct entire grammars of what the user can say. For example, let's say you

want to define a rule that allows the user to change the selection to any style, without having to list every utterance separately:

```
%define style
  bold
  italic
  underline
%end
```

The foregoing is called a *category rule.* It is similar to the command rule in that it defines a set of phrases that the user might say. However, it does not specify an action. Instead, this rule defines a token, `<style>`, which can be used in other rules instead of directly enumerating the category's phrases.

```
%rule
  <style>
  change to <style>
%action
  tell application "MyApp"
    set style of selection to ...
  end
%end
```

Defining the rules this way lets you specify the syntax of the style command itself separately from the syntax for the style names. Other commands can also refer to the `<style>` category.

Note that in the action for the preceding rule, an ellipsis (. . .) was used in place of the actual style. The initial example used a constant style, but in this case, the actual style depends on which style the user says. There is a way to pass that information from the category rule to the command rule, by attaching a script fragment to each phrase. The script fragment returns a value representing the meaning of that phrase—for example:

```
%define style
  plain        ; {meaning: plain}
  bold         ; {meaning: bold}
  italic       ; {meaning: italic}
  .
  .
  .
%end
```

For each phrase, the text to the left of the semicolon defines what the user can say, and the text to the right of the semicolon is the AppleScript expression. This technique allows

the rule writer to assign a meaning to each of the possible phrases that the user may utter. The rule that uses this meaning, then, looks like this:

```
%rule
  <s:style>
  change to <s:style>
%action
  tell application "MyApp"
    set style of selection to meaning of s
  end
%end
```

Every reference to a category whose value is needed should be preceded with a variable name. When the subsequent script is executed, the variable will be bound to the value returned by the category rule. For example, if the user says "change to bold," the style category matches the word *bold,* producing as its value the Apple event record {meaning: bold}. The above command rule then matches the entire utterance, executing its script with the variable s bound to the value produced by the corresponding category rule. Finally, the expression meaning of s retrieves the style constant from the meaning record.

Note the use of the meaning property to access the value computed by the category. Whenever a phrase's script is evaluated, the value returned is always coerced into an Apple event record. In the example just given, a record was used as the value of each of the category's phrases. Since it was already a record, it was used as is. If the value is any other data type, it is stored as the meaning property of an Apple event record, and the record is used as the returned value. For example, the following two phrases are equivalent:

```
one          ; 1
one          ; {meaning: 1}
```

Thus, when accessing the value bound to a variable in a category reference, it is usually necessary to get its meaning property.

Here is another example of using categories to define a grammar for numeric digits:

```
%define digit
  one        ; 1
  two        ; 2
  three      ; 3
  .
  .
  .
  nine       ; 9
%end
```

This category defines a simple grammar that will recognize a single spoken digit and return the numeric value of that digit. A script can access the value returned by a category by preceding the category reference with a variable name:

```
%rule
  what is <n:digit> plus <m:digit>
%action
  .
  .
  .
  set x to (meaning of n) + (meaning of m)
  .
  .
  .
%end
```

Using the techniques described so far, you can define a category for recognizing whole numbers less than 100. First, define a rule to recognize the tens words:

```
%define tens
  twenty    ; 20
  thirty    ; 30
  .
  .
  .
 ninety    ; 90
%end
```

This rule is exactly like the definition of digit just given. Next, define a rule to recognize the teens words:

```
%define teens
  ten       ; 10
  eleven    ; 11
  .
  .
  .
 nineteen  ; 19
%end
```

Finally, define the rule that combines all the parts and returns the correct value for multiword phrases:

```
%define uhundred
  <n:digit>     ; n
  <n:teens>     ; n
  <n:tens>      ; n
  <n:tens> <d:digit>; (meaning of n) + (meaning of d)
%end
```

For the first three phrases, the value returned is simply the value recognized by the subordinate category. For example, a single digit is a valid number to be recognized by this rule, and its value is simply the value returned by the digit rule. In the case of the fourth phrase, you want to recognize spoken numbers such as *twenty-five.* The script for this phrase essentially computes the meaning of speaking these two words in sequence. This is a trivial example, but the general mechanism is a powerful one that can be used to associate meaning with a wide variety of spoken commands.

Note that in the fourth phrase above you did not write n + d as the script. This is because the values bound to n and d are Apple event records, not numbers. In the previous cases, you were simply passing on the values, so you could leave them as records; but when you want to do arithmetic, you need to access the meaning explicitly. An equivalent, if more verbose, expression is the following:

```
<n:tens> <d:digit>; {meaning: (meaning of n) + (meaning of d)}
```

Sometimes the meaning of an utterance resides simply in the words spoken. For example, consider a phone-dialing application in which you want to acknowledge that the spoken command is being carried out:

```
%define name
  John Doe      ; {phone: "555-7442"}
  Bob Strong    ; {phone: "555-3295"}
  .
  .
  .
%end

%rule
  call <n:name>
%action
  dial (phone of n)
  acknowledge saying "Now dialing"
%end
```

In this example, each person's phone number is attached as part of the `meaning` structure. Notice that a different property is used. This works fine; you can use any properties that you want as long as you return a record. With speech recognition, however, there is always the possibility of a mistaken recognition. It would be better to tell the user the name of the person that the system is dialing, so that if it fails to recognize correctly, the user has a chance to hang up before the call goes through. You could attach the person's name to the meaning structure:

```
John Doe         ; {phone: "555-7442", name: "John Doe"}
```

However, this would be redundant. As a convenience, the Speech Monitor always adds an utterance property to the value generated by a phrase script. The value assigned to this property is a string containing the words that were matched by the category rule. So, you can rewrite the action script of your phone-dialing rule as follows:

```
%rule
  call <n:name>
%action
  dial (phone of n)
  acknowledge saying "Now dialing " & (utterance of n)
%end
```

Speech Rules Files

Speech rules are data structures that determine how spoken commands are interpreted. They are stored as resources either in **speech rules files** or in the resource fork of an application. When speech is started, the System Folder and the Extensions folder are scanned for speech rules files. Any speech rules files found in these two locations are scanned, and the rules in those files become active and are used for spoken command recognition. Rules resources present in an application are loaded when an application is launched.

There are actually two different file types used for speech rules files: one for speech rules files proper and one for macro files. A **speech macro** is a simplified kind of speech rule that can be created with the application Speech Macro Editor. Internally, these files have identical formats, and the speech recognition system does not distinguish between the two.

The `CompileRules` MPW tool is used to generate rules files or rule resources from text files. The syntax to invoke `CompileRules` is

```
CompileRules [ options ] input-file …
```

Any number of input files may be specified. The valid options are as follows:

-b   The -b option causes all scripts in the file to be precompiled and
                      stored in their binary format. If this option is not specified, the rules
                      will be compiled by the Speech Monitor at run time, on demand.

-base *integer*   The -base option causes rule resources to be numbered beginning
                      at the specified ID. If this option is not specified, resource IDs begin
                      at 0. This option is useful in order to prevent resource ID collision
                      when the rule resources are going to be installed in an application
                      file. The rule compiler currently generates resources of types
                      'rule', 'glob', and 'scpt'.

-c *creator*   The -c option may be used to specify a creator for the output file. If
                      not specified, the creator is set to '????'.

-category *category-name*

                      This option is used in conjunction with the -generate option to
                      cause phrases generated by a particular category to be generated. If
                      this option is not used, then phrases are generated from the set of all
                      possible commands.

-generate all

-generate *integer*

                      The -generate option is used to print to standard output a list of
                      utterances generated by the grammar defined by the input files. If
                      all is specified, then all possible utterances are listed. Otherwise,
                      the number of utterances specified by *integer* is printed out,
                      generated at random according to the method defined by the
                      -method option.

-method total | phrase | mixed

                      This option is used to specify the method of generating random
                      utterances. The total method generates each utterance from the
                      total set of possible utterances with equal probability. The phrase
                      method generates utterances such that each phrase from a rule is
                      equally likely to be chosen. The mixed method uses the phrase
                      method to choose a top-level command at random and then uses
                      the total method to expand any categories contained in that
                      utterance. The default method is mixed.

-o *output-file*   The output file is designated with the -o option. If this option is not
                      specified, the input files are read and checked for correct syntax, but
                      no output is created.

-p   The -p option causes informative progress messages to be written
                      to standard output as the rules are compiled.

-unique   The -unique option is used in conjunction with the -generate
                      option in order to force all generated utterances to be unique.

The CompileRules tool must be run on a system that contains the scripting systems to
be compiled (that is, AppleScript). Errors in the speech rules file will result in messages
being written to standard output with the error and line of the file where each error
occurred. The format for the text file is given in the next section.

There are two kinds of speech rules: command rules and category rules. Command rules are like speech macros; in fact, speech macros are instances of command rules. They cause a specified action to occur when the Speech Monitor hears a particular phrase. Every command rule has the following parts:

■ A list of *phrases,* each of which defines a phrase that the user may utter to cause the action below to be carried out. Each phrase has an optional script that can define a semantic value to be associated with the phrase and can be accessed in the action's script. The phrase itself consists of a list of tokens that are references to either words or categories. Words are like terminals in a grammar, and categories are like nonterminals.

■ An optional *context* that defines when the command rule is active. If the context is empty, the rule is always active.

■ An optional *condition,* which is a script that determines whether or not the rule should be considered active. This is like the context, except that it is evaluated rather than constant, and it is evaluated after the utterance has been recognized. It is useful for resolving ambiguities when more than one rule has matched the user's utterance.

■ An optional Boolean *acknowledge flag* that causes the command to be acknowledged in the standard (nonverbal) way. If it is desired to provide verbal acknowledgment, then the flag should be `false`, and the acknowledge AppleScript command should be used. Normally this flag is used for very short commands, such as menu items, dialog box buttons, and so on.

■ An optional *target clause*, which indicates a default target for the condition and action scripts. If no target is specified, the default target for the scripts is the Speech Monitor itself. The target can be changed in the script by using the `tell` clause of AppleScript.

■ An *action,* which is a script that is executed when one of the rule's phrases has been uttered by the user and recognized. The action's script may refer to variable bindings created by any of the phrase's scripts that have matched the user's utterance. The default target is `Speech Monitor`, so that any scripts sent to the Speech Monitor can be used without the standard `tell application` block.

■ An optional *index clause,* which consists of a list of index terms used by the help system. It is recommended that this clause always be provided.

■ An optional *description clause,* which is a textual description of the effect of the command. This field is used by the help system.

Category rules are used to create subgrammars, which may be used by command rules and other category rules. Each category rule defines a set of phrases that may be recognized as part of an utterance. Category rules do not have actions and are relevant only when they are referred to by command rules (directly or indirectly). Every category rule has a name, which is used to refer to the category in other rules.

In addition, there are two subtypes of category rules: internal and external. *Internal* categories are rules that have their phrases listed explicitly in the rule. The list of phrases has exactly the same form and function as in command rules. The variable bindings assigned by the phrases' scripts can be used by any rule that refers to the category. *External* categories are rules that have their phrases computed by a script. The value returned by the script should be either a list of strings or a single string of phrases

separated by newline characters. External categories have an additional option called *dynamic,* which determines when the script is evaluated. Nondynamic external categories have their scripts evaluated once, when speech starts up (or if there is a context, each time the context makes a transition from not active to active). Dynamic external categories have their scripts evaluated every time an utterance is detected and their context is active (if the rule has a context).

When a speech rules file is saved in the Macintosh System Folder or Extensions folder and speech is on, a `reload rules` command needs to be sent to the Speech Monitor or speech must be shut off and restarted to load the changes. The Speech Macro Editor does this, for example, when you save a macro file that is stored in one of these two places. You can also use AppleScript to make this happen. The following script causes the named speech rules file to be loaded or reloaded:

```
tell application "Speech Monitor"
  reload rules (alias "Hard Disk:Rules:My Rule File")
end
```

Speech Rules File Syntax

Speech rules are stored as resources either in speech rules files stored in the System Folder or in the resource fork of an application. The resources in these files are created from a text form using an MPW tool called `CompileRules`. The syntax for the text form of speech rules files is described in the next section. These notations are used:

■ Anything appearing between brackets is optional. The brackets themselves do not appear in the source file.

■ Ellipsis points (three periods) are used to denote zero or more repetitions of the preceding element.

■ A name in italics is a reference to another syntactic element, defined elsewhere. Any other item not in italics should appear in the source file exactly as indicated.

Command Rules

The general form of a rule is this:

```
%rule
phrase
  .
  .
  .
[%context [ context ... ] ]
[%acknowledge]
[%target [ signature ] ]
```

```
[%condition [ script-type ]
script]
%action [ script-type ]
script
[%index term [, term ... ] ]
[%description
description ]
%end
```

The `%target` statement is used to set the default target for all scripts that occur in the rule. This includes any phrase scripts, the condition script, and the action script. If no target is specified, the default target is used, and if no default has been set, then the default target is the Speech Monitor. Note that the script's target affects what terminology is available to the script. For example, if the values computed by phrase scripts refer to application-specific terminology, then either the script must explicitly use a `tell` statement or the rule must have a target specified. If no signature is specified in the target statement, the Speech Monitor becomes the default target for the rule's scripts.

If the `%acknowledge` statement is present, a standard (nonverbal) acknowledgment is executed according to the settings in the Speech Setup control panel.

The `%condition` statement specifies a script that evaluates to either `true` or `false` to determine whether to execute the action associated with this rule. Unlike `context`, this check is done after the rule has matched and thus is useful for resolving ambiguity when more than one rule matches the user's utterance.

The `%index` statement is used by the help system to allow the user to find speech commands that are relevant to a particular topic. This helps the user know what to say at any given point. As an example, a rule having to do with sending faxes may have the following index clause:

```
%rule
fax this to <person>
  .
  .
  .
%index email, fax, send
%end
```

The `%description` statement should be an English description (on as many lines as needed) that the help system can use to explain the effect of the command to the user.

The order of the `%` statements is flexible.

Phrases and AppleScript Clauses

A phrase defines a sequence (or set of sequences) of words that may be uttered by a user and recognized by the speech system. Syntactically, a phrase is specified as a sequence of space-separated tokens, each of which is either a word or a category reference. Words that contain nonalphabetic characters must be enclosed in quotation marks—for example:

```
"don't" save
```

A category reference consists of an opening angle bracket (<), followed by an optional label and a colon, followed by a category name, followed by a closing angle bracket (>)—for example:

```
<n:number>
```

If the value returned by the number category is not needed in the clause's script, then the label and its colon may be omitted:

```
<number>
```

If the label is used, it can be referred to as a property in the value script attached to the phrase (for category rules) or in the action script (for command rules)—for example:

```
%define ...
   .
   .
   .
print page <n:number>; {start: meaning of n, end: meaning of n}
   .
   .
   .
%end
```

or, alternately,

```
%rule
print page <n:number>
%action
... meaning of n ...
%end
```

In category rules, each phrase may have an associated script that computes its value. There is no way to specify another script type for the value scripts of phrases—AppleScript is the only script system currently supported.

For command rules, you can refer to the category reference variables in the action script itself, as illustrated in the preceding examples.

Internal Category Rules

The following format is used for defining internal categories:

```
%define name [ open ]
[%target [ signature ] ]
phrase [; value-script ]
   .
   .
   .
%end
```

In this format, *name* is the category name. It must be a single word without punctuation. The *phrase* and clause formats are explained in the previous section.

External Category Rules

The following format is used for defining external categories:

```
%define name external [dynamic] [ open ]
[%context [ context ... ] ]
[%target [ signature ] ]
%action [ script-type ]
script
%end
```

The `external` and `dynamic` properties are indications that the possible phrases of this category should be obtained by executing the script that follows, using the *script-type* indicated. If *script-type* is not specified, then the default will be used and consequently must have been specified in a prior default statement. The value returned by the script should be either a list of strings or a single string of phrases separated by newline characters. These values are then active phrases for use within the grammar.

If *context* is specified for an external (not dynamic) rule, the action will be reevaluated whenever that context becomes active.

The string containing the exact source text as opposed to the words spoken is an additional property that is present for external categories. It is available through the `source` property. For example, consider an external category that returns the names of all sounds stored in the System Folder, assuming the class `'sound'` is implemented in the application theApp:

```
%define sound external
%action AppleScript
tell application "theApp"
   name of every sound
end tell
%end
```

One of the sounds might have a name such as `Click1`. The pronunciation generated for this item might be something like *click one.* However, to cause the sound to be played, you need its exact name. For this reason, the Speech Monitor provides both the words that are recognized in the utterance property and the exact name of the item in the source property—for example:

```
%rule
play <s:sound>
%action AppleScript
utterance of s        →  "Click 1"
source of s           →  "Click1"
%end
```

The `dynamic` option of category rules causes the rule to be evaluated every time speech is detected. Extensive use of dynamic categories is not recommended, since this will slow down the apparent response time of the speech recognizer.

Here is an example category that loads the values of the first two cells of an Excel spreadsheet as possible phrases:

```
%define <possibleAnswers> external
%context application "Excel"
%action AppleScript
tell application "Excel"
  {value of cell 1, value of cell 2}
end tell
%end
```

Context Specifiers

Contexts may be specified for both command rules and external category rules. A *context specifier* is a declarative representation that tells when the rule should be considered to be active. A context specifier is simply a list of context descriptors, each of which must be "active" in order for the whole context specifier, and thus the rule, to be considered "active." The syntax for a context specifier is this:

```
%context [ context ... ]
```

Each of the one or more context descriptors can be one of the following:

```
application name
application id
window name
user name
suite id
```

The application context descriptor causes a rule to be active only when a particular application is active. The application may be specified by its name or its signature. If the name is specified, `CompileRules` looks for an application with that name and uses its signature. Examples of valid application contexts are

```
%context application "MyApp"        application named MyApp
%context application 'MACS'         the Finder
%context application *              any application
```

The window context descriptor causes a rule to be active only when a specific window of the active application is active. Windows can be identified by either their name, their kind (the `windowKind` field of `WindowRecord`), or (for dialog boxes) their resource ID. If a window descriptor is used by itself, without an accompanying application descriptor, the rule will be active whenever the descriptor matches the current context, regardless of what application it belongs to. Here is an example of a valid window context:

```
%context window "Speech Setup"        window named Speech Setup
```

The user context allows a rule to be active when a particular user name has been entered in the Sharing Setup control panel. This allows users to have their own rule sets on a machine that is used by more than one person—for example:

```
%context user "Bob Strong"            enable Pig Latin rules
```

Default Statements

The default statement can be used to define rule characteristics that apply to all following rules in the speech rules file. This alleviates the need to repeat the specification for each rule. Valid default statements have these forms:

```
%default context context …
%default script script-type
%default target signature
```

In the first case, the context descriptors are as specified in the context clause of rules— for example:

```
%default context application "MyApp"
```

In order to negate the effect of a default context inside a rule definition, simply specify the desired context, or an empty context if the rule is to be globally active:

```
%default context application "MyApp"
   .
   .
   .
%rule
```

```
hello
%context
%action
acknowledge saying "Yo"
%end
```

The default itself can be undone by specifying an empty context descriptor:

```
%default context
```

A default script type can be specified as follows:

```
%default script AppleScript
```

Subsequent scripts would not have to specify the script type in their condition and action clauses.

Global Scripts

A speech rules file may contain any number of scripts that are executed when the speech rules file is loaded. These are useful for defining handlers (subroutines) that are available to speech rules. Scripts defined using the `%global` clause are executed in a global context and thus can be used to define handlers and properties that are available to all rules, even in other files. Scripts defined using the `%local` clause are executed in a context that belongs to the speech rules file itself and thus cannot be shared with rules and scripts in other files:

```
%global [ script-type ]
   .
   .
   .
%end
```

or

```
%local [ script-type ]
   .
   .
   .
%end
```

If the optional script type is not specified, the default script type is used. It must have been previously defined. Any number of these clauses can appear in a file. Note, however, that scripts using different scripting systems cannot share handlers or properties.

CompileRules Error Messages

The following are descriptions of the error messages that can be generated by the `CompileRules` tool. For syntax errors, a line is printed with the filename and line number in a form that is suitable for executing in the MPW Shell (for example, by entering triple-click–Enter). Doing so will open the source file and set the selection to the line containing the error.

`Cannot create output file`

> The file you specified with the -o option could not be created. Possible reasons are that you specified a file on a locked volume or in a read-only folder; the volume containing the file you specified is full; or the startup volume, which is used as temporary storage during compilation, is full.

`Cannot find input file`

> One of the input files you specified does not exist.

`Command doesn't need any arguments`

> Indicates that an argument was encountered for the acknowledge clause. This clause does not take any arguments.

`Couldn't get file info for ...`

> One of the input files you specified could not be accessed.

`Didn't expect this:`

> During the processing of a rule phrase, some lexical element was found that didn't make sense. Typically this is caused by unpaired angle brackets, a missing label delimiter, an improperly quoted word containing special characters, or the like.

`Empty phrase script`

> A rule phrase was encountered that had the script delimiter (semicolon), but no script was found.

`Empty rule phrase`

> An empty rule phrase was encountered. All rule phrases must have at least one word or category reference.

`External rule shouldn't have a phrase`

> Indicates that a phrase was specified on an external category rule. External categories shouldn't have phrases, since their phrases are computed externally by a script.

`File is not a text file`

> One of the input files you specified was not a text file. `CompileRules` can compile only text files.

`Invalid context`

> Indicates that the context descriptor had invalid syntax. See "Context Specifiers," earlier in this chapter, for a description of valid context descriptor syntax.

`Missing category name`
> The category name is missing from a category definition. All category definitions must specify the category name.

`Missing context descriptor`
> Indicates that a context specifier was encountered that had no context descriptor. At least one context descriptor must be included.

`Missing default type`
> Indicates that the script type is missing from a default script statement.

`Missing script type`
> This error occurs when a script specifier occurs without a script type, when no default has been specified for the file. Either a default script type must be specified or every script specifier must include a script type.

`No input files specified!`
> You must specify at least one input source file.

`Premature end of phrase`
> Indicates that the end of a phrase was encountered when expecting a closing category reference delimiter. This is typically caused by a missing right angle bracket, label, or category name.

`Rule already has a condition`
> Indicates that more than one condition clause was found in a rule definition. Only a single condition may be specified for a rule.

`Rule already has a context`
> Indicates that more than one context was specified for a rule. A rule can have only one context specified, although that specification can contain multiple context descriptors.

`Rule already has an action`
> Indicates that more than one action clause was found in a rule definition. Only a single action may be specified for a rule.

`Trouble writing to temporary file...`
> An error occurred while writing to the temporary file used during compilation. Possible reasons are that the startup volume is full or a disk error occurred.

`Unknown category option`
> Indicates that one of the category options specified was unknown. Valid category options are `open`, `external`, and `dynamic`.

`Unknown or invalid command`
> Indicates that an unknown clause was encountered. The only valid rule clauses are `%context`, `%condition`, `%target`, `%acknowledge`, `%action`, and `%end`. The only valid clause for a global script is `%end`.

`Unknown rule file command`
> Indicates that an unknown speech rules file command was encountered. Valid speech rules file commands are `%default`, `%global`, `%local`, `%define`, and `%rule`.

```
Unknown scripting system
```
This error occurs when a script type is specified that is not registered with the system. This can occur when the script type is misspelled or when the scripting component was not installed at system startup time.

Apple Events Speech Events

The following defines the syntax of Apple events that are implemented by the Speech Monitor and can be invoked by speech rule scripts:

```
acknowledge [ success | failure | progress ]
  [ of hearing | recognizing | understanding | responding ]
  [ saying text ]
  [ caption text ]
reload rules [ file ]
```

An Example: A Simple Checkbook

Following are the complete rules for a simple checkbook grammar. For lack of a real application to control, the rules simply type the results into the Note Pad application.

```
%define uten
    one     ; 1
    two     ; 2
    three   ; 3
    four    ; 4
    five    ; 5
    six     ; 6
    seven   ; 7
    eight   ; 8
    nine    ; 9
%end

%define digit
    zero    ; 0
    oh      ; 0
    <x:uten> ; x
%end
```

```
%define tens
   twenty   ; 20
   thirty   ; 30
   forty    ; 40
   fifty    ; 50
   sixty    ; 60
   seventy  ; 70
   eighty   ; 80
   ninety   ; 90
%end

%define teens
   ten       ; 10
   eleven    ; 11
   twelve    ; 12
   thirteen  ; 13
   fourteen  ; 14
   fifteen   ; 15
   sixteen   ; 16
   seventeen ; 17
   eighteen  ; 18
   nineteen  ; 19
%end

%define utwenty
   <x:uten> ; x
%end

%define uhundred
   <x:digit>; x
   <x:teens>; x
   <x:tens> ; x
   <x:tens> <y:uten>; (meaning of x) + (meaning of y)
%end

%define uthousand
   <x:uhundred>; x
   <x:uten> hundred; (meaning of x) * 100
   <x:uten> hundred <y:uhundred>; (meaning of x)
      * 100 + (meaning of y)
   <x:uten> hundred and <y:uhundred> ; (meaning of x)
      * 100 + (meaning of y)
%end
```

```
%define money
    <x:uthousand> dollars; (meaning of x) * 100
    <x:uthousand> dollars and <y:uhundred> cents
        ; (meaning of x) * 100 + (meaning of y)
    <x:uhundred> <y:uhundred>; (meaning of x) * 100 +
        (meaning of y)
    <x:uten> <y:uhundred> <z:uhundred>
        ; (meaning of x) * 10000 + (meaning of y) * 100 +
            (meaning of z)
%end

%define merchant
    Emporium
    Sears
    JC Penney
    Marshalls
    Macys
    Nordstrom
    Pacific Gas and Electric
    Pacific Bell
%end

%rule
%context application "npad"
    pay <n:merchant> <x:money>
    pay <x:money> to <n:merchant>
%action applescript
    do menu "Clear"
    type "pay " & meaning of x & " to " & utterance of n
%index checkbook, pay
%description
    Pays the vendor the amount requested (actually simply types
        the vendor amount into the open Note Pad window).
%end

%rule
    open checkbook
%action AppleScript
    do menu "Note Pad"
%index checkbook
%description
    Opens the Note Pad to begin checkbook function.
%end
```

```
%rule
%context application "npad"
    close checkbook
%action AppleScript
    do menu "Quit"
%index checkbook
%description
    Closes the Note Pad to stop checkbook function.
%end
```

The rule to pay <merchant> <money> uses the Speech Monitor to actually type the result into the Note Pad. No "tell application" block is needed, because the Speech Monitor sets itself to be the default application. Ideally, the Note Pad should be AppleScript-aware, so the script could be

```
tell application "Note Pad"
    set ourResult to "pay " & (meaning of x) & " to " &
        (utterance of n) & "\r"
    copy ourResult to contents of selection
end tell
```

and no doMenu or type command would be needed.

System Software Modifications

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* covers miscellaneous changes to the system software in the Macintosh Quadra 840AV and Macintosh Centris 660AV, including a new manager for the internal and external SCSI (Small Computer System Interface) ports. It contains five chapters:

- Chapter 9, "SCSI Manager 4.3," describes the new SCSI Manager in the Macintosh Quadra 840AV and Macintosh Centris 660AV.

- Chapter 10, "DMA Serial Driver," covers the new hardware-independent serial driver that uses direct memory access (DMA).

- Chapter 11, "Video Driver," describes changes to the video driver for the Macintosh Quadra 840AV and Macintosh Centris 660AV.

- Chapter 12, "New Age Floppy Disk Driver," lists changes to the floppy disk driver and tells you how they affect floppy disk compatibility with other Macintosh computers.

- Chapter 13, "Virtual Memory Manager," details how the Virtual Memory Manager no longer disables interrupts when performing certain tasks.

SCSI Manager 4.3

This chapter describes the new SCSI Manager architecture for the Macintosh Quadra 840AV and Macintosh Centris 660AV computers. It contains functional specifications describing the features, interface, compatibility, and performance of the new SCSI Manager. For hardware details of SCSI support in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see "SCSI Connection," in Chapter 2.

In addition to the capabilities of the former SCSI Manager, the new SCSI Manager

- supports major new SCSI features such as disconnect and reconnect

- supports services such as fully asynchronous SCSI input and output

- provides a more hardware-independent API that minimizes the SCSI-specific tasks that a device driver must perform

- provides full use of whatever SCSI hardware is available

- supports existing SCSI device drivers with minimum or no modifications

This chapter starts with "SCSI Manager 4.3 Features," which describes the current feature set, compatibility issues, performance, and some of the developer issues raised by changes to the SCSI Manager. Everyone should read this section.

"Design Overview" describes the layered structure of the new SCSI Manager and lists the general functions provided by each of the layers.

"Implementation" describes specific hardware and software dependencies of the new SCSI Manager. Compatibility with the previous SCSI Manager, the virtual bus, and data transfer methods are also discussed here.

Two sections, "Guidelines for SCSI Device Driver Developers" and "Guidelines for SIM/HBA Developers," contain information for specific types of developers. If you are developing either a SCSI device driver, a SCSI interface module (SIM), or a host bus adapter (HBA) you should read these sections.

Finally, "SCSI Manager 4.3 Reference" discusses the actual API for the SCSI Manager 4.3, and "Summary of the SCSI Manager 4.3" lists its code interface.

SCSI Manager 4.3 Features

The SCSI Manager 4.3, in its Macintosh Quadra 840AV and Macintosh Centris 660AV release, supports the following new or improved features.

- *Parameter-block programming interface for SCSI I/O requests.* A parameter block contains all the information required to complete each SCSI I/O transaction. Additionally, this interface provides a hardware-independent view of the SCSI Manager. This independence allows the same device driver to work with any supported SCSI controller.

- *Asynchronous SCSI I/O.* SCSI Manager 4.3 handles both synchronous and asynchronous I/O requests. In addition, it allows multiple device drivers to maintain multiple outstanding requests.

■ *Phase-cognizant implementation.* SCSI Manager 4.3 follows the phases driven by the target and performs the appropriate operations as specified in the SCSI I/O request parameter block. Driver clients no longer need to worry about SCSI bus phases. This eliminates a major source of development difficulties found in the old SCSI Manager.

■ *Disconnect/reconnect features.* Disconnect/reconnect capability helps maximize SCSI bus utilization. It allows a device to disconnect and release control of the SCSI bus while the device processes a command from the host and to reconnect when the device is ready to communicate with the host. This allows the computer to submit requests to multiple targets so that those requests are executed in parallel. An example of this is a disk array application that can issue a request to one disk, which disconnects, and then issue another request to a different disk. The two disks can be performing seek operations simultaneously, thereby cutting down on the average seek time.

■ *Parity support.* For the first time, parity is completely supported. This applies both to transmission of parity (which has always been the case) and to detection and handling of bad parity on reception. For compatibility reasons, a client can disable the parity detection on a per-transaction basis.

■ *SCSI-2 support.* All SCSI-2 mandatory messages and protocol actions are supported as defined for an initiator. In addition, there are several optional features, such as disconnect/reconnect, that are also supported. There are optional SCSI-2 hardware features, such as Fast or Wide SCSI, that are anticipated by the architecture and API of SCSI Manager 4.3; when compatible hardware is available, device drivers will not have to be modified to take advantage of it.

■ *Autosense feature.* The SCSI Manager automatically performs a request sense operation in case of a check condition and retrieves the sense data. This provides support for contingent allegiance conditions and unit attention conditions.

■ *SCSI direct memory access.* SCSI Manager 4.3 can make use of any onboard direct memory access (DMA). This feature allows the host to perform other functions while data bytes are transferred to or from the SCSI bus.

■ *Full support for multiple buses.* SCSI Manager 4.3 supports a full complement of devices on each available SCSI bus; this support allows a CPU with internal and external SCSI buses to access up to 14 SCSI targets instead of 7. In addition, third parties can create NuBus or PDS cards, with advanced SCSI adapters, that drivers can access through SCSI Manager 4.3 in exactly the same manner as through the standard SCSI bus. Users install a faster SCSI bus on an accessory card and move some or all of their SCSI devices to the new bus. Those devices will continue to work even if the SCSI devices, the SCSI drivers on those devices, and the SCSI bus are all made by different vendors. If these new SCSI adapters use 16-bit or 32-bit buses, all 16 (or 32) targets are addressable.

■ *Full support for multiple logical units on each target.* SCSI Manager 4.3 allows full access to all 0–7 *logical unit numbers (LUNs)* on a target. These LUNs are treated as separate entities—I/O requests are queued according to LUN, and each LUN can maintain its own internal request queue (when target queuing is supported).

Compatibility

SCSI Manager 4.3 fully supports the current Macintosh SCSI Manager interface. It supports all calls and transfer information block (TIB) pseudoinstructions, except for `scComp` (compare) which is very rarely used. The lack of `scComp` is because of the support for DMA, which does not easily permit compare operations. Future implementations of the SCSI Manager are not guaranteed to maintain this level of compatibility with the old SCSI Manager API.

▲ **WARNING**

Applications or drivers that bypass the current SCSI Manager for any part of a transaction are not supported and will probably result in a fatal error. ▲

System Performance Impact

The performance impact of the SCSI Manager can be viewed from several different perspectives. Viewed from a raw byte-to-byte transfer level on the SCSI bus, SCSI Manager 4.3 performs like the old SCSI Manager. This is mostly a hardware issue—the performance of the SCSI Manager is tied to the level of performance of the hardware underneath. For information about Macintosh Quadra 840AV and Macintosh Centris 660AV SCSI performance, see "SCSI Connection," in Chapter 2.

However, viewed from the system level, the asynchronous capability provides significant increases in performance by allowing SCSI clients to regain control of the system while a SCSI I/O request is in progress. In addition, the support for disconnect/reconnect allows the system to have multiple I/O requests in operation on multiple targets concurrently, allowing another significant gain.

Because the Macintosh Quadra 840AV and Macintosh Centris 660AV hardware supports DMA, SCSI Manager 4.3 allows even more CPU cycles to be used for non-SCSI activity while a SCSI transaction is in progress. Just how many more depends on how much time is spent transferring data bytes. Another, though smaller, factor is how much the CPU uses the memory bus during DMA operations, because the DMA and CPU contend for the bus. An example of this factor is the relative difference between 68030 and 68040 bus use. Compared to the 68030, the 68040 has a higher cache hit rate (due to the larger cache) and a correspondingly smaller bus usage for the same set of instructions.

Impact on Developers

Two main product areas can take advantage of the new features of SCSI Manager 4.3: drivers for SCSI devices and add-on SCSI buses.

Almost all existing drivers and other clients of the SCSI Manager will continue to run without problems, as described in "Compatibility," earlier in this chapter. But most developers will want to modify their drivers to make use of the features of the new

manager. "Guidelines for SCSI Device Driver Developers," later in this chapter, provides general principles critical to developers of drivers.

Developers of add-on SCSI bus adapters clearly gain from the new architecture. These NuBus adapter cards provide one or more additional SCSI buses, each of which typically provides higher throughput and/or enhanced capabilities beyond the hardware supplied by Apple on the main logic board. In the past there was no standard software interface for accessing more than one SCSI bus or for accessing the advanced features of these buses. Because of this, developers of these cards have had to provide their own SCSI Manager, which controlled only their SCSI bus.

The new SCSI Manager 4.3 architecture improves this situation significantly, as further described in "Design Overview," later in this chapter. The makers of SCSI adapter cards can continue to supply the software for controlling their bus. But now, this software accesses the SCSI Manager 4.3, providing it the ability to direct I/O requests to that bus. The clients of the SCSI Manager (drivers or applications) can access a SCSI device in the same manner, with the same calls and parameters, whether that device is connected to the Apple SCSI bus or to a third-party NuBus SCSI adapter. "Guidelines for SIM/HBA Developers," later in this chapter, explains general principles critical to this effort.

The support for third-party add-on SCSI buses provides another incentive for driver writers to recode. With the new SCSI Manager's application programming interface, their drivers will work with devices that are on any available SCSI bus.

Design Overview

This section provides a high-level overview of SCSI Manager 4.3.
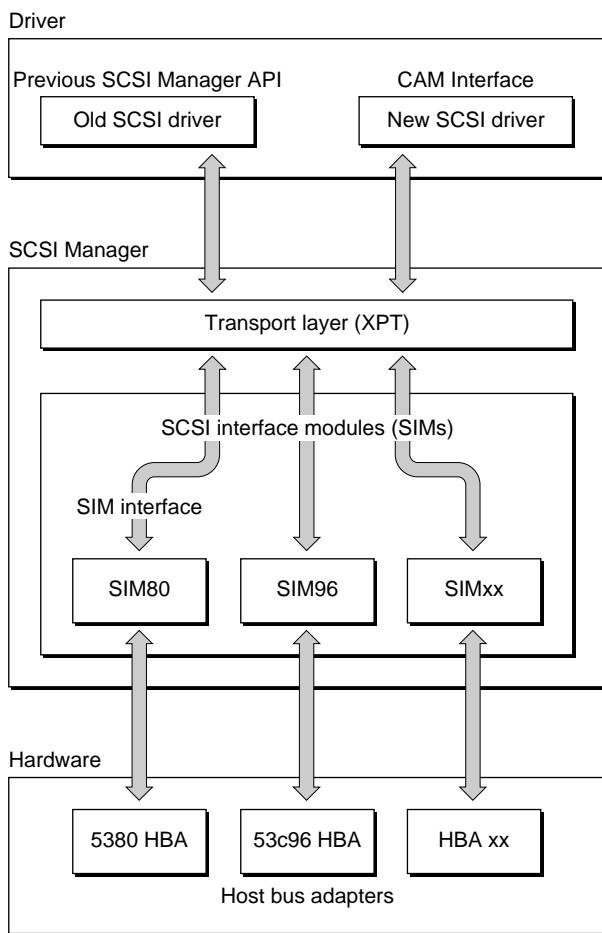
General Concepts

The SCSI Manager 4.3 application programming interface strongly resembles the software interface specified by ANSI X3T9 in the Common Access Method document (CAM). The SCSI Manager 4.3 interface however, contains Apple-specific areas for backwards compatibility and conformity with the Macintosh operating environment.

Intrinsic in the CAM-like interface is a CAM-like design. In CAM, there are two main layers—the transport (XPT) layer and the SIM layers. The XPT sits on top of multiple SIMs. Each SIM is responsible for controlling one host bus adapter (HBA), which constitutes the hardware associated with a specific SCSI bus adapter. There are a few requests that are handled entirely in the XPT layer, but in most cases the XPT simply passes the request to the SIM that has been registered to handle the HBA specified in the request.

These relationships are diagrammed in Figure 9-1.

Figure 9-1   SCSI Manager software hierarchy

Those readers familiar with the CAM document should note that Apple has adopted alternatives to some of the terms used by CAM. Table 9-1 shows the terms, their meanings, and their Apple equivalents.

Table 9-1   CAM to ACAM terminology conversion

CAM	Apple	Meaning
HBA	Bus or HBA	HBA is an acronym for host bus adapter, which contains all the hardware associated with a single SCSI bus adapter. This could be a bus on the main logic board, a NuBus card with a SCSI bus adapter, or any other native or attached SCSI bus. If there is DMA circuitry associated with that bus, it is also considered part of the HBA or bus.

continued

Table 9-1 CAM to ACAM terminology conversion (continued)

CAM	Apple	Meaning
Path	Bus	CAM uses "Path" to specify a particular HBA or bus attached to the system. Similarly, CAM functions and parameter blocks frequently include a `Path_ID` which is renamed `BusID` in this chapter.
CCB	SCSI_PB	CAM control blocks are the same as Apple's SCSI parameter blocks or `SCSI_PB`.
In	Out	When referencing routine parameters, CAM uses the SCSI convention of direction with respect to the initiator. This is backward from the standard way of describing parameters and results for functions. For instance, if the SCSI Manager has a function with an input parameter `BusID`, this is typically considered "in," but CAM refers to its direction as "out." This is because, in SCSI, if a parameter is sent to the XPT, it is sent toward the target (away from the initiator), or "out." Likewise, results returned from functions are considered "in" by CAM but are referred to as "out" in this chapter.
Out	In	See comment above.

Transport Layer

There is one XPT layer per system. This forms the access point for all clients of the SCSI Manager and has the following responsibilities:

- Provide the means to register HBAs, their characteristics, and their respective software entry points in SIMs.

- Route the request (parameter block or `SCSI_PB`) to the proper SIM.

- Provide higher-level facilities for old SCSI Manager interface compatibility. This consists of maintaining a translation list of `SCSISelect` IDs and their corresponding HBAs and directing them accordingly.

- Provide Operating System (OS) services to SIMs to isolate SIMs from OS dependencies. Such services include registration of interrupt handlers, static data space allocation and deallocation, and so on.

SCSI Interface Modules

Beneath the XTP layer lies one or more SIMs, each of which is responsible for interpreting the requests directed to it by the XPT. Each SIM "owns" one HBA. If there are multiple identical HBAs, there will be multiple identical SIMs. It is important to realize that a SIM designates not a code entity (such as a code resource), but instead represents the process or task which is responsible for controlling a particular HBA. For instance, if one SIM is coded for the Macintosh Quadra 900 (which has two identical 53c96 chips), there will be two SIM "instances," one for the internal bus and one for the external bus.

The SIM's responsibilities can be broken into three main areas: queue maintenance, bus servicing, and assorted software services. The bus service routines are HBA-specific. The other two areas are very similar between various SIMs. Specifically, the SIM handles:

■ queuing of multiple operations for all LUNs on same and on different targets and assigning tags for tag queuing (when supported)

■ maintaining the queue, including freezing and unfreezing for queue recovery as necessary

■ posting completed operations back to the requesting client (callback to device driver or application)

■ managing the selection, disconnection, reconnection, and data pointers of the SCSI HBA protocol

■ performing all interface functions to the SCSI HBA

■ managing the data transfer path hardware (SCSI bus), including DMA circuitry and address mapping, and establishing DMA resource requests

■ distinguishing abnormal behavior and performing error recovery, as required

■ providing a time-out mechanism for tracking `SCSI_PB` execution using values provided by the peripheral driver

■ supporting old SCSI Manager calls (optional on a SIM-by-SIM basis)

CAM Deviations

Apple has used the Common Access Method as a guideline during the creation of the SCSI Manager 4.3. CAM was never an attempt to provide either source-level or binary-level compatibility between different platforms. Considering this, it was viewed as more beneficial to provide a SCSI Manager interface that fit in with other Macintosh interfaces than to provide one that was similar to those on DOS or UNIX® platforms.

Implementation

The mechanics of issuing a call is slightly different than with the old SCSI Manager—instead of the stack-based `SCSIDispatch` trap, a register-based A-trap is used (`SCSIAtomic`). Several routines are accessed through one A-trap, distinguished by a routine selector word that is now one of the register parameters. C and Assembler macros and glue code are available that allow each of the routines to be called with a single line of code.

There are several routines that can be called in this manner. `SCSIRegisterBus` is used by a SCSI interface module during initialization to inform the XPT of its presence and its ability to handle SCSI requests. Various entry points and details of the SIM are passed to the XPT in a `SIMinitInfo` parameter block and, after registration, the XPT fills in other fields in the same parameter block, specifying details of the registration required by the SIM.

A `SCSIDeregisterBus` routine is provided to undo the effect of the
`SCSIRegisterBus` routine. This is not likely to be needed in the current Macintosh
environment.

The third routine, `SCSIAction`, is used by clients of the SCSI Manager to issue all other
requests. Beside the selector word designating `SCSIAction`, the only other parameter is
a pointer to a SCSI parameter block (`SCSI_PB`). A variety of functions are requested
through this parameter block interface, the most important being `SCSI_ExecIO`,
the function used to request a complete SCSI I/O transaction. Most of the functions
available have corresponding SCSI parameter blocks defined to carry the input
parameters as well as the results. For instance, the `SCSI_BusInquiry` function requires
a `SCSI_BusInquiry_PB` parameter block, a pointer to which is passed to `SCSIAction`.

The `SCSI_ExecIO_PB` parameter block contains the destination of the SCSI I/O request
(which bus, target, and logical unit), the command descriptor block (CDB), the descrip-
tion of the data buffer(s) which either supply or receive the data, and the results of the
operation, as well as a variety of other fields and flags required to completely specify
the transaction.

Different SIM implementations may require additional fields beyond the standard public
fields in the `SCSI_ExecIO_PB` parameter block. Some of these may be input or output
fields providing access to special capabilities provided by a SIM or they may be fields
private to the SIM required during the processing of this request. In either case, the
`SCSI_ExecIO_PB` parameter block may vary in size depending upon which bus is
being addressed. In order to determine the size for a particular bus, a client must issue a
`SCSI_BusInquiry` request (also through `SCSIAction`) which returns the size of the
`SCSI_ExecIO_PB` parameter block as well as additional information about the specified
bus. An appropriately sized parameter block can then be allocated, filled out, and issued
to that bus to perform the requested `SCSI_ExecIO` function.

`SCSI_BusInquiry` is also used by a client to determine various hardware and software
characteristics of a SIM/HBA. This may be required to adequately form a request that
takes advantage of all of the SIM's capabilities.

`SCSI_ExecIO` calls can be made either synchronously or asynchronously. If the call is
asynchronous, the caller may determine whether the action is complete either by
checking the result field of the `SCSI_ExecIO_PB` parameter block or, preferably, by
supplying an address of a completion routine (callback). Because of special interrupt
handling considerations, you should issue all `SCSI_ExecIO` requests that need to be
performed synchronously directly to the SCSI Manager, rather than issuing them
asynchronously and then performing a sync-wait action in the client program. More
details of this requirement can be found in "Guidelines for SIM/HBA Developers," later
in this chapter.

When an error occurs during a `SCSI_ExecIO` request, the SIM may freeze the queue
for the LUN on which the error occurred, to allow the client to perform any required
error recovery. Upon completion of the recovery process, the client should issue a
`SCSI_ReleaseQ` request to reenable the normal handling of I/O requests to that LUN.

Other `SCSIActions` that may affect SCSI devices are the `SCSI_ResetBus`,
`SCSI_ResetDevice`, `SCSI_AbortCommand`, and `SCSI_TerminateIO` functions.

There are a class of SCSIAction requests that are used by SCSI device drivers and SCSI utilities to determine which Macintosh device drivers are responsible for which SCSI devices. These are SCSI_SetRefNum, SCSI_RemoveRefNum and SCSI_GetNextRefNum. These routines allow a client to maintain or examine the relationship between driver refNums and SCSI DeviceIdents (bus+target+LUN).

Target mode (although not supported in the initial SIM implementation) would be accessed through the SCSI_EnableLUN and SCSI_TargetIO functions.

Optional Features Not Supported in the SIM

If any of the following functions are requested of the SIM in the Macintosh Quadra 840AV or Macintosh Centris 660AV, an appropriate error code is returned:

■ *Synchronous data transfer.* The SIM does not initially support synchronous data transfer. Any I/O requests designating synchronous data are rejected with a status of "Unable to provide the requested capability" and can simply be reissued without the synchronous data transfer request.

■ *Target command queuing.* The SIM does not support SCSI-2 target queuing in its initial release. Eventual support is planned. Until that time, any I/O requests with the QueueActionEnabled bit set are rejected.

■ *HBA engine support.* None of Apple's SIMs support HBA engines. A bus inquiry returns an engine count of zero. The engine inquiry and execute engine request functions return request completed with an error value.

■ *Target mode.* Target mode is not currently supported.

Although these features are not supported in the initial implementation, third-party SIMs could provide support for these items because the XPT layer still delivers requests of these types to all installed SIMs.

Compatibility and Emulation

The old SCSI Manager routines (SCSIGet, SCSISelect, SCSIComplete, and so on) will continue to work under SCSI Manager 4.3 with very few compatibility problems.

A SIM/HBA may or may not be capable of supporting the old routines. When a SIM registers its HBA with the XPT, it must identify its oldCallCapable status—its ability to support the old routines or not. All Apple-supplied SIM/HBAs are capable of handling old calls.

SCSIGet calls set a flag that prevents any additional SCSIGet calls but perform no other operation. Upon the receipt of the SCSISelect call, the XPT issues a SCSI_OldCall request to the SIM which places it, like all other SCSI_ExecIO_PB parameter blocks, in its queue. Any SCSI parameter blocks that are awaiting initial execution (as well as any received while the old API transaction is in effect) are queued and do not begin execution until after a SCSIComplete is received and completed, which is when the queue is released. Any additional SCSIGet or SCSISelect calls received after an old API emulation has already begun are rejected with an error.

While the old API emulation is in progress, SCSI parameter blocks resident in or destined for queues in other HBAs are not affected—they continue to be executed as requested.

The old SCSI Manager routines supported by SCSI Manager 4.3 are listed in Listing 9-1.

Listing 9-1   Supported old SCSI Manager routines

```
OSErr SCSIReset (void);
short SCSIStat (void);
OSErr SCSIGet (void);
OSErr SCSISelect (short targetID);
OSErr SCSISelAtn (short targetID);
OSErr SCSICmd (Ptr buffer, short count);
OSErr SCSIRead (Ptr tibPtr);
OSErr SCSIRBlind (Ptr tibPtr);
OSErr SCSIWrite (Ptr tibPtr);
OSErr SCSIWBlind (Ptr tibPtr);
OSErr SCSIMsgIn (short *message);
OSErr SCSIMsgOut (short message);
OSErr SCSIComplete (short *stat, short *message, unsigned long
        wait);
```

SCSIReset calls are executed synchronously, but they can reset only buses that are controlled by SIMs that are capable of handling old calls.

The SCSIStat routine works and returns results as accurate as possible for the current old-call bus. If there is no current old-call bus, then the result indicates bus-free. If it is difficult for SCSI interface modules (SIM/HBA) to determine the exact state of the REQ signal during certain periods, for instance between the functions provided by old API calls, the SIM can be written so that the it does not return control to the XPT (for example, with an rts instruction) until a valid phase is on the bus.

There are several variances in support for transfer instruction blocks (TIBs). The first affects the scComp (compare) instruction, which is no longer supported in SCSI Manager 4.3. This results from the support for DMA hardware, which does not permit a compare operation. This should pose few compatibility problems, since it is rarely used; there were several previous versions of the SCSI Manager that did not perform compare operations properly.

Virtual Bus

SCSI Manager 4.3 has explicit support for multiple buses (HBAs), allowing a client to specify a target based on its bus number as well as its target ID and LUN. To support old API calls which understand only a target ID, the technique first used in the Macintosh Quadra 900 is expanded to include not only built-in SCSI buses but add-on NuBus and PDS buses as well.

In the Macintosh Quadra 900, SCSI transactions are directed to the first bus which responds to a select for the requested ID. The ID specified in a `SCSISelect` routine is called the "virtual ID" because it designates a device on the single "virtual bus" (which encompasses both internal and external buses). When a `SCSISelect` call is made, a selection of the virtual ID is attempted on the internal bus first, and if there is no response, the selection is attempted on the external bus. Once a successful selection of a virtual ID occurs, all subsequent `SCSISelect` calls are directed to the bus on which that selection occurred. Until a successful selection has occurred on one of the buses, the virtual ID is not assigned to a particular physical bus. Once established, a virtual ID to physical bus mapping is not changed until restart.

The virtual bus is maintained by the compatibility portion of the XPT layer, which determines the virtual-to-physical ID mapping. The XTP layer does this by attempting to select the virtual ID on each of the HBAs that can handle old calls until it finds a device that responds. The HBAs are searched in order of registration, which in most cases is first internal, then external, then additional third-party add-in SCSI cards. The Macintosh Quadra 840AV and Macintosh Centris 660AV have no separate external bus.

Data Transfer Descriptions

Clients of SCSI Manager 4.3 can use several different structures to describe the source (or destination) memory buffers for data transfer to (or from) a SCSI device. The easiest is the single buffer descriptor, consisting of a buffer address and a buffer length. A more difficult descriptor is needed when there are discontiguous areas of memory that make up a client's data transfer. These are specified by a single scatter/gather (S/G) list. Buffer descriptors include address and length. There is an additional parameter for how many items are in the S/G list.

In the old SCSI Manager, the TIB is made up of a series of transfer instructions. During the execution of a `SCSIRead`, `SCSIWrite`, `SCSIRBlind`, or `SCSIWBlind`, the TIB instructions (transfer and increment address, transfer and don't increment, add longword to address, move longword, loop, compare, and stop) are interpreted by the SCSI Manager to determine the source and destination of the data. Additional details can be found in *Inside Macintosh*, Volume IV.

Although S/G lists are simpler than TIBs, TIBs were actually designed for an additional purpose—they are also used to show the SCSI Manager where long delays (greater than 16 µs) may be found in the data transfer. This was required for the Macintosh Plus because of the lack of hardware handshaking between the SCSI chip and the CPU. It was required in all later Macintosh computers as well for slightly different reasons. SCSI Manager 4.3 only supports TIBs for old API calls. All new API calls must specify either a single buffer descriptor or S/G lists.

A second aspect of the TIB is used by the `scHandshake` field of the `SCSI_ExecIO_PB` parameter block. This field is a series of words, each of which specifies the number of bytes between potential delays in the SCSI data transfer. For instance, `1,511 TIB` is a common TIB structure needed to work with drives which have a 512 byte block and sometimes have a delay between the first and second bytes in the block as well as a delay between the last byte of a block and the first byte of the following block. This TIB

structure translates to an `scHandshake` of 1, 511, 0… which translates to a request to transfer 1 byte, synchronize then transfer 511 bytes, synchronize, transfer 1 byte, and so on. As can be seen from the example, this structure is null-terminated and can have a maximum of 8 byte counts/handshake points.

Guidelines for SCSI Device Driver Developers

SCSI device drivers and other clients written to take advantage of the SCSI Manager 4.3 must continue to perform all of the operations associated with their counterparts when dealing with the old SCSI Manager. In addition, they will be have to follow some new rules that asynchronous data transfer and multiple-bus support require.

A SCSI device driver interfaces with its client (typically through the Device Manager) and with the SCSI Manager. Because the old SCSI Manager was completely synchronous, SCSI drivers were synchronous as well. If a driver is rewritten to issue asynchronous SCSI requests, it can also be rewritten to behave asynchronously as well (with respect to its clients). This is critical in order for the benefit to reach the application level.

Booting and Drive Mounting

For earlier ROMs, the OS scans the SCSI bus from ID 6 to ID 0, looking for all devices that have an `Apple_HFS` as well as an `Apple_Driver` partition. When one is found, the driver is loaded and executed and installs itself into the unit table. The driver then places an element in the drive queue for any HFS partitions that are on the drive. The Start Manager then records these `DrvQElements`, mounts the startup volume, and attempts to start up the computer.

There are six unit table slots ($20 through $26) reserved for SCSI drivers for devices at IDs 0 through 6 respectively. In the Macintosh Quadra 840AV and Macintosh Centris 660AV architecture, which allows the addition of many SCSI buses, device drivers must be able to allocate their unit table slots dynamically. All SCSI drivers, including drivers written for SCSI Manager 4.3, must attempt to install themselves in the unit table at the same location specified under the old calls, `$20+SCSI_ID`. This attempt lets both old and new drivers serve as boot devices when booting from an earlier machine. If a driver finds that the slot is already full, it should search for an empty slot in the `48 ($30)` to `UnitNtryCnt` range.

SCSI Manager 4.3 is able to distinguish between old drivers that support only the old API and new drivers that are aware of SCSI Manager 4.3. Drivers aware of SCSI Manager 4.3 are identified in the partition map as type `Apple_Driver43` instead of the previous type, `Apple_Driver`. Because the old ROMs checked only the first 12 characters of the type before loading and executing the driver, both new and old drivers will work on older machines.

Once the drivers have been loaded and executed, the ROM searches for the default startup device in the drive queue. If it is there, it is mounted and the boot process begins. In the Macintosh SE ROM and all ROMs since, the boot drive is identified by a driver

reference number. The unit table slots for SCSI drivers are always in the range $20 through $26—the slots set aside for SCSI drives at IDs 0 through 6 respectively. This works fine when drivers have the same `refNum` between boots but, in the Macintosh Quadra 840AV and Macintosh Centris 660AV, drivers allocate unit table slots dynamically.

Currently, the driver reference number (a word) is stored in parameter RAM (PRAM) and is used by the Start Manager to pick the startup device. However, SCSI Manager 4.3 designates the startup device by using `DeviceIdent` (containing the bus number, SCSI ID, and LUN of a device), which supports multiple buses. To access devices on multiple-bus CPUs, you must know which device to boot from and whether the external device can be mounted at `LateLoad` time.

Some devices may be "hidden" from access to the old API calls if a device with the same ID is found on a higher-priority HBA. With earlier ROMs, the old SCSI Manager loads all drivers that are found. On multiple-bus machines, this does not include those devices with the same IDs as devices on higher priority buses. When SCSI Manager 4.3 is running, access to those devices can be made only through the new API and hence only by new drivers (`Apple_Driver43`). `LateLoad`, which runs after SCSI Manager 4.3 has been installed, scans all known buses and load all additional new drivers found, using the new API. The new drivers then mount their respective drives. SCSI Manager 4.3 also loads additional old drivers if those drivers are accessible by emulating the old API.

When SCSI Manager 4.3 is present at startup (in ROM), all new (`Apple_Driver43`) drivers are loaded from all drives found. Then pre-4.3 (`Apple_Driver`) drivers are loaded if they are found on a device (accessible via emulation of the old API) that corresponds to the virtual-to-physical mapping for their SCSI ID.

If a pre-4.3 ROM loads an `Apple_Driver43` driver, it treats it exactly like an `Apple_Driver` driver. This means that during initialization, the Start Manager makes a call to the beginning of the driver (defined as the first byte) with register D5 set to the SCSI ID of the device the driver was loaded from. To provide complete compatibility, an additional entry point has been defined for `Apple_Driver43` at 8 bytes from the start. If this entry point is called, it means that SCSI Manager 4.3 is present and that a `DeviceIdent` value is in register D5. No other registers are valid.

There could be situations where SCSI Manager 4.3 becomes active after an `Apple_Driver43` driver is loaded. This would occur if a newer system was patching an older ROM; the old ROM would load and install the driver and the system would come up later, installing SCSI Manager 4.3. To recognize the appearance of the new SCSI Manager, every `Apple_Driver43` driver should check for the presence of the `_SCSIAtomic` trap ($A089). The best place to do this check is at the first `accRun` tick (from `dNeedTime` flag). This tick happens after the system patches are in place.

Asynchronous Behavior

The successful execution of asynchronous I/O requires a whole set of rules that were not a concern when dealing with the synchronous SCSI Manager. The general form of an asynchronous SCSI driver is different than that of an old synchronous driver. When a client makes an I/O request, the Device Manager queues that request in the driver's I/O

queue and then make a call to the driver's `_Prime` routine. That routine should stuff a `SCSI_ExecIO_PB` parameter block (PB) with the parameters necessary to complete the request (or multiple PBs if required) and send them to the SCSI Manager through `SCSIAction`. The proper SIM then adds the request to its queue and possibly start working on it before returning back to the driver.

At this point, virtually nothing can be assumed by the driver about the request. If it was accepted it may have only been queued or it may have proceeded all the way to completion. If the return value is a value other than `noErr`, it is the result of input parameter errors. Either the parameter block was built incorrectly or it contains an error in one or more parameters. If the return value provided by `SCSIAction` is `noErr`, the command has been accepted and the contents of the `SCSI_ExecIO_PB` are no longer valid. This is because of the asynchronous nature of the SCSI Manager.

▲ **WARNING**

Once a parameter block has been accepted by the SCSI Manager, no attempt should be made by the driver to read it. The current parameter block being worked on by SCSI Manager 4.3 may be from a different request and completely incorrect for the driver. ▲

Typically a callback routine is supplied with the `SCSI_ExecIO_PB` parameter block. This routine allows the SIM to asynchronously notify the client that the request has completed. SCSI drivers must always use such callbacks.

▲ **WARNING**

SCSI drivers must always use a callback routine when issuing asynchronous requests. If a callback routine is not supplied the client cannot be notified asynchronously. Being notified asynchronously would require that the driver perform a sync-wait action, which is not permitted because of virtual memory compatibility factors. ▲

For SCSI requests that can be handled synchronously, such as those required during error handling or initialization, the driver should issue those requests to the SCSI Manager synchronously, rather than asynchronously, and then wait for the `scResult` field to change from `scsiReqInProg`. The latter process is effectively a synchronous request, except that the sync-wait is performed in the driver. This is not allowed. A further explanation is given in the next section.

An asynchronous I/O request issued by a client to a driver may occur at interrupt time. This eliminates the possibility of allocating memory to handle the request at the time the request arrives. This means that any `SCSI_ExecIO_PBs`, S/G lists and any other structures that are needed for the processing of the I/O should be allocated at driver initialization time. Unlike a synchronous request, none of these can be allocated on the stack because they would disappear when the driver returned from the `_Prime` routine.

When issued asynchronously the resulting action may start at any time and may end at any time. There is no implied ordering of these events with respect to earlier or later requests. An earlier request may be started later, or a later request may complete earlier. However, a series of requests to the same device (bus ID + target ID + LUN) is issued to that device in the order received.

Virtual Memory Operation

There is a possibility that an application may disable interrupts and then cause a page fault. Because this page fault translates to a synchronous SCSI driver request, the SCSI Manager handles the resulting SCSI request without the benefit of interrupts. The Macintosh Quadra 840AV and Macintosh Centris 660AV require that all sync-waits be performed either in the SCSI Manager or in the Device Manager where there are hooks that provide the sync-wait loop the ability to poll the SCSI interrupt sources.

If a driver has received a synchronous I/O request (typically from `_Prime`), the driver has two options. Either it can issue the subsequent SCSI Manager request synchronously as well, or it can issue the SCSI request asynchronously and simply return back to the Device Manager. The Device Manager sits in a sync-wait loop, awaiting the completion of the request. The driver should call (or jump to) `IODone` after it receives the SCSI completion callback. If the single driver request translates to multiple SCSI requests, the driver can issue each of those requests synchronously or it can issue them asynchronously and return back to the Device Manager. The driver should, in this case, call `IODone` after the callbacks for all of the SCSI requests have been received.

▲ **W A R N I N G**
Under no condition should the driver use a sync-wait loop.
If it does, the SCSI Manager will never be allowed to clear the
interrupts and will hang indefinitely. ▲

Never perform sync-waiting in the driver itself. The wait must be controlled by the Device Manager (by returning from the `_Prime` routine) or in the SCSI Manager (by issuing the SCSI request synchronously).

As explained in Chapter 13, "Virtual Memory Manager," virtual memory (VM) in the Macintosh Quadra 840AV and Macintosh Centris 660AV executes I/O completion routines, Time Manager tasks, VBL and slot VBL tasks, deferred tasks, and `PPostEvent` actions without disabling interrupts. If a completion routine is to be run while VM is running the deferred user function queue (with interrupts enabled), VM queues this new completion routine at the tail of the deferred user function queue. This assures that routines of the types listed above will execute in their original order.

The SCSI completion routines (callbacks from `SCSI_ExecIO`) are similar to `IOCompletion` routines except for one major difference. Because they can cause page faults and typically occur at "interrupt time," `IOCompletion` routines are handled by VM; if it's safe for paging at the time the call to `IODone` is made, the `IOCompletion` routine is executed immediately. If it isn't safe for paging, VM defers execution of the `IOCompletion` routine until it is safe.

Like `IOCompletion`, SCSI completion routines are usually called at interrupt time. The difference is that VM does not intercept them. This means that SCSI completion routines are called even if it is not safe for paging, so they are not allowed to cause page faults. For SCSI drivers, that is not usually a problem—their whole world is usually held anyway. In response to a SCSI completion, a driver typically calls `IODone`, which makes a call to the client's `IOCompletion` routine, which could cause a page fault. This is not a problem, because, as mentioned already, VM defers the call until it is safe for paging.

Guidelines for SIM/HBA Developers

Developers of SCSI HBAs should ship their products with SIMs that are compatible with SCSI Manager 4.3, to enable other vendors' drivers and devices to work with their SCSI adapter.

SIM Initialization and General Operation

For the Macintosh Quadra 840AV and other machines with ROMs that contain SCSI Manager 4.3, third-party SIMs can register their HBAs as soon as their code is executing. This happens during NuBus configuration ROM setup (`PrimaryInit`).

For systems without SCSI Manager 4.3 in ROM, SIMs must wait until SCSI Manager 4.3 is up and running before registering with the XPT. This means that a drive on a third-party HBA cannot be used as a boot device nor as the backing store for VM, but can be mounted once SCSI Manager 4.3 is running and the HBA is installed. This secondary driver loading and drive mounting happens after SCSI Manager 4.3 is operational.

To initialize itself, the SIM issues a `SCSIRegisterBus` call, with a pointer to a `SIMinitInfo` structure that has been filled out with the entry points, required static data space size and `oldCallCapable` status of the SIM. The `SIMinitInfo` structure is shown in Listing 9-2. This structure can be disposed after the routine finishes, because the XPT makes a copy of the data.

Listing 9-2   SIM initialization information structure

```
typedef struct {              // used for SCSIRegisterBus call
   uchar    *SIMstaticPtr;    // <- ptr to the SIM's static vars
   long     staticSize;       // -> bytes SIM needs for static
                              //    variables
   long     (*SIMinit)();     // -> pointer to SIM init routine
   long     (*SIMaction)();   // -> pointer to SIM action routine
   long     (*SIM_ISR)();     // -> pointer to the SIM ISR routine
   void     (*NewOldCall)();  // -> pointer to the SIM NewOldCall
   long     intrptSource;     // -> interrupt source specifier
   Boolean  oldCallCapable;   // -> true if this SIM can handle
                              //    old SCSI Manager calls
   ushort   busID;            // <- bus # for the registered bus
   void     (*XPT_ISR)();     // <- pointer to the XPT ISR
   void     (*MakeCallback)();// <- pointer to the XPT layer's
                              //    MakeCallback routine
} SIMinitInfo;
```

The XPT allocates the requested number of bytes for the SIM static space and assigns the next bus number to this SIM. A pointer to the allocated memory is returned as is the `busID` that was assigned, in the appropriate fields of the `SIMinitInfo` parameter block. The XPT also fills in two fields indicating entry points into the XPT. The `XPT_ISR` entry point should be used by the SIM when the XPT has not provided sufficient interrupt registration functions. The `MakeCallback` routine should be called when the SIM completes a `SCSI_ExecIO` request and needs to make the SCSI completion callback to the client.

The `SIMinit` routine is called during the execution of the `SCSIRegisterBus` routine. It is passed the `SIMstaticPtr` (address of SIM's static data space) as well as a pointer to the `SIMinitInfo` structure after the XPT has filled in all of the required fields. `SIMinit` attempts to initialize all data structures and hardware and returns an appropriate result code indicating whether this was successful. If a failure result is returned, the XPT does not register the SIM.

Once the registration is complete, the XPT makes calls to the `SIMaction` entry point whenever a `SCSIAction` call is received that is destined for this bus. The XPT passes a pointer to the `SCSI_PB` parameter block and a pointer to the SIM's static space into the `scSIMPrivate` parameter of the `SIMaction` routine. The SIM should parse the `SCSI_PB` parameter block for illegal or unsupported parameters and return an appropriate failure code if either of these are found. All `SCSI_ExecIO` requests should be treated asynchronously by the SIM; all other types of requests should be treated synchronously.

Once the `ExecIO` request has been queued, the SIM should return back to the XPT. When the request finishes, the SIM calls the XPT's `MakeCallback` routine. This makes the call to the client's specified completion routine (if it exists).

The parameter blocks appear to the client to be queued on a per-LUN basis as queue freezing and unfreezing is performed one LUN at a time. In actuality, the SIM may queue all `SCSI_ExecIO_PB` parameter blocks in the same queue no matter which LUN is the destination. This helps maintain a first-in, first-out sequence of the parameter blocks.

Support for the Old SCSI Manager

Upon registration, every SIM should specify whether or not it supports old SCSI Manager routines. If it indicates that it does, the XPT adds it to the list of buses searched when a `SCSISelect` call is received.

The handling of an old API transaction differs from the handling of new API `SCSI_ExecIO_PB` parameter blocks. The main difference is the presence of communication between the XPT and the SIM during the transaction. This communication consists of the old API calls (to the SIM) and the results returned (from the SIM) at the completion of the routines. The XPT is responsible for catching and converting the old calls into the proper format and submitting them to the SIM. It also takes the results for each of the calls from the SIM and returns back to the client with those results.

`SCSIGet` calls are handled entirely within the XPT; the XPT simply notes that the call was made by setting an internal flag and returning back to the caller. `SCSISelect` calls cause the XPT to generate a `SCSI_ExecIO` parameter block and submit it to the SIM via the `SIMaction` entry point. This parameter block is filled in with an `scFunctionCode` field of `SCSI_OldCall` and an `scDeviceIdent` field containing the bus number of this SIM, the target ID requested in the `SCSISelect` call, and a LUN of 0. This parameter block should be queued with all other `SCSI_ExecIO_PB`s.

The SIM should attempt a select of the specified device and return the result of that select back to the XPT (`scsiReqComplete` if successful and `scsiSelTimeout` if not). Old call results are not communicated through the `scResult` field, as this would be interpreted as completion of the entire transaction rather than only the portion of the transaction resulting from the single old call. Instead, the SIM should place the result in the `oldCallResult` field. As additional old calls are made, the XPT fills in the appropriate fields of the `SCSI_ExecIO_PB` and calls the SIM's `NewOldCall` entry point. Table 9-2 shows the old call parameters and the fields that are filled in by the XPT.

Table 9-2   Old call parameter conversion

Call	Parameter	Dir	ExecIO field	Notes
SCSIGet	—			XPT only
SCSISelect/ SCSISelAtn	targetID	→	scDeviceIdent	`busID` = this SIM, LUN = 0
SCSICmd	*buffer	→	scCDB	Pointer in field
	count	→	scCDBLen	
SCSI<data>	*tibPtr	→	scDataPtr	Pointer in field
SCSIComplete	*stat	←	scSCSIstatus	Status in field
	*message	←	scSCSImessage	Message in field
	wait	→	scConnTimer	`TimeMgr` format
SCSIMsgIn	*message	←	scSCSImessage	Message in field
SCSIMsgOut	message	→	scSCSImessage	Message in field
SCSIReset	—			SCSI_ResetBus_PB
SCSIStat	—			XPT only

To provide the highest level of compatibility with the old SCSI Manager, every SIM should be able to perform a SCSI arbitration and selection process independently of a SCSI message-out or command phase, in order to register itself as being capable of handling old SCSI calls. If it must have the CDB or message-out bytes in order to perform the selection operation, then it will be unable to adequately execute the `SCSISelect` call. Without this ability, the SIM must always return `noErr` to a

SCSISelect (SCSI_OldCall function), a result that produces a false indication of the presence of a device at that ID. This would cause all future SCSISelects to that ID to be directed to only this bus. The result would be that no devices installed on buses that registered after this bus would be accessible through the old API.

Interrupt Support

Each SIM passes the address of its interrupt service routine and an interrupt source identifier (ISR) to the XPT during the SCSIRegisterBus routine. The XPT installs an ISR at the specified source so that when that interrupt happens, it can make the call to the SIM_ISR routine, passing the address of the SIM's static data space. The XPT performs some VM-required operations before and after the call to the SIM_ISR when VM is turned on.

The same SIM_ISR entry point is used by the XPT to get the SIM to check for the presence of an interrupt. Checking for an interrupt is required during various situations where interrupts are disabled but SCSI operations may still be in operation. Hence the SIM_ISR must be written to verify that the interrupt is in fact present before attempting to handle it. If an interrupt is handled during the routine, the SIM should return a nonzero result to the XPT.

Handshaking of Data Bytes

The old SCSI Manager provided TIBs to perform two functions: designation of data buffers (scatter/gather) and designation of handshaking requirements for a transfer. The latter function refers to the handshaking between the processor and the SCSI controller chip. This was originally required during Macintosh Plus blind transfers because there was no hardware handshaking that prevented the processor from overflowing or underflowing the 5380 chip.

In Apple platforms after the Macintosh Plus, the handshaking information was used to prevent bus errors when the target failed to deliver the next byte within a processor bus error timeout or when the SCSI Manager attempted to read it from the SCSI interface chip. This timeout is 250 ms for the Macintosh SE and 16 µs for the Macintosh II and all Macintosh models since. The SCSI Manager blindly read (or wrote) data bytes until it reached the end of an scInc or scNoInc pseudoinstruction. When the next scInc or scNoInc was encountered, the SCSI Manager first explicitly polled the SCSI chip to make sure that it was ready with data (for a read) or ready to accept data (for a write). In this way, TIBs were used to make the SCSI Manager synchronize with the target at times in the transfer when the target was slow in accepting bytes.

The new SCSI Manager still requires this handshaking information for non-DMA SCSI transfers such as those used on all earlier models. There is no possibility of bus errors with the Macintosh Quadra 840AV or Macintosh Centris 660AV, because the DMA hardware does not attempt to transfer data until the SCSI controller indicates that it is ready.

Handshaking is handled similarly for third-party HBAs. With DMA there is no need for the explicit handshaking. With non-DMA transfers, however, a SIM must pay attention to the handshaking description that is part of the `SCSI_ExecIO_PB`. The form of the descriptor is much simpler than TIBs and explicitly specifies which bytes in which to expect delays from the target. In an environment where bus errors may occur if the handshaking description is inaccurate, the SIM should provide a bus error handler that can recover, retry, and pick up the transfer where it was interrupted. Because bus-error exception processing differs among the members of the 68000 processor family, several handlers are required, some of which are not trivial. In addition, it is impossible to predict what will happen in later 68000 processors with different exception handling that might force rewriting and redistribution of any SIMs with bus error handlers.

DMA Support

For HBAs with DMA support, the direct memory access process typically requires that the data buffer affected by the transfer be locked down (so that the physical addresses won't change) and that it be noncacheable. Locking data buffers was previously difficult to manage because of severe restrictions on when `LockMemory` could be called.

`LockMemory` is now allowed at interrupt time but only if the affected pages are already held. `GetPhysical` is also allowed at interrupt time and continues to have its previously restriction of only working with pages that are locked.

SCSI Manager 4.3 Reference

Many SCSI bus–related functions are available to the client. All of them are accessed by calling a single entry point (`SCSIAction`) with a SCSI parameter block (`SCSI_PB`) and are designated by the function code element of the `SCSI_PB` header. The structure of the `SCSI_PB` body (past the header) varies depending upon the function requested.

The parameter block consists of function types, parameter structures, action flags and status flags necessary to perform most SCSI requests. SCSI I/O requests are performed by allocating a SCSI parameter block and filling in the necessary fields to describe and specify the necessary actions the SCSI Manager needs to perform the requested function. The status of both the I/O request and actual SCSI bus transaction are returned through the parameter block. These functions may be specified to complete either synchronously or asynchronously with respect to the calling client.

By far the most important and commonly used request passed to `SCSIAction` is to execute a SCSI I/O request. It is this request that actually performs the SCSI transaction between the computer and the target. All of the parameters required by the SCSI Manager to accomplish a complete transaction are contained in the `SCSI_ExecIO_PB` parameter block that is passed to `SCSIAction`.

Besides routines driven by `SCSI_PB`, the XPT provides several others as well. These routines fall into two categories: routines of interest to a driver-type client and routines of interest to an operating system module (such as a SIM).

Note that in the remainder of this chapter, certain data types have the following
definitions:

```
#define ushort      unsigned short
#define uchar       unsigned char
#define ulong       unsigned long

typedef struct DeviceIdent
{
    uchar       diReserved;         // unused
    uchar       bus;                // SCSI - Bus #
    uchar       targetID;           // SCSI - Target SCSI ID
    uchar       LUN;                // SCSI - LUN
} DeviceIdent;
```

Data Structure

This section describes the general parameter block data structure that provides
information and control in SCSI Manager 4.3. There are many different parameter blocks
all using the same template, SCSI_PB. Specific parameter blocks are discussed with the
routines that use them. This section describes the parameter block header and the
construction of the SCSI_PB parameter block.

SCSI Manager Parameter Block

Each client of the SCSI Manager allocates a SCSI_PB parameter block and fills in the
required fields before passing it to the SCSIAction function. A function-specific
SCSI_PB consists of two parts: the SCSI_PB header (SCSIHdr), that part common to all
types of SCSI_PBs, and the SCSI_PB body, containing SCSI parameters specific to the
function's SCSI_PB (the size and fields of which vary depending on the function).

The common parameter block header definition is the following:

```
#define SCSIPBHdr \
    struct SCSIHdr *qLink;      // (internal) Q link to next PB
    short           qType;      // (unused) Q type
    ushort          scVer;      // -> version of the PB
    ushort          scPBLen;    // -> length of the entire PB
    FunctionType    scFunctionCode;// -> function selector
    OSErr           scResult;   // <- returned result
    DeviceIdent     scDeviceIdent; // -> (bus + target + LUN)
    CallbackProc    scCompFn;   // -> callback on completion function
    ulong           scFlags;    // -> flags for operation
// end of SCSIPBHdr
```

Note

Several fields in the parameter block are operating system dependent. In this document the direction shown by arrows is with respect to the SCSI Manager—for example, in SCSIPBHdr. This is opposite to the convention followed by ANSI X3T9, the Common Access Method document, as explained in "CAM Deviations," earlier in this chapter. ◆

The SCSI parameter block header structure uses SCSIPBHdr, as follows:

```
typedef struct SCSIHdr
{
    SCSIPBHdr
} SCSIHdr;
```

*qLink   Reserved for Apple use only. A pointer to the next parameter block in the SCSI queue.

qType   Reserved for Apple use only. The queue type.

scVer   Version of the parameter block. Used by SCSI Manager to determine the format of this parameter block.

scPBlen   The length in bytes of the PB, including the PB header.

scFunctionCode
               A function selector that specifies the service being requested by the SCSI device driver. See also "SCSIAction," later in this chapter.

scDeviceIdent
               A function selector that specifies the device that the request is directed towards. This field is of type DeviceIdent, defined above.

scResult   A value returned by the SCSI Manager after the function is completed. A scsiReqInProg status indicates that the request is still in progress or queued.

               Valid scResult return values are:

               noErr   Request completed without error
               scsiReqInProg   Request in progress
               scsiReqAborted   Request aborted by the host
               scsiUnableToAbort   Unable to abort request
               scsiReqCmplWErr   Request completed with an error
               scsiBusy   SCSI subsystem busy
               scsiReqInvalid   Request invalid
               scsiBusInvalid   Bus ID supplied invalid
               scsiDevNotThere   SCSI device not installed/there
               scsiUnableTermIO   Unable to terminate I/O request
               scsiSelTimeout   Target selection timeout
               scsiCmdTimeout   Command timeout
               scsiMsgRejectRcvd   Message reject received

`scsiSCSIBusReset`	SCSI bus reset sent/received
`scsiUncorParity`	Uncorrectable parity error occurred
`scsiAutosenseFail`	Autosense: request sense command fail
`scsiNoHBA`	No HBA detected
`scsiDataRunErr`	Data overrun/underrun
`scsiUnexpBusFree`	Unexpected bus free
`scsiSequenceFail`	Target bus phase sequence failure
`scsiPBLenErr`	Parameter block length supplied is inadequate
`scsiProvideFail`	Unable to provide requested capability
`scsiBDRsent`	A SCSI BDR bus request message was sent to the target
`scsiReqTermIO`	Request terminated by the host
`scsiLUNInvalid`	LUN supplied is invalid
`scsiTIDInvalid`	Target ID supplied is invalid
`scsiFuncNotAvail`	The requested function is not available
`scsiNoNexus`	Nexus not established
`scsiIIDInvalid`	Initiator ID invalid
`scsiCDBRcvd`	The SCSI CDB has been received
`scsiSCSIBusy`	SCSI bus busy
`scsiSIMQFrozen`	The SIM queue frozen with this error
`scsiAutosenseValid`	Autosense data valid for target

`scDeviceIdent`

A longword that uniquely identifies a device that this request is directed toward. The `DeviceIdent` designates a bus ID, target SCSI ID, and LUN. A routine is provided to decode a `DeviceIdent` value into these components if required, but the objective is to eliminate the physical addressing characteristics of the transport layer (SCSI bus) from the API.

`scCompFn`   A pointer to the callback completion function.

`scFlags`   A longword that contains the bit settings to indicate special handling of the requested function. The number and meaning of the flags vary by function code and are described in function-specific areas:

Flag descriptions

`scsiDirMask`

Bit field used to specify direction of transfer. Values can be

`scsiDirIn`	Data direction in
`scsiDirOut`	Data direction out
`scsiDirNone`	No data movement

`scsiDisAutosense`
> Disable autosense feature

`scsiScatterValid`
> Scatter/gather list is valid. If this flag is clear, the values in the `scData` and `scDataLen` fields are the starting address and length of a block of data. If this flag is set, the `scData` field is a pointer to an S/G list. Each element of the S/G list is itself a description of a block of data. In addition, when set, the `scSGlistCnt` field contains the number of S/G entries, and the `scDataLen` field contains the total number of bytes in the data transfer. This last field is required for easy calculation of the `scDataResidLen` value.

`scsiCDBLinked`
> The PB contains a linked CDB. This bit/function is not supported in the built-in SIM.

`scsiQEnable`
> SIM queue actions are enabled. This bit/function is not supported in the built-in SIM.

`scsiCDBIsPointer`
> The CDB field contains a pointer. If clear, the `scCDB` field contains the actual CDB. If set, the `scCDB` field contains a pointer to the CDB. In either case, the `scCDBLen` field contains the number of bytes in the command.

`scsiDisDisconnect`
> Disable disconnect. This flag, when set, prevents the SIM from setting the `DiscPriv` bit in the identify message used for this I/O. If clear (default), `DiscPriv` is set, allowing the target to disconnect.

`scsiInitiateSync`
> Attempt sync data xfer, and SDTR

`scsiDisSync`
> Disable sync; go to async

`scsiSIMQHead`
> Place parameter block at the head of SIM queue

`scsiSIMQFreeze`
> Return the SIM queue to frozen state

`scsiSIMQNoFreeze`
> Disallow SIM queue freezing

`scsiCDBPhys`
> CDB pointer is physical

`scsiDataPhys`
> SG/buffer data pointers are physical

`scsiSenseBufPhys`
> Autosense data pointer is physical

`scsiMsgBufPhys`
> Message buffer pointer is physical

`scsiNxtPBPhys`
> Next parameter block pointer is physical

`scsiCallBackPhys`
> Callback function pointer is physical

`scsiPhysMask`
> At least one pointer is physical

`scsiDataBufValid`
> Data buffer valid

`scsiStatusBufValid`
> Status buffer valid

`scsiMsgBufValid`
> Message buffer valid

`scsiTgtPhaseMode`
> The SIM will run in phase mode

`scsiTgtPBAvail`
> Target parameter block available

`scsiDisAutoDisc`
> Disable autodisconnect

`scsiDisAutsaveRest`
> Disable autosave/restore pointers

Routines

This section describes the routines used to control and inquire from the different layers of the SCSI Manager hierarchy, as shown in Figure 8-1 (page 366). The order of discussion is:

1. Driver routines

2. SCSI Interface Modules calls to the transport layer

3. Transport layer calls to SCSI Interface Modules

Driver Routines

Driver routines are used by the client to control and inquire from the transport layer. For most operations using the SCSI Manager, these are the only routines that are needed.

SCSIAction

The `SCSIAction` routine executes the request specified in the `SCSI_PB` parameter block. Certain types of requests are handled by the XPT (such as those dealing with the SCSI device table), but most are handled by the SIM/HBA. The `SCSI_PB` header contains a function code specifying the requested operation. The codes are described later in this section, along with the parameter blocks that correspond to those functions.

```
void   SCSIAction (SCSI_PB *)
```

Operation

Drivers make all of their SCSI I/O requests using this function. It is designed to take advantage of all features of SCSI that could be provided by virtually any HBA/SIM combination. The parameter `SCSI_PB` block contains all of the parameters that the XPT and SIM need to completely transact the I/O request.

The `SCSIAction` function typically returns with a status of 0 indicating that the request was queued successfully. Function completion can be determined by polling for nonzero status or through the use of the callback on completion field. When the completion routine is called, it has the same static variable pointer (A5) that existed when the Execute SCSI I/O request was received. If A5 was invalid when the I/O request was made, it is also invalid when in the callback.

The callback routine should follow this format:

```
void   CompFn (SCSI_ExecIO_PB * thePB);
```

When issued asynchronously, execute SCSI I/O requests are performed as such; in other words, the resulting action may start anytime and may end at any time. There is no implied ordering of these events with respect to earlier or later requests. An earlier request may be started later and a later request may complete earlier. However, a series of requests to the same device (bus ID + target ID + LUN) is issued to that device in the order received.

SCSIAction Function Codes

`SCSIAction` function codes are used by SCSI Manager clients to specify requests. Table 9-3 lists the hexadecimal function codes that SCSI Manager 4.3 supports on its initial release.

In Table 9-3, note that codes $00 through $0F cover common functions; codes $10 through $1F cover SCSI control functions; and codes above $7F are reserved by Apple.

Table 9-3 SCSI Manager 4.3 function codes

Code	Function	Operation (CAM names)	Supported
$00	SCSI_Nop	NOP (No Operation)	√
$01	SCSI_ExecIO	Execute SCSI I/O	√
$02	(reserved)	Get Device Type	
$03	SCSI_BusInquiry	Path (Bus) Inquiry	√
$04	SCSI_ReleaseQ	Release SIM Queue	√
$05–$0F	(reserved)	Set Async callback	√
$10	SCSI_AbortCommand	Abort SCSI command	√
$11	SCSI_ResetBus	Reset SCSI bus	√*
$12	SCSI_ResetDevice	Reset SCSI device	√
$13	SCSI_TerminateIO	Terminate I/O process	√
$14–$7F	(reserved)		
$80	SCSI_GetVirtualIDInfo	Get DeviceID of virtual ID	√

* Not recommended; see warning on page 392.

SCSI_ExecIO

The most commonly executed request of the SCSI Manager is to perform an I/O command, as defined by the SCSI_PB parameter block with a selector code of SCSI_ExecIO. The resulting data structure is the following:

```
typedef struct SCSI_ExecIO_PB
{
   SCSIPBHdr                        // header information fields
   uchar          *scDrvrStorage;// <> ptr used by the driver
   struct SCSI_IO *scCmdLink;     // -> ptr to the next linked cmd
   ulong          scAppleRsvd0;   //    reserved
   uchar          *scDataPtr;     // -> ptr to data buffer
                                  //    or S/G list
   ulong          scDataLen;      // -> data transfer length
   uchar          *scSenseBufPtr;// -> ptr to autosense buffer
   uchar          scSenseBufLen; // -> size of autosense buffer
   uchar          scCDBLen;       // -> number of bytes for the CDB
   ushort         scSGlistCnt;   // -> number of S/G list entries
   ulong          scAppleRsvd1;   //    reserved
   uchar          scSCSIstatus;  // <- returned SCSI device status
   char           scSenseResidLen;// <-autosense residual length
```

```
    ushort          scAppleRsvd2;  //    reserved
    long            scDataResidLen;// <- transfer residual length
    CDB             scCDB;         // -> actual CDB or ptr to CDB
    long            scTimeout;     // -> timeout value (Time
                                   //    Manager format)
    uchar           *scMsgPtr;     // -> pointer to message buffer
    ushort          scMsgLen;      // -> num bytes in msg buffer
    ushort          scVUFlags;     // -> vendor (Apple) unique flags
    uchar           scTagAction;   // -> what to do for tag queuing
    uchar           scAppleRsvd3;  //    reserved
    ushort          scAppleRsvd4;  //    reserved
    // Apple-specific public fields
    uchar           *scSGBase;     // -> base data for S/G entries
    ushort          scSelTimeout;  // -> select timeout value
    ushort          scXferType;    // -> transfer type
    DataXferProc    scDIxfer;      // -> data in function
    DataXferProc    scDOxfer;      // -> data out function
    ushort          scHandshake[8];// -> handshaking structure
    ulong           scAppleRsvd5;  //    reserved
    long            scConnTimeout; // -> connection timeout value
    uchar           scSIMpublics[8];//   for use by 3rd-party SIMs
    uchar           publicExtras[4];//   for a total of 48 bytes
    // XPT layer privates (for old API emulation)
    Ptr        savedA5;        // the A5 of the client
    ushort     scCurrentPhase;// <- phase upon completing old call
    short      selector;      // -> selector specified in old call
    ushort     oldCallStatus; // I/O status of old call
    uchar      scSCSImessage; // <- Returned SCSI device message
    uchar      XPTprivFlags;  // <> various flags
    uchar      XPTextras[4];  // for a total of 16 bytes
} SCSI_ExecIO_PB;
```

Field descriptions

SCSIPBHdr   Shorthand for the SCSI Manager parameter block structure. See
                 "SCSI Manager Parameter Block," earlier in this chapter, for details.

*scDrvrStorage   A pointer used by the peripheral driver to access the SCSIHdr.

*scCmdLink   A pointer to the next linked command.

scAppleRsvd0   Reserved.

*scDataPtr   A pointer to the data buffer or the S/G list.

scDataLen   Length of data buffer to be transferred.

*scSenseBufPtr   A pointer to the autosense data buffer. Used to get information
                 about the autosense status.

scSenseBufLen   Size of the autosense data buffer.

scCDBLen	Length of the CDB in bytes.
scSGlistCnt	Reserved. Number of entries in the S/G list. Used only by the operating system.
scAppleRsvd1	Reserved.
scSCSIstatus	A byte that returns the SCSI device status. Contains the status of the specified SCSI device.
scSenseResidLen	Autosense residual length.
scAppleRsvd2	Reserved.
scDataResidLen	Data transfer residual length.
scCDB	Actual or a pointer to the CDB.
scTimeout	Length of time specified before timeout of the SCSI bus.
*scMsgPtr	A pointer to the message buffer.
scMsgLen	Number of bytes in the message buffer.
scVUFlags	Apple-specific flags. These flags define the Apple-specific operations supported by SCSI Manager 4.3.

Flag Descriptions

scsiNoParityCk

Disables the checking of parity on incoming data. Parity continues to be generated for outgoing data.

scsiDisSelAtn

Disables the sending of the Identify message for LUN selection. The DeviceIdent still specifies the LUN so that the request gets placed in the proper queue. As always, the LUN field in the CDB is untouched. The purpose is to provide compatibility with pre-SCSI-2 devices that did not support the inquiry+LUN concept as described in the SCSI-2 documentation.

scsiSavePtrOnDisc

If this flag is set, the SCSI Manager automatically does a Save Data Pointer operation when it receives a Disconnect message from the target. If this flag were clear, operation would be as specified in SCSI-2; in particular, there is no implied Save Data Pointer when a Disconnect message is received, and if a disconnect actually did occur, the data pointer would revert to the value last saved. The purpose of this bit is to provide compatibility with devices whose designers did not understand the function of the Save Data Pointer and Disconnect messages.

scsiNoBucketIn

When set, no bit-bucketing on data-in is performed for this transaction. Bit-bucketing normally occurs when the device (target) wants to supply more data than the computer (initiator) is expecting. This can happen if the `SCSI_Exec_IO` parameter block has inconsistent parameters—with the CDB indicating a request for more data than the S/G list provides. If this bit is set and the extra data condition occurs, the SCSI Manager request

terminates and the bus is left in `data_in` phase. A `SCSI_ResetBus` request must be issued to clear the bus. Due to the impact of a SCSI Reset, this bit should only be set for debugging.

scsiNoBucketOut

When set, no bit-bucketing on data-out is performed for this transaction. This is the inverse of bit-bucketing described above and normally occurs when the target is asking for more data than was supplied in the I/O request. Again, this bit should only be used for debugging purposes.

scsiExecSync

This flag causes I/O to be executed synchronously (it returns from a `SCSIAction` call only when complete).

scTagAction	Specifies what action is taken for tag queuing.
scAppleRsvd3	Reserved. SCSI Manager private data area.
scAppleRsvd4	Reserved. SCSI Manager private data area.

Apple-specific fields

*scSGBLase	A pointer to the base data in an S/G entry.
scSelTimeout	A field that allows the client to set an alternate select timeout value. The timeout is specified in milliseconds but there is no guaranteed accuracy because different HBAs have different capabilities, including only being able to handle the standard 250 ms. A value of 0 designates this default time length.
scXferType	An option that selects which type of transfer to use during the data phase. This roughly corresponds to blind versus polled. This option is provided for backward compatibility with a few devices. For nearly every device, this field should be zero, which selects the default, fastest, most reliable transfer routine for the selected bus. The number of specialized transfer types available on a particular HBA is available in the `scXferTypes` field of the `BusInquiry` parameter block.
*scDIxfer	A pointer to a client-supplied function used by the SCSI Manager during the data in phase. If null, the SIM's routine is used.
*scDOxfer	A pointer to a client-supplied function used by the SCSI Manager during the data-out phase. If null, the SIM's routine is used.
scHandshake[8]	A structure used for handshake operations.
scAppleRsvd5	Reserved for Apple use only.

`scConnTimeout`   A value used to time out SCSI operations.

`scSIMpublics[8]`

Basic allocation for use by third-party SIM vendors.

`publicExtras[4]`

Expanded allocation for third-party SIM vendors, providing a total of 48 bytes.

SCSI_AbortCommand

The `SCSI_AbortCommand` function asks that a SCSI Manager request be canceled by identifying the parameter block associated with the request. It should be issued on any I/O request (not completed) that the driver wishes to cancel. Success of the Cancel function is never assured. This request does not necessarily result in an Abort message being issued over SCSI.

```
// Abort SCSI Manager Request parameter block
typedef struct SCSI_AbortCommand_PB
{
    SCSIPBHdr                   // header information fields
    SCSIHdr     *scThePB;   // -> pointer to the PB to abort
} SCSI_AbortCommand_PB;
```

`SCSIPBHdr`   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

`*scThePB`   A pointer to the parameter block to be canceled.

SCSI_ResetBus

This `SCSI_ResetBus` function is used to reset the specified SCSI bus.

```
typedef struct SCSI_ResetBus_PB
{
    SCSIPBHdr                   // header information fields
} SCSI_ResetBus_PB;
```

`SCSIPBHdr`   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

▲   **W A R N I N G**
This function should not be used in normal operation. It can be used only in the unlikely event that a client is unable to use the SIM/HBA due to a faulty device disabling the bus. ▲

SCSI_ResetDevice

The `SCSI_ResetDevice` function is used to reset the specified SCSI target. This function should not be used in normal operation, but if I/O to a particular device hangs up for some reason, drivers can abort the I/O and reset the device before trying again. This request shall always result in a Bus Device Reset message being issued over SCSI.

```
typedef struct SCSI_ResetDevice_PB
{
    SCSIPBHdr              // header information fields
} SCSI_ResetDevice_PB;
```

SCSIPBHdr   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

SCSI_TerminateIO

The `SCSI_TerminateIO` function requests that a SCSI Manager I/O request be terminated by identifying the parameter block associated with the request. This function should be called for any I/O request that has not completed and that the driver wishes to terminate. Success of the termination process is never assured. This request does not necessarily result in a `TerminateIOProcess` message being issued over the SCSI bus.

```
typedef struct SCSI_TerminateIO_PB
{
    SCSIPBHdr                     // header information fields
    SCSIHdr     *scThePB;    // -> a pointer to the parameter block
                                  // to terminate
} SCSI_TerminateIO_PB;
```

SCSIPBHdr   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

*scThePB   A pointer to the parameter block to be canceled.

SCSI_GetVirtualIDInfo (Apple-specific)

The `SCSI_GetVirtualIDInfo` routine returns the device ID for the specified virtual ID. This function is typically used by a peripheral driver during the transition from ROM-based previous SCSI Manager to a system file-based SCSI Manager 4.3. If no device has yet been found on any of the `oldCallCapable` buses, the `scExists` Boolean value is `FALSE` and the `DeviceIdent` field should be ignored.

```
typedef struct SCSI_GetVirtualInfo_PB
{
    SCSIPBHdr                      // header information fields
    ushort      scVirtualID;   // -> SCSI ID of device
                               //  in question
    Boolean     scExists;      // <- true if device exists
} SCSI_GetVirtualInfo_PB;
```

scHdr   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

scVirtualID
              Identification of a device on either internal or external bus.

scExists   A Boolean value that returns `true` if the device exists on the bus.

Note
The `DeviceIdent` value is returned in the header of this parameter block which makes this the only function that returns a value in the `SCSIHdr` outside of the `scStatus` field. ◆

SCSI_ReleaseQ

The `SCSI_ReleaseQ` function releases a frozen SIM queue for the selected LUN.

```
typedef struct SCSI_ReleaseQ_PB
{
    SCSIPBHdr                  // header information fields
} SCSI_ReleaseQ_PB;
```

SCSIPBHdr   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

SCSI_BusInquiry

The SCSI_BusInquiry function is used to get information on the specified HBA, including the number of HBAs installed.

```
typedef struct SCSI_BusInquiry_PB
{
    SCSIPBHdr                       // header information fields
    uchar    scVersionNum;       // <- version number for controller
    uchar    scHBAInquiry;       // <- mimic of INQ byte 7
    uchar    scTargetMdFlags;  // <- flags for target mode support
    uchar    scSIMMisc;           // <- misc feature flags
    ushort   scEngineCnt;        // <- number of engines on bus
    // Apple-specific fields through scVUrsrvd (14 bytes total)
    ushort   scXferTypes;        // <- number of transfer types
                                    //    for this HBA
    ushort   scCntrlrType;       // <- type of SCSI controller used
    ulong    scVUflags;          // <- various Apple-specific flags
    uchar    scVUrsrvd[14-VU_used];// <- vendor-unique reserved
                                         //    leftovers
    ulong    scSIMPrivSize;     // <- size of SIM private data area
    ulong    scAsyncFlags;       // <- event cap. for Async callback
    uchar    scHiBusID;           // <- highest bus ID in subsystem
    uchar    scInitiatorID;      // <- initiator ID on SCSI bus
    ushort   scReserved;          //    reserved
    char     scSIMVend[16];      // <- vendor ID of the SIM
    char     scHBAVend[16];      // <- vendor ID of HBA
    ulong    scOSDreserved;      //    reserved [OSD]
    char     scCntrlFamily[16];// <- family of SCSI controller
    char     scCntrlType[16];   // <- family of SCSI controller
} SCSI_BusInquiry_PB;
```

Standard field descriptions

SCSIPBHdr   Shorthand for the SCSI Manager parameter block structure. See "SCSI Manager Parameter Block," earlier in this chapter, for details.

scVersionNum   The version number field is used by the client to verify that the SIM can handle the requests the client was designed to issue:

Value	Meaning
$00–07	Prior to revision 1.7
$08	Implementation version 1.7
$09–FF	Revision number; for example $31 = 3.1

scHBAInquiry	These flags indicate basic SCSI capabilities of the subsystem (SIM + HBA).

Bit	Meaning
7	Modify data pointers
6	Wide bus 32
5	Wide bus 16
4	Synchronous transfers
3	Linked commands
2	(reserved)
1	Tagged queuing
0	Soft reset

scTargetMdFlags	Target mode is not supported in the initial versions of SCSI Manager 4.3 and consequently, this field returns 0.

Bit	Meaning
7	Processor mode
6	Phase cognizant mode
5-0	(reserved)

scSIMMisc	These flags are meant to designate how the SCSI Device Table is generated and maintained.

Bit	Meaning
7	0 = scanned low to high 1 = scanned high to low
6	0 = removables included in table 1 = removables not included in table
5	1 = inquiry data not kept by XPT
4-0	(reserved)

scEngineCnt	As engines are not supported, this value is always 0 for Apple-supplied SIMs and HBAs.

Apple-specific field descriptions

scXferTypes	A field that returns the number of data transfer types available on this HBA. These transfer types are roughly analogous to blind, polled, and so on. They are provided purely for the sake of compatibility with unusual devices that have specific timing requirements. Apple SIMs provide two transfer routines that resemble blind (1) and polled (2) modes. Here this field is 2. The driver specifies which transfer type to use during a particular I/O in the scXferType field in the SCSI_ExecIO_PB parameter block. The scXferTypes value returned from a bus inquiry is the maximum value supported in the Exec SCSI I/O request.
scCntrlrType	A field that designates the SCSI controller chip used in this HBA.

scVUflags	Following are the currently defined Apple-specific flags for HBAs:

Bit	Meaning
0	DMA transfer available and supported
1	Fast synchronous capable
2	Single-ended (0) or differential (1)
3	Bus has no external connectors (i.e. cable cannot extend outside case)
4	HBA is capable of supporting old-API calls from XPT

scHBAname[16]   An HBA product name— an ASCII text HBA identifier. It is meant to correspond to a commonly known product name for the HBA such as WhopperSCSI SE30.

scVUrsrvd[14-VUused]
As specified by CAM, a field for vendor-unique data that contains 14 bytes less the part used by Apple.

scSIMPrivSize   As specified by CAM, this field designates how many bytes of data are in the SIM's private data area (static).

scAsyncFlags   Flags that indicate which types of asynchronous events are generated by this SIM. A client may register with the XPT to receive a callback when any of these events occur.

scHiBusID   If no bus IDs exist, i.e. no SCSI buses are registered, then the highest bus ID assigned is $FF, the ID of the XPT.

scInitiatorID   SCSI Device ID (of Initiator)—For all Apple-supplied HBAs, this field is 7. It is highly recommended that all third-party HBAs also use ID 7 for their initiator.

scReserved   Reserved for Apple use.

scSIMVend[16]   Vendor ID of SIM-supplier—This is an ASCII text vendor identifier. Apple Computer is designated "Apple Computer".

scHBAVend[16]   Vendor ID of HBA-supplier This is an ASCII text vendor identifier. Apple Computer is designated "Apple Computer".

scCntrlFamily[16]
A field that designates the family of parts that the SCSI controller chip belongs to. It is meant to describe primarily the programming interface to the part. For instance, 5380, 53c80, and IIfx SCSIDMA chips all have a family of NCR 5380.

scCntrlType[16] Specific type of SCSI controller.

SCSI Interface Module Calls to Transport

The routines described in this section are used by a SIM to communicate with the transport layer. Their calls should all be supported by SIM developers.

SCSIRegisterBus

The `SCSIRegisterBus` routine is called to register an HBA for use with the transport (XPT). Several characteristics of the HBA are specified as well as the software entry point SIM and the number of bytes required for a static data space (for global variables). The XPT returns a `BusID` that is used for that HBA as well as a pointer to the allocated static space.

```
long   SCSIRegisterBus (SIMinitInfo * SIMinfo);
```

`SIMinitInfo` is defined as:

```
typedef struct {                  // used for SCSIRegisterBus call
   uchar   *SIMstaticPtr;    // <- ptr to the SIM's static vars
   long    staticSize;       // -> bytes SIM needs for static
                             //    variables
   long    (*SIMinit)();     // -> pointer to SIM init routine
   long    (*SIMaction)();   // -> pointer to SIM action routine
   long    (*SIM_ISR)();     // -> pointer to the SIM ISR routine
   void    (*NewOldCall)();  // -> pointer to the SIM NewOldCall
   long    intrptSource;     // -> interrupt source specifier
   Boolean oldCallCapable;   // -> true if this SIM can handle
                             //    old SCSI Manager calls
   ushort  busID;            // <- bus # for the registered bus
   void    (*XPT_ISR)();     // <- ptr to the XPT ISR
   void    (*MakeCallback)();// <- pointer to the XPT layer's
                             //    MakeCallback routine
} SIMinitInfo;
```

Field descriptions

`SIMstaticPtr`	A pointer to the allocated space for the SIM's static variables.
`staticSize`	A longword that specifies the number of bytes needed by the SCSI interface module for its static variables.
`*SIMinit`	A pointer to this SIM's initialization routine.
`*SIMaction`	A pointer to this SIM's action routine.
`*SIM_ISR`	A pointer to this SIM's interrupt service/polling routine.
`*NewOldCall`	A pointer to this SIM's routine for accepting old SCSI Manager calls.
`oldCallCapable`	A Boolean value that is `true` if this SIM can handle old SCSI Manager calls.
`intrptSource`	The interrupt source for this SIM's HBA.
`busID`	The bus number of the bus that this SIM is registered to use.
`*XPT_ISR`	A pointer to the XPT's interrupt service routine, used when the SIM has an interrupt source besides the one specified in `SIMinitInfo`.
`SIMstaticPtr`	A pointer to this SIM's static variables.

SCSIDeregisterBus

The `SCSIDeregisterBus` routine is called to deregister an HBA when it is no longer available for use.

```
long    SCSIDeregisterBus (ushort busID);
```

busID   The bus number of the bus that this SIM is registered to use.

Transport Calls to SCSI Interface Modules

These routines are used by the transport to control the SIM. This section includes all the previous SCSI Manager routines that the new SCSI manager supports. Their calls should all be supported by SIM developers.

SIMinit

The `SIMinit` routine is called by the XPT to initialize the SIM's state. The SIM, in turn has the responsibility of optionally initializing the HBA.

```
void  SIMinit (Ptr SIMstaticPtr, long busID);
```

SIMstaticPtr
            A pointer to the previously allocated SIM static data area.
busID   Bus identification for this HBA.

SIMAction

The `SIMAction` routine is called by the XPT whenever a `SCSIAction` call is received that needs to be serviced by the SIM.

```
long  SIMAction (SCSI_PB *thePB, Ptr SIMstaticPtr);
```

*thePB   A pointer to the parameter block.

SIMstaticPtr
            A pointer to the previously allocated SIM static data area.

Summary of the SCSI Manager 4.3

Constants

```
/******************************************************************/
// Defines for the SCSIMgr scResult field in the parameter block header.
/******************************************************************/


#define scsiReqInProg       1                   // PB request is in progress


#define scsiReqAborted      (0xE100+0x02)       // -7934 = PB request aborted by
                                                // the host
#define scsiUnableToAbort   (0xE100+0x03)       // -7933 = Unable to Abort PB
                                                // request
#define scsiReqCmplWErr     (0xE100+0x04)       // -7932 = PB request completed
                                                // with an error
#define scsiBusy            (0xE100+0x05)       // -7931 = SCSI subsystem is busy
#define scsiReqInvalid      (0xE100+0x06)       // -7930 = PB request is invalid
#define scsiBusInvalid      (0xE100+0x07)       // -7929 = bus ID supplied is
                                                // invalid
#define scsiDevNotThere     (0xE100+0x08)       // -7928 = SCSI device not
                                                // installed there
#define scsiUnableTermIO    (0xE100+0x09)       // -7927 = unable to terminate I/O
                                                // PB request
#define scsiSelTimeout      (0xE100+0x0A)       // -7926 = target selection timeout
#define scsiCmdTimeout      (0xE100+0x0B)       // -7925 = command timeout
#define scsiMsgRejectRcvd   (0xE100+0x0D)       // -7923 = message reject received
#define scsiSCSIBusReset    (0xE100+0x0E)       // -7922 = SCSI bus reset sent
                                                // received
#define scsiUncorParity     (0xE100+0x0F)       // -7921 = uncorrectable parity
                                                // error occurred
#define scsiAutosenseFail   (0xE100+0x10)       // -7920 = autosense: Request
                                                // sense cmd fail
#define scsiNoHBA           (0xE100+0x11)       // -7919 = no HBA detected error
#define scsiDataRunErr      (0xE100+0x12)       // -7918 = data overrun/underrun
#define scsiUnexpBusFree    (0xE100+0x13)       // -7917 = unexpected bus free


#define scsiSequenceFail    (0xE100+0x14)       // -7916 = target bus phase
                                                // sequence failure
```

```
#define scsiPBLenErr        (0xE100+0x15)  // -7915 = PB length supplied is
                                           // inadequate
#define scsiProvideFail     (0xE100+0x16)  // -7914 = unable to provide
                                           // required capability
#define scsiBDRsent         (0xE100+0x17)  // -7913 = a SCSI BDR message was
                                           // sent to target
#define scsiReqTermIO       (0xE100+0x18)  // -7912 = PB request terminated
                                           // by the host


#define scsiLUNInvalid      (0xE100+0x38)  // -7880 = LUN supplied is invalid
#define scsiTIDInvalid      (0xE100+0x39)  // -7879 = target ID supplied is
                                           // invalid
#define scsiFuncNotAvail    (0xE100+0x3A)  // -7878 = the required function is
                                           // not available
#define scsiNoNexus         (0xE100+0x3B)  // -7877 = Nexus is not established
#define scsiIIDInvalid      (0xE100+0x3C)  // -7876 = initiator ID is invalid
#define scsiCDBRcvd         (0xE100+0x3E)  // -7874 = SCSI CDB has been
                                           // received
#define scsiSCSIBusy        (0xE100+0x3F)  // -7873 = SCSI bus busy

#define scsiSIMQFrozen      0x40           // SIM queue frozen with this error
#define scsiAutosenseValid  0x80           // autosense data valid for target

#define scsiResultMask      0x00C0         // mask for high (QFZN and
                                           // AUTOSNS_VALID) bits


// ------------------------------------------------------------------


// Defines for the SCSIMgr flags field in the parameter block header.

// 1st Byte
#define scsiDirReserved     0x00000000  // data direction (00: reserved)
#define scsiDirIn           0x40000000  // data direction (01: DATA IN)
#define scsiDirOut          0x80000000  // data direction (10: DATA OUT)
#define scsiDirNone         0xC0000000  // data direction (11: no data)
#define scsiDirMask         0xC0000000  // data direction mask
#define scsiDisAutosense    0x20000000  // disable autosense feature
#define scsiScatterValid    0x10000000  // S/G list is valid
#define scsiCDBLinked       0x04000000  // parameter block contains a
                                        // linked CDB
#define scsiQEnable         0x02000000  // SIM queue actions are enabled
#define scsiCDBIsPointer    0x01000000  // CDB field contains a pointer
```

```
// 2nd Byte
#define scsiDisDisconnect  0x00800000  // disable disconnect
#define scsiInitiateSync   0x00400000  // attempt Sync data xfer, and SDTR
#define scsiDisSync        0x00200000  // disable sync, go to async
#define scsiSIMQHead       0x00100000  // place PB at the head of SIM Q
#define scsiSIMQFreeze     0x00080000  // return the SIM Q to frozen state
#define scsiSIMQNoFreeze   0x00040000  // disallow SIM Q freezing
#define scsiCDBPhys        0x00020000  // CDB pointer is physical


// 3rd Byte
#define scsiDataPhys       0x00002000  // S/G buffer data pointers are
                                       // physical
#define scsiSense BufPhys  0x00001000  // autosense data pointer is physical
#define scsiMsgBufPhys     0x00000800  // message buffer pointer is physical
#define scsiNxtPBPhys      0x00000400  // next parameter block pointer is
                                       // physical
#define scsiCallBackPhys   0x00000200  // callback function pointer is
                                       // physical
#define scsiPhysMask       0x00000100  // at least one pointer is physical


// 4th Byte - Target Mode Flags
#define scsiDataBufValid   0x00000080  // data buffer valid
#define scsiStatusBufValid 0x00000040  // status buffer valid
#define scsiMsgBufValid    0x00000020  // message buffer valid
#define scsiTgtPhaseMode   0x00000008  // SIM will run in phase mode
#define scsiTgtPBAvail     0x00000004  // target PB available
#define scsiDisAutoDisc    0x00000002  // disable autodisconnect
#define scsiDisAutsaveRest 0x00000001  // disable autosave/restore pointers


;// APPLE Unique flags - scVUFlags

#define scsiNoParityCk     0x0002      // disable parity checking
#define scsiDisSelAtn      0x0004      // disable select with attention
#define scsiSavePtrOnDisc  0x0008      // do SAVEDATAPOINTER when DISCONNECT
#define scsiNoBucketIn     0x0010      // don't bit-bucket in during this I/O
#define scsiNoBucketOut    0x0020      // don't bit-bucket out during this
                                       // I/O
#define scsiExecSync       0x0040      // execute this I/O synchronously
```

```
// ----------------------------------------------------------------------
// Defines for the SIM/HBA queue actions. These values are used in the
// SCSI_ExecIO_PB, for the queue action field.

#define scsiSimpleQTag     0x20          // tag for a simple queue
#define scsiHeadQTag       0x21          // tag for head of queue
#define scsiOrderedQTag    0x22          // tag for ordered queue
// Defines for the Bus Inquiry parameter block fields.
#define scsiVERSION        0x22          // binary value for the current vers
#define busMDP             0x80          // supports MDP message
#define busWide32          0x40          // supports 32 bit wide SCSI
#define busWide16          0x20          // supports 16 bit wide SCSI
#define busSDTR            0x10          // supports SDTR message
#define busLinkedCDB       0x08          // supports linked CDBs
#define busTagQ            0x02          // supports tag queue message
#define busSoftReset       0x01          // supports soft reset
#define busTgtProcessor    0x80          // target mode processor mode
#define busTgtPhase        0x40          // target mode phase mode
#define busScansHi2Lo      0x80          // bus scans from ID 7 to ID 0
#define busNoRemovable     0x40          // removable dev not included in scan
#define busDMAavail        0x01          // DMA is available
#define busFastSCSI        0x02          // HAL supports fast SCSI
#define busDifferential    0x04          // singleEnded (0) or Differential (1)
#define busNoExtern        0x08          // HAL has no external connectors
#define busOldAPI          0x10          // HAL is old API capable
```

Data Type

```
typedef struct {  // directions for SCSIRegisterBus: ( -> parm, <- result)
 uchar   *SIMstaticPtr;        // <- ptr to the SIM's static vars
 long    staticSize;          // -> num bytes SIM needs for static vars
 long    (*SIMinit)();        // -> pointer to the SIM init routine
 long    (*SIMaction)();      // -> pointer to the SIM action routine
 long    (*SIM_ISR)();        // -> pointer to the SIM ISR routine
 void    (*NewOldCall)();     // -> pointer to the SIM NewOldCall routine
 Boolean oldCallCapable;      // -> true if this SIM can handle old-API calls
 ushort  busID;               // <- bus number for the registered bus
 void    (*XPT_ISR)();        // <- ptr to the XPT ISR
 void    (*MakeCallback)();   // <- pointer to the XPT layer's
                              //    MakeCallback routine
} SIMinitInfo;
```

Routines

```
void  OSErr    SCSIAction(SCSI_PB *);
long  OSErr    SCSIRegisterBus(SIMinitInfo *);
long  OSErr    SCSIDeregisterBus(SIMinitInfo *);
```

DMA Serial Driver

The DMA Serial Driver for the Macintosh Quadra 840AV and Macintosh Centris 660AV is a complete reimplementation of the classic serial driver previously documented in *Inside Macintosh.* The reasons for this change are

■ to improve the maintainability and transportability of the serial driver by writing it in a high-level language

■ to modularize hardware-dependent support features, speeding the development of serial driver versions for new hardware

These goals mesh with the extensive changes required to support a DMA serial I/O model on the Macintosh Quadra 840AV and Macintosh Centris 660AV hardware. While the documented API for the DMA Serial Driver is supported and compatible with the classic serial driver, there are a few technical changes internally which could affect driver clients that are not particularly well behaved.

The new Serial Driver does not assume anything about the hardware. Any function that requires knowledge of the hardware results in a call to a **hardware abstract layer (HAL),** an API layer that makes the driver hardware-independent. By supplying a new HAL, the same serial driver can support many different hardware platforms. The first new HAL, called PSCHAL, was developed to support the Macintosh Quadra 840AV and Macintosh Centris 660AV hardware.

It is not necessary to read this chapter to use the new DMA Serial Driver. However, some serial driver clients were written to take advantage of the hardware implementation of the previous serial driver. The internal structures are not the same as in the previous serial driver. Any software that relies on the serial driver's internal structures must be rewritten. Hence, developers wishing to maintain compatibility with the new DMA Serial Driver should read this chapter and test their existing serial driver clients for changes in the hardware implementation.

This chapter explains the change in the architecture of the DMA Serial Driver and then the changes in implementation that could affect existing drivers. For information about serial port hardware in the Macintosh Quadra 840AV and Macintosh Centris 660AV computers, see "Serial Ports," in Chapter 2.

Architecture

At the top level, presenting the familiar Device Manager API, is a serial driver that handles `Open`, `Close`, `Read`, `Write`, `Control`, `Status`, and `KillIO` calls. The driver maintains a set of variables referenced by `dCtlStorage` that are not compatible with the variables of the classic serial driver. The driver never explicitly references the Macintosh hardware and never makes any assumptions about whether the hardware is a standard SCC, SCC with IOP, SCC with PSC, or any other specific configuration. The DMA Serial Driver is a standard 'SERD' resource of ID 1. The preliminary version number for this driver is 8.

To support the documented API, anytime a required function would involve knowledge of the hardware a call is initiated to a serial HAL resource. Through a parameter block interface, the HAL handles requests from the serial driver that require specific knowledge of the hardware.

A HAL is simply a code resource with a predefined, private API. By interchanging HAL resources, the same serial driver can support a number of widely different hardware configurations. The first HAL implemented is PSCHAL, a DMA HAL for the Macintosh Quadra 840AV and Macintosh Centris 660AV. This HAL is largely a superset of what would be required for the traditional Macintosh serial platform; by stripping out some DMA code, for example, a simpler "SCCHAL" for the SCC could be generated.

Changes in Implementation

This section discusses the following areas affected by changes in the hardware and software implementation of the DMA Serial Driver:

■ interrupt handling

■ DMA versus non-DMA transmissions

■ elimination of the `PollProc` mechanism

■ use of the DMA capability

Interrupt Handling

The HAL has responsibility for receiving all interrupts generated by the serial hardware. This is in line with the HAL's responsibility as keeper of the hardware. The HAL dispatches serial driver interrupt handlers through the "Level 2" vector tables, including external/status interrupts. It is the responsibility of the driver to make callbacks to the HAL to perform hardware-dependent tasks at interrupt time, including secondary dispatch of external/status interrupts. Driver-level interrupt handlers usually run as deferred tasks with interrupts enabled.

The interrupt dispatch table structure is preserved as an element of the driver/HAL interface. The familiar `Lvl2DT` (SCCDT) and `ExtStsDT` tables are still used. DMA interrupts are processed through these vectors as well as SCC interrupts, so there is more complexity required in the interrupt handlers to process a given interrupt properly. In general, this complexity is not in the driver but is instead pushed down into the HAL. Register conventions across these dispatch tables may or may not be preserved; for example, SCC addresses may not be stored in registers A0 or A1.

These changes in interrupt handling should be transparent to any serial driver client, but they do significantly alter the interrupt handler code paths from those used in the former serial driver.

DMA Versus Non-DMA Transmissions

The PSC DMA hardware presents a minor limitation in that all serial data transfers must begin on longword boundaries. As a result, not all data can be transferred using DMA. Therefore, PSCHAL uses a mixed DMA/SCC model where DMA is used if possible and convenient. If DMA is not convenient, the classic character-oriented SCC interrupt model is employed until synchronization is regained with a longword boundary. Maximum performance benefit occurs with large, uninterrupted transfers.

When receiving data, there are new requirements on the receive buffer size and alignment. Although the driver client can request any buffer size and alignment, the driver uses only receive buffers which are 64 bytes or larger, aligned to a cache line boundary and a multiple of 16 bytes in length. The driver attempts to ensure that the buffer is also locked in physical memory and physically contiguous. If a buffer passed to `SerSetBuf` does not meet these requirements, the driver attempts to carve out a subset of the given buffer which does meet them. If that is not possible, the driver reverts to its internal default 64-byte buffer. This should have little impact on driver clients, who should make no assumptions about the serial driver's internal use of the receive character buffer. `SerSetBuf` and `PBWrite` will fail if called when interrupts are masked. The driver will be unable to lock the receive buffer for DMA.

PollProc Mechanism

The `PollProc` mechanism, whereby serial characters are received with interrupts disabled by LocalTalk or other applications, is not supported on the Macintosh Quadra 840AV or Macintosh Centris 660AV. `PollProcs` are completely disabled. The PSC is capable of reading incoming serial data while interrupts are disabled. Polling by other software components threatens data integrity just as failure to poll did in the past. All occurrences of polling in components outside the serial driver should be disabled. The driver itself does not supply a `PollDtaIn` equivalent (the `PollProc` low memory is always nil).

DMA Use

PSCHAL uses all three serial DMA channels, each in a fixed direction. On port A, the SCCA DMA channel (channel 4) is used to receive and SCCATx (channel 6) is used to transmit. This allows full-duplex serial DMA on port A. On port B, SCCB (channel 5) is used to transmit. Full-duplex serial DMA is not supported on port B, because the printer port is used primarily for output and not for high-speed input. For hardware details, see "Serial Ports," in Chapter 2.

During DMA input, any `Read` call to the driver and any `SerGetBuf` Status call requires that pending DMA be terminated to determine an accurate accounting of characters received. Terminating DMA ensures that all received characters are immediately available, but degrades driver performance. If your application calls `SerGetBuf` in a loop you might want to rewrite it to work around this requirement.

Video Driver

The Macintosh Quadra 840AV and Macintosh Centris 660AV computers are the first Macintosh CPUs to provide both video-out and video-in capabilities built into the main logic board. This chapter discusses the system software changes that support these features. The hardware for video input and output is discussed in "Video and Graphics I/O," in Chapter 2.

Before reading this chapter, you should already be familiar with video drivers based on the Macintosh Slot Manager. See *Designing Cards and Drivers for the Macintosh Family,* third edition, for background technical information.

Video Television Output

The user can control the video output portion of the video driver in the Macintosh Quadra 840AV and Macintosh Centris 660AV by means of the Monitors control panel, using the Options button. The Macintosh Quadra 840AV and Macintosh Centris 660AV hardware supports video ouput not only through the standard DB-15 monitor connector but also through a composite video connector on the back panel.

In addition to the standard RGB monitor output, video output is available in either NTSC or PAL television format. With NTSC format, underscan produces a resolution of 512 by 384 pixels resolution, while overscan produces a resolution of 640 by 480 pixels. With PAL format, underscan produces a resolution of 640 by 480 pixels, while overscan produces a resolution of 768 by 576 pixels. When driving an interlaced display or television, the hardware can implement a flicker-free mode called *Apple convolution.* This mode is selectable through a checkbox on the Options dialog box of the Monitors control panel. Apple convolution is not supported in more than 256 colors or when a video input window is active.

Because of the limited resolutions of the NTSC and PAL standards, the video driver allows the user to switch from an RGB display to a television output only when the RGB display resolution is 512 by 384, 640 by 480, or 768 by 576 pixels. The driver provides family modes for all Apple monitors in these resolutions, if physically possible. Thus, a user who has a 16-inch color display with a resolution of 832 by 624 pixels can change the family mode to 512 by 384, 640 by 480, or 768 by 576 pixels. The driver will center the active video on the display and the user will see more black around it than in the standard 832 by 624 resolution. After doing this, the Option dialog box of the Monitors control panel will show enabled radio buttons to switch the output to one of the television formats.

The Macintosh Quadra 840AV and Macintosh Centris 660AV video driver lets the user connect a television set as the computer's sole display. This is done by the `PrimaryInit` code; if there is no monitor connected to the DB-15 port, the code checks a bit in its slot PRAM to determine whether the user has enabled the boot-on-television feature. If the bit is set, the video driver opens and the monitor output is displayed on television equipment connected to the composite output ports. The Options dialog box of the Monitors control panel provides a checkbox to allow the user to select this feature.

Monitor output is directed to the video output connector in television format only if there is no monitor connected to the DB-15 connector. If the user has not clicked the checkbox in the Options dialog box of the Monitors control panel, this feature can also be enabled by holding down the Command-Option-T-V keys during startup. If this is done, the machine will boot up, play the boot beep, and replay the boot beep a short time later. At that moment the user can release the keys and the computer will continue the startup process, using the connected television set as its main display.

New Control and Status Routines

To let video displays go into a power-saving mode if the sync lines are dropped, two new routines have been added to the video driver:

csCode = 11	**csParam**	**=VDFlagPtr**	**[SetSyncs]**
→	csModeflag	mode value	[byte]

csCode = 11	**csParam**	**=VDFlagPtr**	**[GetSyncs]**
←	csModeflag	mode value	[byte]

The `SetSyncs` control routine promotes evergy conservation by disabling the sync outputs going to the monitor, thereby setting power-saving monitors in a low-power mode. The same routine can then be used to reenable the syncs outputs. A `csMode` value of 0 enables the sync outputs, and a `csMode` value of nonzero disables the sync outputs. While the sync outputs are disabled, the monitor will show black.

The `GetSyncs` status routine returns a value that indicates the state of sync outputs. If `csMode` is 0 it means that the syncs are enabled, and if `csMode` is nonzero it means they are disabled.

NuBus Block Moves

Video data movement to and from accessory cards often require block transfers, which are supported by the MUNI chip as described in "NuBus Interface," in Chapter 2. Block transfers from NuBus are always enabled, but block transfers to NuBus must be enabled by one of the following two procedures:

■ by programming the card's configuration ROM

■ by using the trap macro `_SlotBlockXferCtl`

These procedures are described in the next sections.

Note
The system software fully supports the NuBus block transfer sResource IDs. The `sBlockTransferInfo` and `sMaxLockedTransferCountsResource` IDs are included in the system's board sResource. ◆

Configuration ROM Programming

The configuration ROM on the card must support slave block transfers of size 4, which is the only size that the MUNI can generate. The Macintosh system searches the card's configuration ROM after `PrimaryInit` has run, and looks in the board's sResource list for the sResource ID of the `sBlockTransferInfo` data structure. If the sResource ID indicates that the card supports slave transfer sizes of size 4, the MUNI will be programmed to enable block transfers to that slot. The ROM does not support the automatic enabling of block transfers to NuBus if these transfers are not supported in all the operational modes of the card. For further information, see *Designing Cards and Drivers for the Macintosh Family,* third edition, and the *NuBus Block Transfers* technical note.

Using the Trap Macro SlotBlockXferCtl

You can also use a programmatic interface to enable or disable block transfers to NuBus. The trap macro `_SlotBlockXferCtl` is accessed through the `_HwPriv` trap, with a selector of `0x0c`. The interface is the following:

```
Trap Macro:        _SlotBlockXferCtl

HwPriv Selector:   0x0c

On Entry:   A0 (long)   (bits 31-9) reserved
                        (bit 8)     0 to disable block xfer to
                                      a slot, 1 to turn it on
                        (bits 7-0)  slot number, range 1-14

On Exit:    D0 (long)   0 if we're on a MUNI-based system & good
                        slot value, paramErr if not

            A0 (long)   if noerr, previous state of block xfer for
                        each slot (1 = on, 0 = off)
                        (Bits 31-15 reserved, Bit 14 = slot 14,
                          bit 1 = slot 1, bit 0 reserved)

Destroys:   D1,D2,A1
```

New Age Floppy Disk Driver

The system software for the Macintosh Quadra 840AV and Macintosh Centris 660AV computers contains a modified version of the traditional floppy disk driver covered in *Inside Macintosh.* The new version is designed to support the New Age floppy disk controller, described on page 15.

This chapter describes the support in the Macintosh Quadra 840AV and Macintosh Centris 660AV for floppy disk reading and writing, plus changes to the floppy disk driver operation and API.

Floppy Disk Support

The New Age floppy disk driver supports the Apple 800K GCR floppy disk drive and the Apple SuperDrive floppy disk drive. It does not support the Apple 400K GCR floppy disk drive or the Macintosh HD20 hard disk drive.

With an Apple 800K GCR drive, the New Age floppy disk driver reads from and writes to the following disk formats:

■ Apple 400K

■ Apple 800K

■ ProDos GCR

With an Apple SuperDrive, the Newage floppy disk driver reads from and writes to the formats just listed plus the following:

■ 720K MFM disks

■ 1440K MFM disks

Programming Interface Changes

The New Age floppy disk driver is very similar to the floppy disk driver used in Macintosh Quadra computers and previous models. Most of the prime, control, and status routines are supported and should appear the same to application software; the calling conventions are identical. However, three control routines—`TrackCache`, `KillI/O`, and `TagBuffer`—are no longer supported.

`TrackCache`, a control routine with a `csCode` of 9, is no longer supported because the read process would try to cache everything on the track being read. If it failed to read everything on that track, as it might on a copy-protected disk, it would only read and cache what was requested. Similarly, the write process would cache up to a track of data being written out.

`KillI/O`, a control routine with a `csCode` of 1, and `TagBuffer`, a control routine with a `csCode` of 8, are also not implemented. Calls to `TagBuffer` return a result code of –17 and calls to `KillI/O` return a result code of –1.

Operational Compatibility

Besides the three unsupported control routines listed in the previous section, there are a few minor differences between the New Age floppy disk driver and previous Macintosh floppy disk drivers.

A call to `TrackDump` with search mode 0 no longer starts its data stream at the beginning of the track. Instead, it starts after the address field of the first sector (GCR sector 0 or MFM sector 1). `TrackDump` is a control routine with a `csCode` of 8.

A call to `DriveStatus` with a drive reference number that identifies an uninstalled floppy drive returns an error code of –56 and puts invalid data in the `csParam` field. A call to `DriveStatus` with a drive reference number of 0 or 1 returns valid data. `DriveStatus` is a status routine with a `csCode` of 8.

The New Age floppy disk driver does not return any of the following error codes:

`noDriveErr`	–64	Drive not installed
`badBtSlpErr`	–70	One of the address mark bit slip nibbles was incorrect (GCR)
`badDBtSlp`	–73	One of the data mark bit slip nibbles was incorrect (GCR)
`initIWMErr`	–77	Unable to initialize IWM
`twoSideErr`	–78	Tried to read a double-sided disk on a single-sided drive
`spdAdjErr`	–79	Unable to correctly adjust the drive speed (GCR, 400K drives only)
`seekErr`	–80	Wrong track number read in sector's address field

Floppy driver calls to an uninstalled drive return an `nsDrvErr` error (no such drive error) instead of `noDriveErr`.

The New Age controller returns only one error code for a bad address mark. There is no differentiation in the address mark between a bad slip bit and a wrong track number. Consequently, the `badBtSlpErr`, `seekErr`, and `noAdrMkErr` (couldn't find valid address mark) errors have all been merged into `noAdrMkErr`. Similiarly, `badDBtSlp` and `noDtaMkErr` (couldn't find valid address mark) have been merged into `noDtaMkErr`.

The error codes `initIWMErr`, `twoSideErr`, and `spdAdjErr` are not applicable to the New Age driver.

The `noNybErr` error used to mean a byte timeout. With the New Age driver it indicates a timeout error resulting from waiting for New Age to respond to a command.

Virtual Memory Manager

There is one substantial change to the Virtual Memory Manager in the Macintosh Quadra 840AV and Macintosh Centris 660AV, made to accommodate the new SCSI Manager (described in Chapter 9, "SCSI Manager 4.3").

Virtual memory (VM) no longer disables interrupts when executing these tasks:

■ I/O completion routines

■ Time Manager tasks

■ VBL/slot VBL tasks

■ deferred tasks (as they exist today)

■ `PPostEvent` actions

These tasks are placed in a deferred user function queue. If a user function, such as a completion routine, is requested while the VM is running the deferred user function queue (with interrupts enabled), VM places the user function at the end of the deferred user function queue. This ensures that routines of the types listed above will execute in their original order.

In earlier Macintosh systems, while virtual memory is servicing a page fault it defers the execution of I/O completion routines, Time Manager tasks, VBL and slot VBL tasks, Deferred Tasks, and `PPostEvents` until it is page fault safe. VM disables dispatching of the VBL/Slot VBL tasks and the Deferred Tasks when it services a page fault. I/O completion routines, Time Manager tasks and `PPostEvent` actions, are placed in a deferred user function queue. Some Interrupt Service routines may execute the `DeferUserFn` trap to install code in the same deferred user function queue. These deferred user functions are run only when VM is sure that it is safe. When VM runs these functions it disables interrupts until the entire deferred user function queue is emptied. In earlier systems, this was a simple way to ensure that these asynchronous tasks were executed in the order they were queued.

VM now executes these functions without disabling interrupts. For these routines to execute in the expected order, if a user function (like a completion routine) is to be run while VM is running the deferred user function queue (with interrupts enabled), VM places this new completion routine at the tail of the deferred user function queue.

For general information about memory implementation in the Macintosh Quadra 840AV and Macintosh Centris 660AV, see Chapter 2, "Hardware Details."

Appendixes

This part of the *Macintosh Quadra 840AV and Macintosh Centris 660AV Developer Note* contains four appendixes. They contain information that can help you with specific development tasks:

■ Appendix A, "DSP d Commands for MacsBug," describes three new d commands added to Macsbug that help in debugging DSP code.

■ Appendix B, "BugLite User's Guide," covers a DSP module installer with a graphical user interface. It helps programmers create and install tasks to be executed by the DSP.

■ Appendix C, "Snoopy User's Guide," tells you how to use a browser and debugger for the DSP. It helps programmers debug real-time tasks that run on the DSP.

■ Appendix D, "Mechanical Details" contains foldout drawings of the physical mounting facilities that are provided for internal SCSI devices and accessory cards in the Macintosh Quadra 840AV and Macintosh Centris 660AV.

DSP d Commands for MacsBug

This appendix describes new MacsBug d commands used for debugging DSP3210 digital signal processor code being run on Macintosh platforms.

These d commands are specific to the DSP3210. The dissassembly instruction assumes the data is in DSP3210 code format. Before using MacsBug to locate a problem in the DSP3210 code you should first attempt to use Snoopy, the DSP browser/debugger. Additional information about d commands can be found in the MacsBug and Macintosh debugging documentation available from APDA.

The first section, "Getting Started," tells you how to install the new d commands in MacsBug. The next section, "Using the d Commands," shows how to find the specific DSP desired and locate a specific module and section running on that DSP. The last section, "d Commands Reference," provides a description of how each command is used and shows the default template used by each command.

Getting Started

Use ResEdit to install the d commands and templates into the Debugger Prefs file.

There are four basic d commands used in DSP3210 debugging and twenty five templates. The d commands are used to show information about the DSPs and the clients, tasks, modules, and sections that are installed on each one.

Using the d Commands

To locate the data you are interested in you must first find out what devices are available. Use the dsps command, which produces a display such as the following:

```
dsps
 Device    Name     Ref  Clients Slot Proc TimeShare RealTime FrameCt EVT
 000dd740 .DSP3210 ffca 0002     000e 0000 00000000 fee387cc 00015351 00015351

 Task      Name       RefNum   Modules  Flags   Vector    Client    RefCon
 fee387cc Input      fee387cc fee385e4 AtR      00146984 000dd80c 00000000
 fee37884 Preput     fee37884 00000000 atR      00146984 000dd80c 00000000
 fee37808 Midput     fee37808 00000000 atR      00146984 000dd80c 00000000
 fee3778c Postput    fee3778c 00000000 atR      00146984 000dd80c 00000000
 fee37710 Output     fee37710 fee37528 AtR      00146984 000dd80c 00000000
```

Second, find the specific task of interest and use the `Modules` location in the `md` command to display the sections that make up the module. This example uses the first module `Input` that is located at `fee385e4`.

```
md FEE385E4
 Module    Name        Flags Sections Execution   SkipCount Actual    Estimate
 fee385e4 Input        dMsd  00000005 000159b5    00000000  000159b5 00000c80

 Section   Name        Flags   Index     Size      Primary  Secondary Type
 fee38654 Program      LscwAbD 00000000 00000118  5003e100 fee38488  iosft
 fee38694 LAIAO        lscWabD 00000001 000003c0  5003f640 00000000  iOSft
 fee386d4 RAIAO        lscWabD 00000002 000003c0  5003f280 00000000  iOSft
 fee38714 Temp         lscWaBD 00000003 000003c0  5003e218 00000000  ioSft
 fee38754 Globals      LScWaBD 00000004 00000008  5003e5d8 fee37900  iosft
```

Third, select the code section of interest and disassemble it with the `il3210` command. This example uses the first section located at `fee38488`.

```
IL3210 FEE38488
 Disassembling from fee38488
          fee38488   9de5c817   *r21++ = (long) r5
          fee3848c   9de6c817   *r21++ = (long) r6
          fee38490   9df4c817   *r21++ = (long) r18
          fee38494   14200004   r1 = (short) 0x4(4)
          fee38498   969a02ac   r18 = (long) r22 + 0x2ac(684)
          fee3849c   9cf4a000   r18 = (long) *r18
          fee384a0   80000000   NOP
          fee384a4   12940000   call•r18 (r18)
          fee384a8   80000000   NOP
          fee384ac   98050022   r5 = (long) r0 + r2
          fee384b0   949a0310   r4 = (long) r22 + 0x310(784)
          fee384b4   9ce42000   r4 = (long) *r4
          fee384b8   947a03c4   r3 = (long) r22 + 0x3c4(964)
          fee384bc   9ce31800   r3 = (long) *r3
          fee384c0   94240004   r1 = (long) r4 + 0x4(4)
          fee384c4   9ce10800   r1 = (long) *r1
          fee384c8   9be30001   (long) r3 & 1(0x1)
          fee384cc   98010885   if (ne) r1 = (long) r1 + r5
          fee384d0   94d5000c   r6 = (long) r19 + 0xc(12)
          fee384d4   9ce63000   r6 = (long) *r6
```

Additional information can be obtained by using the display memory command `DM` and the templates.

d Commands Reference

The three d commands used in DSP3210 debugging (besides DM) are listed in Table A-1.

Table A-1   d commands

Command	Description
dsps	Display all DSP CPU devices and their associated tasks.
il3210	Disassemble *n* lines of DSP32C from the address specified. If no number is specified, then display half page.
md	Display a list of the modules and their associated sections.

These d commands have predefined templates that are used to display the information in a specific format.

dsps

SYNTAX

```
dsps
```

DESCRIPTION

The dsps command displays all DSP CPU devices and their associated tasks.

This command displays all DSP devices installed in the computer. It also shows all tasks installed and relevent information for finding them in memory. Modules that are installed in a specific task can be displayed using the Modules reference address. The current status of the task is specified by the Task flags shown in Table A-2. Upper case letters indicate the *true* state, lower case letters indicate the *false* state of the flag.

Table A-2   Task flags

Task flags	Description
A	Task is active
T	Toggle the active bit to set the task active
R	Task is in the real-time task list

In the example, the only tasks that are active are input and output. All of the other tasks are inactive and are not set to become active. All of the tasks are in the real-time task list.

EXAMPLE

```
dsps
 Device   Name      Ref   Clients Slot Proc TimeShare RealTime FrameCt EVT
 000dd740 .DSP3210 ffca 0002    000e 0000 00000000 fee387cc 00015351 00015351

 Task      Name      RefNum    Modules  Flags     Vector    Client    RefCon
 fee387cc Input     fee387cc fee385e4 AtR       00146984  000dd80c  00000000
 fee37884 Preput    fee37884 00000000 atR       00146984  000dd80c  00000000
 fee37808 Midput    fee37808 00000000 atR       00146984  000dd80c  00000000
 fee3778c Postput   fee3778c 00000000 atR       00146984  000dd80c  00000000
 fee37710 Output    fee37710 fee37528 AtR       00146984  000dd80c  00000000
```

il3210

SYNTAX

il3210 [*addr* [*n*]]

DESCRIPTION

The il3210 command disassembles *n* lines of dsp3210 code, starting at address *addr*. If no *n* is given, then it displays half page. This command disassembles the data starting at *addr* into DSP3210 code format.

EXAMPLE

```
li3210 FEE38488
  Disassembling from FEE38488
           fee38488   9de5c817   *r21++ = (long) r5
           fee3848c   9de6c817   *r21++ = (long) r6
           fee38490   9df4c817   *r21++ = (long) r18
           fee38494   14200004   r1 = (short) 0x4(4)
           fee38498   969a02ac   r18 = (long) r22 + 0x2ac(684)
           fee3849c   9cf4a000   r18 = (long) *r18
           fee384a0   80000000   NOP
           fee384a4   12940000   call•r18 (r18)
           fee384a8   80000000   NOP
           fee384ac   98050022   r5 = (long) r0 + r2
```

```
fee384b0   949a0310   r4 = (long) r22 + 0x310(784)
fee384b4   9ce42000   r4 = (long) *r4
fee384b8   947a03c4   r3 = (long) r22 + 0x3c4(964)
fee384bc   9ce31800   r3 = (long) *r3
fee384c0   94240004   r1 = (long) r4 + 0x4(4)
fee384c4   9ce10800   r1 = (long) *r1
fee384c8   9be30001   (long) r3 & 1(0x1)
fee384cc   98010885   if (ne) r1 = (long) r1 + r5
fee384d0   94d5000c   r6 = (long) r19 + 0xc(12)
fee384d4   9ce63000   r6 = (long) *r6
```

md

SYNTAX

md [*modulepointer*]

DESCRIPTION

The md command displays modules in a list with their associated sections. Flags are
listed in Table A-3 through Table A-5.

Table A-3   Module flags

Module flag	Description
D	kdspDemandCache
M	kdspModuleAllocated
A	kdspUseActual
D	kdspDontCountThisModule

Table A-4   Section flags

Section flag	Description
L	kdspLoadSection
S	kdspSaveSection
C	kdspClearSection
W	kdspSaveOnContextSwitch

continued

Table A-4   Section flags (continued)

Section flag	Description
A	kdspBankA
B	kdspBankB
D	kdspDSPUseOnly

Table A-5   Section types

Section type	Description
I	kdspInputBuffer
O	kdspOutputBuffer
S	kdspScalableSection
F	kdspFIFOSection
T	kdspITBSection

EXAMPLE

```
md FEE385E4
 Module    Name      Flags   Sections Execution SkipCount Actual    Estimate
 fee385e4 Input      dMsd    00000005 000159b5  00000000  000159b5  00000c80

 Section   Name      Flags   Index    Size      Primary   Secondary Type
 fee38654 Program    LscwAbD 00000000 00000118  5003e100  fee38488  iosft
 fee38694 LAIAO      lscWabD 00000001 000003c0  5003f640  00000000  iOSft
 fee386d4 RAIAO      lscWabD 00000002 000003c0  5003f280  00000000  iOSft
 fee38714 Temp       lscWaBD 00000003 000003c0  5003e218  00000000  iOSft
 fee38754 Globals    LScWaBD 00000004 00000008  5003e5d8  fee37900  iosft
```

BugLite User's Guide

This appendix describes the user interface for BugLite, a tool for accessing and installing digital signal processor (DSP) modules, as DSP tasks, in the real-time data processing subsystem of the Macintosh Quadra 840AV or Macintosh Centris 660AV. The section "Getting Started" describes how to install the application and provides information on the initial display. "Tools of the Trade" describes what the BugLite tools are and how they operate.

"Using BugLite" describes how to select and load a DSP program module. The example also shows how to use the tools in creating a DSP task that plays a record from disk. The final section, "Getting Information," shows what information is available about each module and how to access it.

BugLite is a graphical DSP module installer that allows the DSP programmer to select DSP modules from any mass storage device (for example, a hard disk) and install them into a DSP subsystem. Using the graphical representation of tasks and modules, predefined resource modules can be assembled into a task and run on the DSP subsystem. This relieves the DSP programmer from having to generate a Macintosh application to test DSP code. Additional capabilities provide access to external data files and I/O ports for connecting the task into real data.

For more information on digital signal processing, see Chapter 3, "Introduction to Real-Time Data Processing." Although multiple DSP operations are not available on the Macintosh Quadra 840AV or Macintosh Centris 660AV computer, they are documented here for completeness.

To run BugLite, you need system software version 7.1 or later and at least 1,024 KB available RAM; the preferred size is 1,024 KB.

Getting Started

This section tells you how to install and launch the BugLite tool.

Installation

BugLite operates as an application running on the main processor. Since it relies on the DSP Manager that is in the Macintosh Quadra 840AV or Macintosh Centris 660AV ROM there are no system files to be installed. To use BugLite

■ copy the application to your hard drive
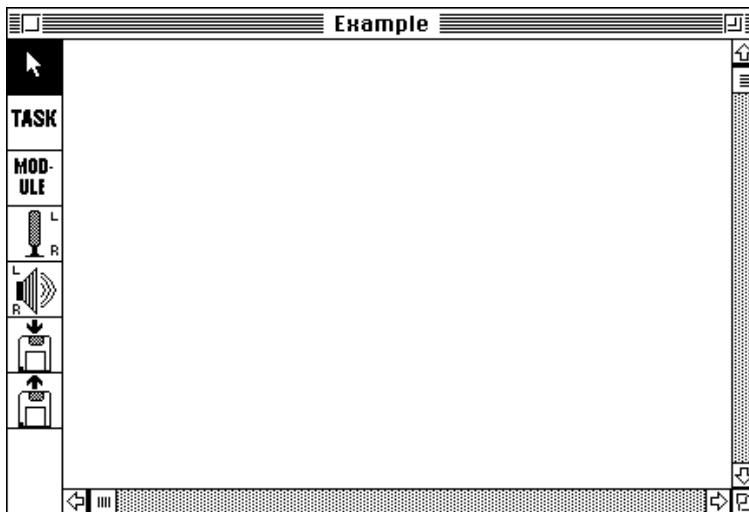
■ launch BugLite

BugLite can reside anywhere on your drive. However, you may find it useful to have BugLite in the same directory as your DSP object code so you don't have to search through multiple directories to locate your source files.

What You See When You Launch BugLite

There are several different objects in BugLite: tasks, modules, sections, and input and output icons. All of these objects are displayed and manipulated graphically within a *task window*. After launching BugLite, the task window, shown in Figure B-1, is displayed. It is within this task window that a task is configured to run on the DSP.

Figure B-1   Task window

The task window displays tasks with their associated modules and any subsystem elements (disk file input or output, sound input or output). It is within this task window that you can create tasks, load modules, and connect sections to other sections, the microphone, the speaker, or disk files. On the left side of the task window is the tool palette, discussed in the next section. Once the task has been configured it can be loaded onto the DSP and executed by selecting the Run button directly below the task's name. See "Using BugLite," later in this appendix.

Tools of the Trade

The tool palette, on the left side of the task window, is used to switch modes of operation within BugLite. The current active tool is highlighted (inverted). Clicking on a tool will make it the current tool. The seven tools (from top to bottom) are shown in this section.

ARROW tool; used to select objects to position them on the screen and connect objects.

TASK tool; used to create a task.

The task tool creates one task. Clicking anywhere in a task window will prompt you for the task name and add the new task to the window. The task can be moved around and connected to modules with the arrow tool.
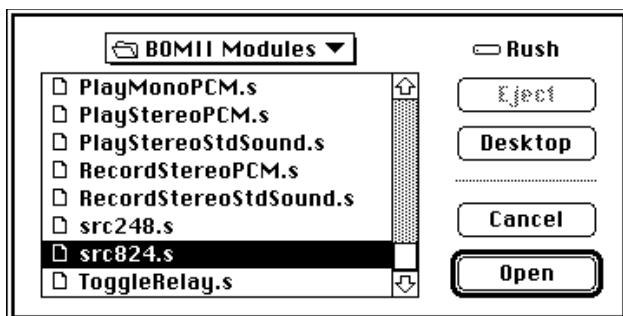
BugLite supports multiple tasks within the task window.

MODULE tool; used to open a module resource file. The standard file selection box will ask for the file to be opened.

The module tool creates one module. Clicking anywhere in a task window will prompt you for the module resource file to open, as shown in Figure B-2.

Figure B-2   Open File dialog box

Normally a module resource file is created using the following steps:

1. **Use the MPW editor to create the DSP3210 source code (example: `Reverb.s`)**

2. **Use the `d32asm` script to build the resource and put it into the source file:**

   `d32asm Reverb.s Reverb.s`   Builds resource and puts it back into the source file

INPUT tool; creates an object that provides access to the stereo sound input stream. This icon must then be connected to the appropriate input AIAO section. The data stream is 3210 floating-point numbers.

OUTPUT tool; creates an object that provides the ability to sum a signal into the stereo output stream. This icon must then be connected to the appropriate output AIAO section. The data stream is 3210 floating-point numbers.

DISK RECORD tool; used to create a new AIFF disk file that can be connected to a FIFO section to store data.

DISK PLAY tool; used to open an existing AIFF file that can be connected to a FIFO section for data input.

All objects in the task window have one or more triangular *nibs* (▶) associated with them. Nibs on the right side of an object are considered output and nibs on the left side are considered input. Two objects are connected by connecting their nibs with the arrow tool. These are the current valid connections:

■ Task to module: installs module into task.

■ Module to module: installs second module into task.

■ Section to section (both sections have same data type and size): allows section data to be shared between two modules.

■ Sound input to AIAO: provides real-time data acquisition from the built-in microphone.

■ AIAO to speaker output: provides real-time data playback to the built-in speaker.

■ AIAO section to disk object: provides access to data on disk or saving data to disk.

Using BugLite
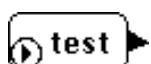
Before attempting to install a task onto the DSP subsystem each module's specification must be reviewed so that correct connections of module sections can be done. BugLite does only minimal checking for incompatible buffers.

To make a task, follow these steps:

1. **Click anywhere in the task window with the TASK tool.**
   The TASK tool will ask for a name for the new task. The name "test" is used in the example, as shown in Figure B-3.

Figure B-3   Graphical representation of a task

2. **Select the first module to be loaded.**
Figure B-4 shows a file input module "File In" used for getting data into the DSP data stream. The MODULE tool uses the standard file selection dialog box to make the selection.

Figure B-4   Graphical representation of a module
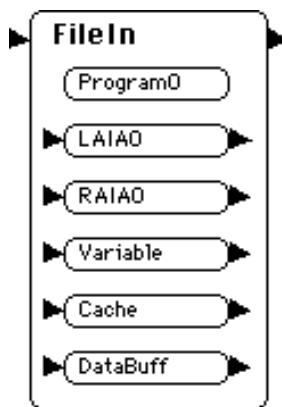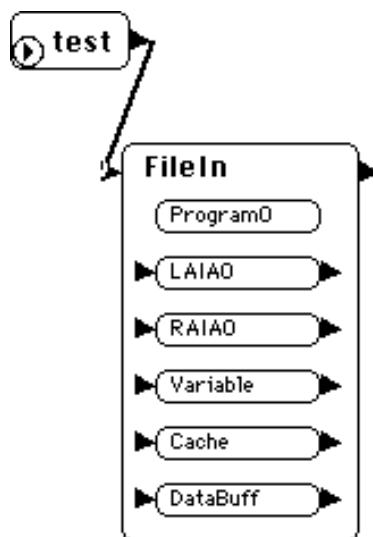
3. **Connect the TASK to the MODULE using the ARROW tool.**
Figure B-5 shows a task connected to a module.

Figure B-5   Task connected to a module

In this example there is a task called "test" that was created with the TASK tool. The task has one module, "File In," connected to it.

BugLite User's Guide

4. **Select the DISK PLAY tool.**
   The DISK PLAY tool uses the standard file selection dialog box to make the selection.

Figure B-6   Disk play of "funky" file

In this example there is a data file called "funky" that is stored on disk. This data is in the AIFF data format.

5. **Connect the DISK PLAY icon ("funky" file) to the "DataBuff" section using the ARROW tool.**

Figure B-7   Disk player connected to input buffer
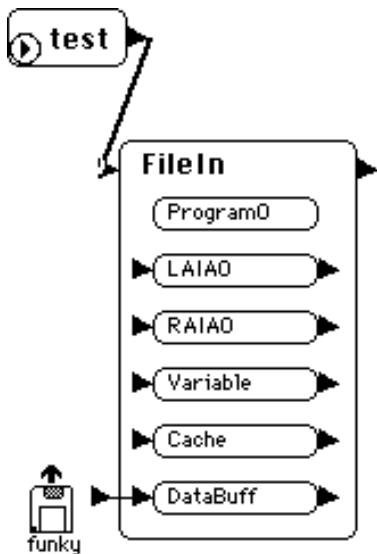
6. **Select the OUTPUT tool and place the icon to the right of the module.**
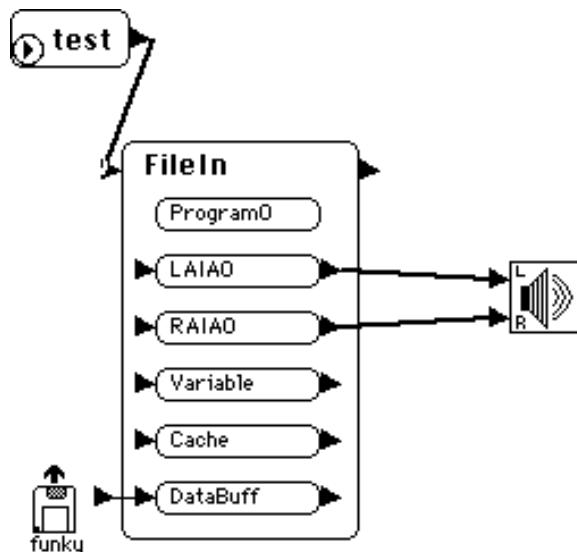
Figure B-8   Speaker connection icon

APPENDIX B

BugLite User's Guide
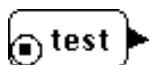
7. **Connect the L and R inputs of the OUTPUT icon to the "LAIAO" and "RAIAO"sections respectively using the ARROW tool.**

Figure B-9   Data output buffers connected to speakers

8. **Click once on the diamond shaped start button in the "test" task icon.**
   The diamond will change into a blinking square.

Figure B-10   Task with task active indicator

The task has now been loaded into the DSP subsystem and the sounds recorded in the "funky" data file will play out of the DSP subsystem audio output.

To disconnect two objects, click in one of the nibs and drag away from it. If the disconnected object is part of a task that is executing, the task is stopped and removed from the DSP.

To delete an object from the task window first select it by clicking on it with the arrow tool, then press the delete key. If the deleted object is part of a task that is executing, the task is stopped and removed from the DSP.

Getting Information

In the menu bar under Object is the Get Info selection. This selection provides information about the selected item. Each task, module, and section has specific kinds of data. Double-clicking an object will also bring up the object's information window.
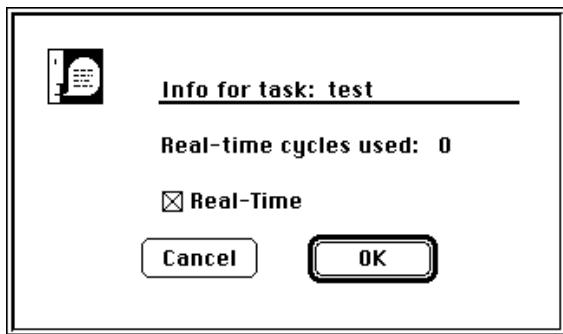
Task Info Window

The task info window provides information on

■ real-time DSP cycles used in processor clock cycles

■ real-time or timeshare mode selection

In Figure B-11 the task "test" shows zero real-time cycles used, indicating that the task has not been run. The task has been set up to be inserted into the real-time task list. This will result in it being allocated a guaranteed bandwidth on the DSP.

Figure B-11   Task Get Info window

Module Info Window

The module information window provides information on

■ Real-time DSP cycles used in processor clock cycles

■ Number of DSP frames that have been executed since the module was installed

■ Skip count setting for this module

APPENDIX B

BugLite User's Guide
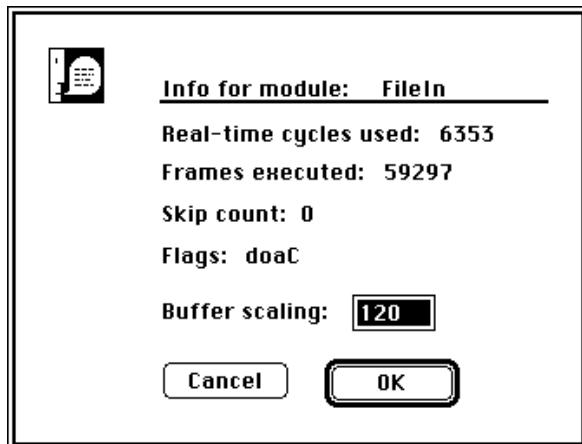
■ Flags for the module: a capital letter means flag is set, a lowercase letter means the flag is clear:

Flag   **Meaning**

d   Demand cache

o   On-chip section table

a   Use actual GPB

c   Count this module in GPB calculation

■ Buffer scaling for this module, user changeable for configuration testing

In the Get Info window shown in Figure B-12, the real-time cycles used are in DSP clock cycles. For this test, the DSP3210 had a frame time of 10 ms and a clock rate of 50 MHz. This results in a total of 500,000 DSP cycles per frame. This module used .0127 percent of the available DSP bandwidth. The value for real-time cycles used is the maximum cycles used in a single frame during a run of 59297 frames.

Figure B-12   Module Get Info window

Info for module: FileIn

Real-time cycles used: 6353

Frames executed: 59297

Skip count: 0

Flags: doaC

Buffer scaling: 120

Cancel   OK

Section Information

The section information window, shown in Figure B-13, provides information on

- Size of the section in bytes

- Type of section: capital letter means flag is set; lower-case means flag is clear:

Flag	Meaning
i	Input buffer
o	Output buffer
s	Scalable section
t	Static section

- Flag settings: capital letter means flag is set; lower-case means flag is clear:

Flag	Meaning
l	Load section
s	Save section
c	Clear section
w	Save on context switch
a	Bank A
b	Bank B
d	DSP use only (only DSP should modify this memory)

The Get Info window in Figure B-13 can be decoded as

- section size: 960 Bytes

- section type: Not a defined section type

- caching flags: Save section, Clear section, Save on context switch, Load section into Bank B, DSP use only

Figure B-13   Section Get Info window

Snoopy User's Guide

Snoopy is a powerful, browser/debugger, in the tradition of SourceBug, for the DSP programming environment. Like SourceBug, it provides breakpoint, single stepping, and code disassembly capabilities. Unlike SourceBug, it also provides editing capabilities and operates on code already installed in the system. Snoopy does not support source-level debugging at this time.

This appendix describes how to use the Snoopy debugger.

The section "Getting Started" tells how to install the Snoopy application and provides detailed information about how it works.

"Using Snoopy" provides valuable information about the menu commands and how to access additional controls and selections when available. The menu commands are listed in the order they would most likely be used. When a command invokes a dialog box, the selections available in the box are discussed immediately following the command. The windows available for displaying more information are detailed in "Additional Information Windows."

To run Snoopy, you need system software 7.1 or later and at least 128 KB of available RAM; preferred size is 768 KB.

Getting Started

This section tells you how to install and launch the Snoopy debugger.

Installation

Snoopy operates as an application running on the Macintosh Quadra 840AV or Macintosh Centris 660AV main processor. To use it:
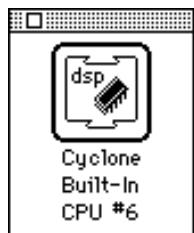
1. Copy the application to your hard drive.

2. Launch Snoopy.

Snoopy can reside anywhere on your hard drive. However, you may find it useful to have Snoopy in the same directory as your DSP object code so you don't have to search through multiple directories to locate your source or symbol files. See "Module Menu," later in this appendix, for instructions on loading and removing symbols.

What You See When You Launch Snoopy

When Snoopy is launched the DSP Control window will show information about the built-in DSP. In multiple DSP systems there would be an icon for each processor. To select the processor to debug, click on the ICON for the desired processor. Figure C-1 shows the ICON for the built-in DSP. To show and hide the DSP Control window see "Windows Menu," later in this appendix.

Figure C-1   DSP Control window

The Real Time Tasks window, shown in Figure C-2, is the primary display window for both code and data display. If there are tasks currently installed on the DSP, they are shown in the scrollable lists at the top of the window.

Note
Standard sound automatically installs itself onto the DSP
to enable the sound functions. ◆
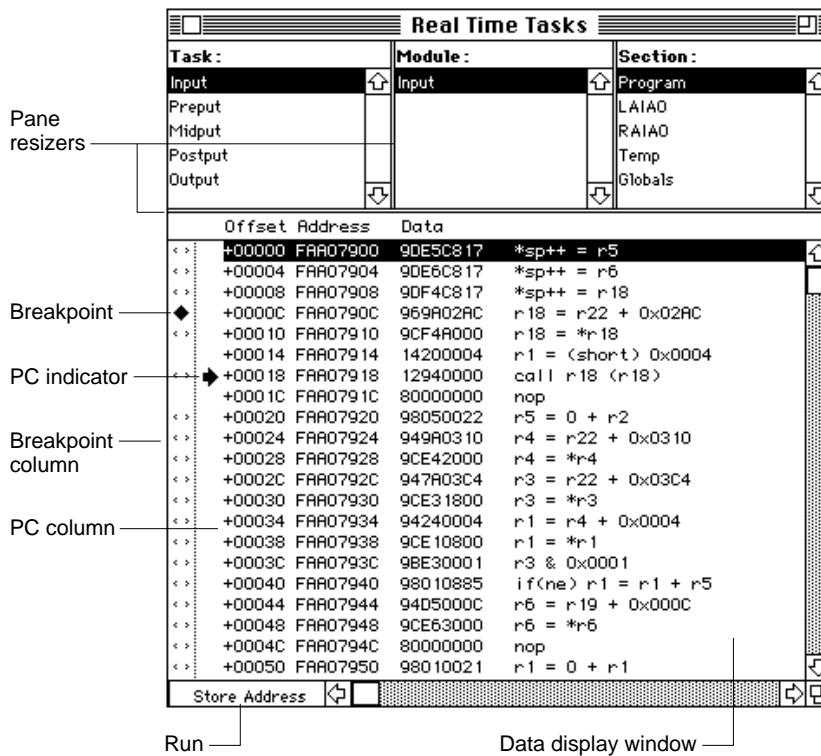
Task/Module/Section Lists

At the top of the Real-Time Tasks window are three scrollable lists that contain (from left to right) the currently installed tasks, the modules belonging to the currently highlighted task, and the sections belonging to the currently highlighted module. This hierarchical representation makes navigation through the potentially large number of tasks, modules, and sections straightforward and intuitive.

For example, in Figure C-2 there is one task installed in the system named Input. Task Input has one module, Input, that has four sections. The section currently being displayed in the Data Display window is Program. If you wanted to view the Temp section, you would simply click on Temp in the list, and it would appear in the Data Display window.

The Data Display Window

The lower half of the Real-Time Tasks window contains the Data Display window. The data belonging to the currently selected section is displayed here in the current format. See "Formatting," later in this appendix, for details.

Snoopy User's Guide

Figure C-2   Real Time Tasks window

Run/Store Address Pop-up Menu

The DSP sections have (potentially) two containers; one at the "storage" address, and the other at the "run" address. The storage address, if there is one, is usually off-chip (in host DRAM or local SRAM), while the run address can either be off-chip or on-chip. The Run/ Store Address pop-up menu, shown in Figure C-3, allows you to select which location you wish to view.

Note
Because the DSP operating system is a caching operating system, it is difficult, if not impossible, to present cached data in a meaningful way while the DSP is running. Consequently, when you switch to a cached "Run Address" and the machine is running, a message will appear in the Data Display window indicating that the data is unavailable. Similarly, if you stop the DSP and attempt to display a cached run address that has yet to be cached, a message will appear indicating that the section is out of scope. ◆

Figure C-3   Run/Store Address pop-up menu

PC Column

The PC (program counter) indicator (an arrow) appears in the PC column, at the appropriate offset, indicating where the DSP is stopped in the specified section. Any window showing the data where the program counter has stopped will also show the PC indicator arrow.
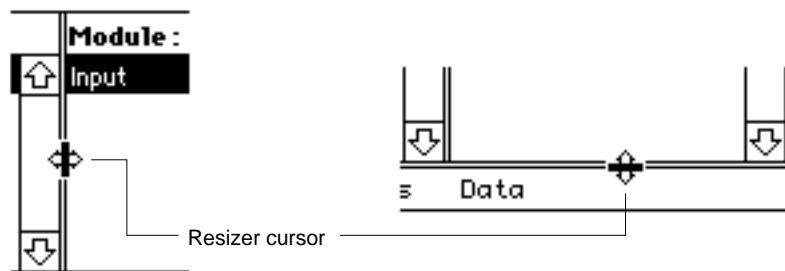
The Breakpoint Column

The breakpoint column, located at the left of the PC column, shows both the current breakpoints and the possible breakpoint/single step locations (called breakpoint candidates). Current breakpoints are indicated by a diamond, and breakpoint candidates are indicated by gray brackets. Breakpoint restrictions are discussed in detail in "Setting and Clearing Breakpoints," later in this appendix.

Pane Resizers

As you can see, the Real-Time Tasks window and the other data display windows are divided by double lines into panes, each of which is resizable. To resize a pane, simply place the cursor on the Pane Resizers, click, drag, and release at the desired point. The cursor becomes a pair of opposing arrows as shown in Figure C-4. Resizing allows you to optimize your screen's real estate.

Figure C-4   Vertical and horizontal pane resizers
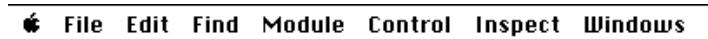
Using Snoopy

This section explains the use of all Snoopy's menu items. There are additional controls that are accessed using either a double click of the mouse or the Option key and a mouse click. These additional controls are explained where appropriate.
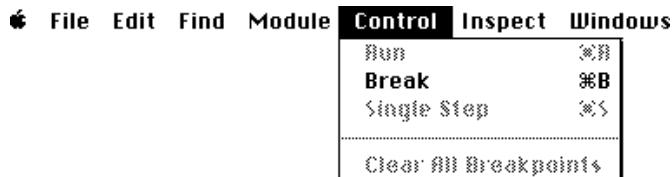
Snoopy User's Guide

Menu Bar
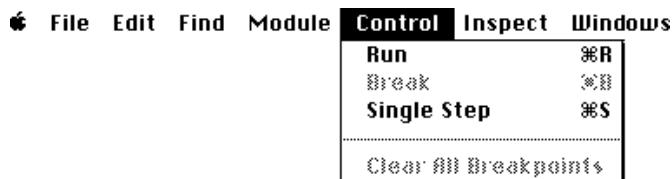
There are two standard menus: File and Edit. The other five menus are specific to
Snoopy, as shown in Figure C-5.

Figure C-5   Menu bar

Control Menu

The Control menu has commands for running (Run), stopping (Break), and single
stepping (Single Step) the DSP. If the DSP is running only the Break command will be
available, as shown in Figure C-6. In order to view a cached program or data from a
module, the DSP must be stopped while executing in the module's program section. This
is done by first using the Break command to halt the DSP, then setting a breakpoint in
the desired module. Breakpoints are explained in "Setting and Clearing Breakpoints,"
later in this appendix.

Figure C-6   Control menu

After the DSP has been stopped using the Break command, the Run and Single Step
commands become available, as shown in Figure C-7.
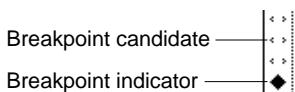
Figure C-7   Control commands after break

The Clear All Breakpoints command is available only if the DSP is stopped and there are
one or more breakpoints set.

Using Snoopy   **441**

Setting and Clearing Breakpoints

To set a breakpoint in a section, move the mouse to the breakpoint column and click in any row that has a breakpoint candidate marker. See Figure C-8. (Notice that the cursor has changed to the breakpoint cursor.) To remove a breakpoint, simply click the breakpoint indicator. To remove all breakpoints use the Control menu's Clear All Breakpoints command.

Figure C-8   Setting breakpoints

Breakpoint candidate

Breakpoint indicator
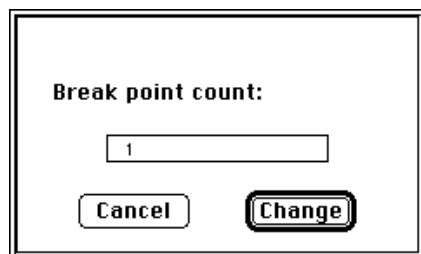
Note
You cannot set or clear a breakpoint while the DSP is running. ◆

Snoopy can also set a breakpoint to the *n*th occurrence of the instruction. To set a multiple breakpoint the breakpoint must already have been set. Use the Option key and click the breakpoint indicator to set the number of times the breakpoint instruction is to be executed before stopping the DSP. A dialog box will appear allowing you to change the pass counter on the specified breakpoint. See Figure C-9. The initial number is always one. If the breakpoint counter is set to four then the breakpoint instruction will execute three times and stop the DSP on the fourth occurrence of the instruction.

Figure C-9   Setting the breakpoint counter

Break point count:

1

Cancel   Change

Breakpoint Restrictions

Because the AT&T DSP3210 is a pipelined device, and because it has minimal provisions for debugging, there are restrictions as to where you can set breakpoints. You can only set a breakpoint at a location that has a breakpoint candidate marker.

Single Stepping

To step to the next available instruction (not necessarily the next instruction), select
Single Step from the Control menu.
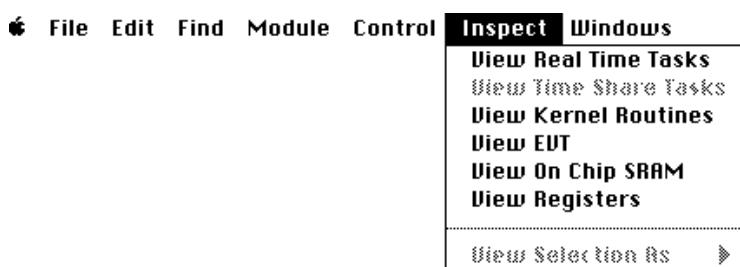
Note
Single Stepping is implemented using breakpoints (remember, no trace).
Because of this, Snoopy can only step within a section or DSP operating
system routine. ◆

Inspect Menu
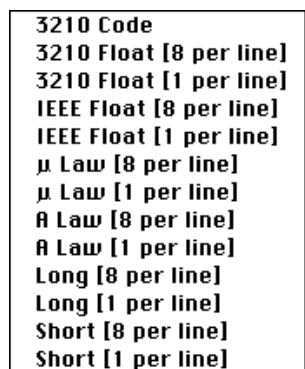
The Inspect menu provides access to additional display windows. These windows are
available to view real-time tasks, timeshare tasks, DSP operating system routines, the
EVT, the on-chip SRAM, and the registers. See Figure C-10.

Figure C-10   Inspect menu

Additionally, the Inspect menu has the View Selection As submenu. This menu is used to
select the different data formats for viewing purposes only. See Figure C-11. Changes
made in the View Selection As menu do not effect the actual data type. To coerce data
into another data type see the "Editing Data" section.

Figure C-11   Data display format menu
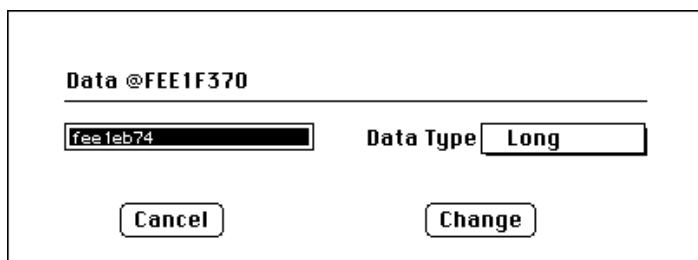
Formatting

Data can be displayed in several formats. To change the data format, highlight the display lines you wish to change and select a format from the View Selection As menu accessed through the Inspect menu. To highlight multiple lines hold down the shift key while dragging the mouse.

Editing Data

Editing data is a point-and-click operation. Simply point to the data element you wish to edit in the data display window and double-click. The dialog box shown in Figure C-12 will appear, allowing you to view and edit the data and the data type.

Figure C-12   Data editing window
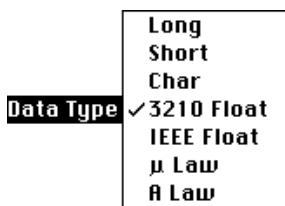
Note
Data viewed as type DSP Code cannot be edited. ◆

▲   **WARNING**
DSP Code can be viewed as some other data type and the editor will allow it to be changed. Extreme caution should be used when changing DSP Code. There are no safeguards to prevent illegal opcodes from being entered. This could result in loss of data or code. ▲

The defined data types that can be selected are shown in Figure C-13. Changes in the data type will coerce the data into the new data type. Use the View Selection As menu if you only want to view the data in a different format.

Figure C-13   Defined data types

Windows Menu

When you launch the Snoopy application, you are presented with the Real-Time Tasks window and the Current PC window. As shown in Figure C-14, the Windows menu lists all open windows at the bottom of the menu and provides a quick way to bring any window to the front. Auto Hide Windows (not implemented) removes the current window when a new window is selected. The Windows menu also provides limited control over open windows.

Figure C-14   Windows menu

Module   Control   Inspect   **Windows**
Update Front Window   ⌘U
Full Titles
Hide DSP Control
Auto Hide Windows

✓**Current PC**
Real Time Tasks

When the DSP is running the frontmost window is not automatically updated. The window must be told to update or the data display will show only old data. This can be done using the Update Front Window command in the Windows menu. Windows such as Registers, Real-Time Tasks, and Timeshare Tasks will not have their data display window updated if the DSP is running.

Windows can also be set to show the full title using the Full Titles command. This includes information describing the specific DSP chip that the module is running on. Also, the DSP Control window can be hidden or displayed from the Windows menu.

Additional Information Windows

This section describes the additional display windows used to provide information about other parts of the DSP. The additional windows provide information about the inside operation of the DSP. There are five additional windows:

■ Current PC

■ Kernel Routines

■ EVT (Exception Vector Table)

■ On-Chip SRAM

■ Registers (in the DSP)

Current PC

The Current PC window shows the section or DSP operating system routine that was running on the DSP when a break was initiated. This window can only show data when the DSP has been stopped. If the DSP has been stopped with no breakpoint set it will always stop at `ExternalIntOne` of the DSP operating system routines. Figure C-15 shows the current program counter when a breakpoint is set in the Standard Sound Input Task:Input Module:Program Section at address `5003E020`. The current PC location has been single stepped two times. Notice that the address `5003E014` has no breakpoint allowed. It will also be stepped over by the single-step procedure.

Figure C-15   Current PC window

The DSP Operating System Routines

DSP operating system routines are shown in the Kernel Routines window; simply select View Kernel Routines from the Inspect menu. The DSP operating system routines will be presented in a browser window similar to the Real-Time Tasks window with the exception of the list at the top. This window is shown in Figure C-16. You can manipulate DSP operating system routines (set breakpoints, single-step, and reformat) the same way you manipulate sections.

Figure C-16   DSP Operating System Routines window

The EVT

The DSP operating system places system information, such as run-time variables and routine addresses in the exception vector table (EVT). To view the EVT window, select View EVT from the Inspect menu. The resulting window is shown in Figure C-17.

Figure C-17   EVT window
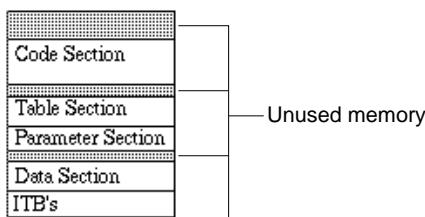
On-Chip SRAM

To view the On-Chip SRAM window, select View On Chip SRAM from the Inspect menu. The resulting window is shown in Figure C-18.

Figure C-18   On-Chip SRAM window

When the DSP has been stopped, all sections that are currently in it's cache can be viewed in the On-Chip SRAM window. The data displayed is an image of the cached code, buffers, tables, and other types of sections. Figure C-19 shows a possible SRAM layout.

Figure C-19   Example of SRAM layout

Registers

To display the DSP registers, select Show Registers from the Edit menu. The resulting window is shown in Figure C-20. Notice that the processor status word is displayed as a group of check boxes in the upper right corner.

Figure C-20   Registers window
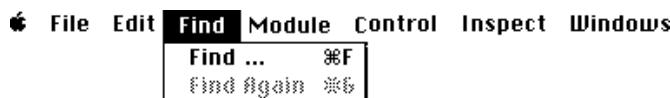
The Registers window displays the contents of all of the DSP registers for the instruction at the current PC location. The Registers window is not updated while the DSP is running. Manual update of the frontmost window is explained in "Windows Menu," earlier in this appendix.

Standard Menus

Snoopy uses the two menus that resemble standard Finder menus: File and Edit. The File menu is used for opening, closing, and saving files. It uses the standard dialog box for all operations. The Edit menu operates like the standard Finder Edit menu.

Find Menu
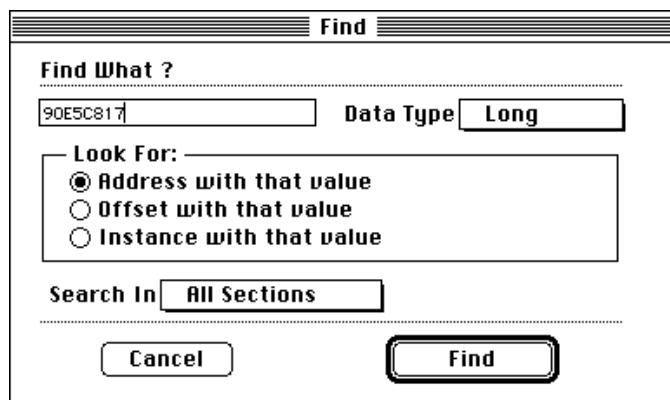
There are two commands in the Find menu. The Find command is used to locate specific strings within the currently selected window. The Find Again command finds subsequent occurrences of the specified information. See Figure C-21.

Figure C-21   Find menu

When the Find command is selected the dialog box in Figure C-22 will appear. The Find command can locate specific data values that are of a specific data type. If the specified value is in the display range of any of the windows, the required window will be selected and the address will be shown. The required window does not need to be the frontmost window or even open for the Find operation to select it.

Figure C-22   Find Command dialog box

There are three ways to look for specific locations in the data display. They are:

■ look for an address with the specified value

■ look for an offset with the specified value

■ look for an instance with the specified value

When trying to locate a specific address location in the Data Display window use the Look For Address with that value. When looking for a relative address within the data display use the Look For Offset with that value. To look for a specific data word within the data window use the Look For Instance with that value.

The default data type is a Long word. This should be used whenever specifying an offset value. When looking for an Instance the data type may also be specified, as shown in Figure C-23.
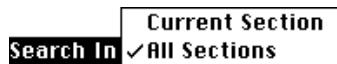
Figure C-23   Find Data Types menu

Snoopy User's Guide

You may also specify the section or sections to be searched, as shown in Figure C-24.

Figure C-24   Search In selection menu

If the data with the specified data type cannot be located, an alert box will display the message Not Found.

Module Menu

Snoopy has some symbolic capabilities. If you declare symbols as global (using the `.global` assembler construct) a symbol table is created with your object code. To access these symbols with Snoopy, choose the Show Symbols For ... command (replacing the ellipsis with the name of the currently selected module) from the Module menu and locate its object file. The currently selected module must be in either the Real-Time Tasks or Timeshare Tasks window. Figure C-25 shows that the module named Input was selected in the Real-Time Tasks window. A standard open-file dialog box is displayed for selecting the appropriate resource file.

Figure C-25   Module menu

To remove the symbols from the Data Display window select the Remove Symbols From Input command. This will remove the symbol lookup information from Snoopy's memory. This command cannot be undone. The symbol file must be reloaded to show symbols.

There are several error messages which may be encountered when attempting to load a symbolic file. For example, if the module symbol resource file cannot be loaded because it does not match the code resources available in the loaded module then the error window shown in Figure C-26 will appear.

Snoopy User's Guide

Figure C-26   Error in loading symbolic table

Mechanical Details

This appendix provides details of the mechanical provisions for mounting internal SCSI devices in the Macintosh Quadra 840AV and Macintosh Centris 660AV enclosures and for installing accessory cards in the Macintosh Centris 660AV. It is intended to guide hardware engineers developing compatible equipment. The mechanical details for internal SCSI device mounting consist of the following seven foldout drawings:

■ Figure D-1 reproduces Apple drawing number 815-1411-A, showing the bezel for the Macintosh Centris 660AV enclosure used with an internal CD-ROM drive.

■ Figure D-2 reproduces Apple drawing number 815-1376-A, showing the blank bezel for the Macintosh Centris 660AV enclosure (without an internal CD-ROM drive).

■ Figure D-3 reproduces Apple drawing number 815-1122-03, showing the mounting sled used for internal 5.25-inch hard disk drives.

■ Figure D-4 reproduces Apple drawing number 805-0503-01, showing the magnetic shield for the Macintosh Centris 660AV bezel used with an internal CD-ROM drive.

■ Figure D-5 reproduces Apple drawing number 805-0517-02, showing the magnetic shield for the Macintosh Quadra 840AV bezel used with internal CD-ROM or hard disk drives.

■ Figure D-6 reproduces Apple drawing number 815-1189-05, showing the blank bezel for the Macintosh Quadra 840AV enclosure (without an internal CD-ROM drive).

■ Figure D-7 reproduces Apple drawing number 815-1186-04, showing the bezel for the Macintosh Quadra 840AV enclosure used with an internal CD-ROM drive.

The following four foldout drawings give mechanical details for mounting expansion cards in the Macintosh Centris 660AV:

■ Figure D-8 reproduces Apple drawing number 805-0530-06, showing the bracket in the Macintosh Centris 660AV that supports an expansion card.

■ Figure D-9 reproduces Apple drawing number 725-0051-02, showing the insulator for the expansion card bracket.

■ Figure D-10 reproduces Apple drawing number 630-0450-10, showing the electromagnetic interference (EMI) shield for the expansion card bracket.

■ Figure D-11 reproduces Apple drawing number 630-0450-12, showing the NuBus adapter card for the Macintosh Centris 660AV. This card is discussed in "Slot Connections," in Chapter 2.

For details of expansion card mounting in the Macintosh Quadra 840AV, see *Designing Cards and Drivers for the Macintosh Family,* third edition.

NOTES :
(UNLESS OTHERWISE SPECIFIED)

1 INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982

2 MATERIAL : ABS CYCOLAC KJC 34187-%. COLOR : APPLE PLATINUM
   PER COLOR APPLE COLOR CONTROL PANEL 912-0037. TOLERANCE PER
   COLOR TOLERANCE SET 912-1037.

3 ALL UNSPECIFIED DRAFT ANGLES TO BE 0° 30'.

4 ALL NON-APPEARANCE SURFACE EDGES TO HAVE
   A 0.25 ±0.03 RADIUS. EXCEPT AT PARTING LINE.

5 FLAT SURFACES TO HAVE A FLATNESS TOLERANCE OF 0.20 PER 25.00,
   NOT TO EXCEED 1.00 OVER THE ENTIRE SURFACE

6 STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.05 PER 25.00,
   NOT TO EXCEED 0.20 OVER ENTIRE LENGTH

7 EXTERIOR (APPEARANCE) SURFACES TO BE TEXTURED
   PER APPLE SPECIFICATION 062-0222 (G-2).

8 MOLD FINISH ON INTERIOR (NON-APPEARANCE) SURFACES
   TO BE SPI-SPE 3.

9 FLASH NOT TO EXCEED 0.13

10 GATE TRIM TO BE 0.5 MAX.

11 PARTING LINE MISMATCH NOT TO EXCEED 0.10

12 NO SINK DEPRESSIONS TO BE VISIBLE ON APPEARANCE SURFACES

13 APPEARANCE SURFACES TO BE FREE OF COSMETIC DEFECTS INCLUDING,
   BUT NOT LIMITED TO, SPLAY, INCLUDED PARTICLES, BURNED PLASTIC
   MARKS AND SIMILAR IMPERFECTIONS. SEE APPLE SPEC. 062-2006.

14 PART TO BE FREE OF MOLD RELEASE ON APPEARANCE SURFACES.

15 MOLD DESIGN TO MINIMIZE GATE BLUSH, FLOW LINES, AND MOLD MARKS.
   MOLD CONSTRUCTION TO CONFORM TO GOOD MOLDING INDUSTRY PRACTICE
   AS STATED IN THE CURRENT EDITION OF 'STANDARDS AND PRACTICES OF
   PLASTIC CUSTOM MOLDERS' BY THE SOCIETY OF THE PLASTIC INDUSTRY, INC.

16 EJECTOR PIN, PARTING LINE, AND GATE LOCATION MUST BE APPROVED BY
   APPLE COMPUTER PRODUCT DESIGN ENGINEERING PRIOR TO MOLD FABRICATION.

17 MOLD TO BE PROPERTY OF APPLE COMPUTER, INC. AND SHALL BE
   MARKED WITH APPLE'S NAME, APPROPRIATE PART NUMBER AND DATE.

18 MARK APPLE PART NUMBER AND REVISION LETTER WITH 3.0 HIGH
   CHARACTERS APPX. 0.3 TALL, NOT TO EXCEED 0.5, IN LOCATION SHOWN.

19 TOLERANCE :

DIM.	TOL.
0 - <50	+/- 0.13
50 - <150	+/- 0.20
150 - <300	+/- 0.33
>300	+/- 0.50

20 NOMINAL WALL THICKNESS 3

21 MARK MATERIAL WITH RECYCLING TRIANGLE AND MATERIAL IDENTIFIER APPX WHERE SHOWN. IDENTIFIER
   "ABS-FR" TO BE MARKED ON INTERIOR NON-FUNCTIONAL (NON-APPEARANCE) SURFACE USING 3.0 HIGH
   CHARACTERS, 2.0 BELOW ISOCELES TRIANGLE (10.0) TALL TO POINT OF INTERSECTION) WITH 1.0
   RADII AT CORNERS. MATERIAL CALLOUT "7" IN 5.0 HIGH CHARACTERS TO BE APPOX CENTERED IN
   TRIANGLE. ALL CHARACTERS TO BE RAISED 0.3, BUT NOT EXCEED 0.5 FROM SURFACE. SEE DWG BELOW.

22 THIS IS A SUPPLEMENTAL CRITICAL FUNCTION DRAWING AND IS TO
   BE USED IN CONJUNCTION WITH FILE 815-1411-03wl.igs. 815-1411-03a.igs
   TO MANUFACTURE AND TO INSPECT THE PART.

23 STARRED (*) DIMENSIONS ARE FOR INCOMING QUALITY CONTROL INSPECTION.

RECYCLING SYMBOL
SCALE 3:1

ABS - FR

SECTION A—A
SNAP FEATURES

FIGURE D-1

CD bezel for Macintosh Centris 660AV

815-1411-A

NOTES :

(UNLESS OTHERWISE SPECIFIED)

1   INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982

2   MATERIAL : ABS CYCOLAC KJC 34187-1. COLOR : APPLE PLATINUM
    PER COLOR APPLE COLOR CONTROL PANEL 912-0037. TOLERANCE PER
    COLOR TOLERANCE SET 912-1037.

3   ALL UNSPECIFIED DRAFT ANGLES TO BE 0° 30'.

4   ALL NON-APPEARANCE SURFACE EDGES TO HAVE
    A 0.25 ±0.03 RADIUS. EXCEPT AT PARTING LINE.

5   FLAT SURFACES TO HAVE A FLATNESS TOLERANCE OF 0.20 PER 25.00,
    NOT TO EXCEED 1.00 OVER THE ENTIRE SURFACE.

6   STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.05 PER 25.00,
    NOT TO EXCEED 0.20 OVER ENTIRE LENGTH

7   EXTERIOR (APPEARANCE) SURFACES TO BE TEXTURED
    PER APPLE SPECIFICATION 062-0222 (G-2).

8   MOLD FINISH ON INTERIOR (NON-APPEARANCE) SURFACES
    TO BE SPI-SPE 3.

9   FLASH NOT TO EXCEED 0.13

10 GATE TRIM TO BE 0.5 MAX.

11 PARTING LINE MISMATCH NOT TO EXCEED 0.10

12 NO SINK DEPRESSIONS TO BE VISIBLE ON APPEARANCE SURFACES

13 APPEARANCE SURFACES TO BE FREE OF COSMETIC DEFECTS INCLUDING,
    BUT NOT LIMITED TO, SPLAY INCLUDED PARTICLES, BURNED PLASTIC
    MARKS AND SIMILAR IMPERFECTIONS. SEE APPLE SPEC. 062-2006.

14 PART TO BE FREE OF MOLD RELEASE ON APPEARANCE SURFACES.

15 MOLD DESIGN TO MINIMIZE GATE BLUSH, FLOW LINES, AND MOLD MARKS.
    MOLD CONSTRUCTION TO CONFORM TO GOOD MOLDING INDUSTRY PRACTICE
    AS STATED IN THE CURRENT EDITION OF "STANDARDS AND PRACTICES OF
    PLASTIC CUSTOM MOLDERS" BY THE SOCIETY OF THE PLASTIC INDUSTRY, INC.

16 EJECTOR PIN, PARTING LINE, AND GATE LOCATION MUST BE APPROVED BY
    APPLE COMPUTER PRODUCT DESIGN ENGINEERING PRIOR TO MOLD FABRICATION.

17 MOLD TO BE PROPERTY OF APPLE COMPUTER, INC. AND SHALL BE
    MARKED WITH APPLE'S NAME, APPROPRIATE PART NUMBER AND DATE.

18 MARK APPLE PART NUMBER AND REVISION LETTER WITH 3.0 HIGH
    CHARACTERS APPX. 0.3 TALL, NOT TO EXCEED 0.5, IN LOCATION SHOWN.

19 TOLERANCE :

DIM.	TOL.
0 - <50	+/- 0.13
50 - <150	+/- 0.20
150 - <300	+/- 0.33
>300	+/- 0.50

20 NOMINAL WALL THICKNESS 3

21 MARK MATERIAL WITH RECYCLING TRIANGLE AND MATERIAL IDENTIFIER APPX WHERE SHOWN. IDENTIFIER
    "ABS-FR" TO BE MARKED ON INTERIOR NON-FUNCTIONAL (NON-APPEARANCE) SURFACE USING 3.0 HIGH
    CHARACTERS, 2.0 BELOW ISOCELES TRIANGLE (10.0) TALL TO POINT OF INTERSECTION) WITH 1.0
    RADII AT CORNERS. MATERIAL CALLOUT "7" IN 5.0 HIGH CHARACTERS TO BE APPOX CENTERED IN
    TRIANGLE. ALL CHARACTERS TO BE RAISED 0.3, BUT NOT EXCEED 0.5 FROM SURFACE. SEE DWG BELOW.

22 THIS IS A SUPPLEMENTAL CRITICAL FUNCTION DRAWING AND IS TO
    BE USED IN CONJUNCTION WITH FILE 815-1376-04w.igs & 815-1376-04a.igs
    TO MANUFACTURE AND TO INSPECT THE PART.

23 STARRED (*) DIMENSIONS ARE FOR INCOMING QUALITY CONTROL INSPECTION.

1.57 2X
*(1.54)
8.07
(2°)
R 2157.81
0.38
167.73
SECTION A—A
SNAP DETAILS

85.90
* 29.60 SYM
ABS - FR
R 2.0
3.5° 2X

61.24 - DRFT
SYM
4.46 + DRFT

162.90 SYM
(63.90)

60° 3X
R1 3X
5
10
3   ABS - FR   2
RECYCLING SYMBOL
SCALE 3:1
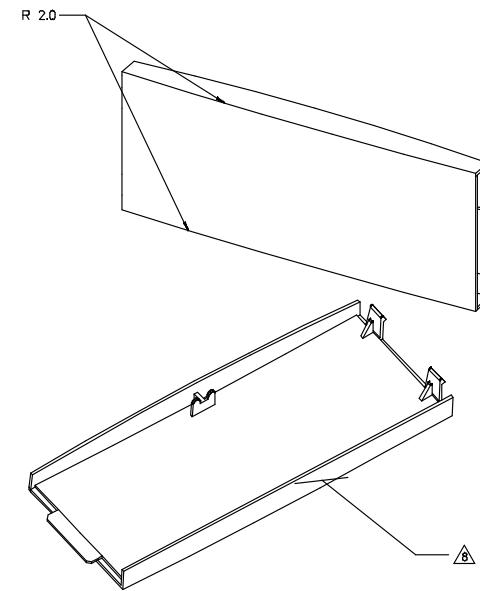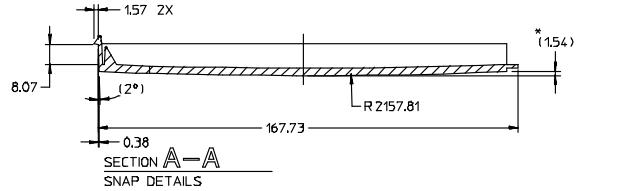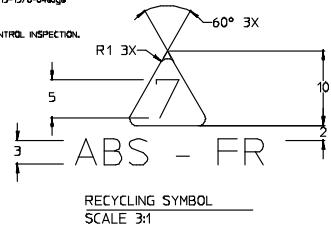
FIGURE D-2

Blank bezel for the Macintosh Centris 660AV

815-1376-A

NOTES: (UNLESS OTHERISE SPECIFIED)

1. INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982.

2. STARRED (*) DIMENSIONS INDICATE PROCESS CONTROL DIMENSIONS.

3. MATERIAL: G.E. CYCLOAC KJC #34187-Y.
   COLOR: APPLE PLATINUM PER COLOR CONTROL PANEL 912-0037,
   TOLERANCE PER COLOR TOLERANCE SET 912-1037.

4. NOMINAL WALL THICKNESS TO BE 2.00.

5. UNSPECIFIED DRAFT ANGLES TO BE 1°.

6. ALL EDGES TO BE 0.25 ±0.12 RADIUS, EXCEPT AT PARTING LINE.

7. GATE TRIM TO BE 0.3 MAX. FLUSH WITH SURFACE.

8. FLASH NOT TO EXCEED 0.13.

9. PARTING LINE MISMATCH NOT TO EXCEED 0.25.

10. STRAIGHTEDGES TO HAVE STRAIGHTNESS TOLERANCE OF 0.07
    PER 25.0 NOT TO EXCEED 0.5 OVER THE ENTIRE SURFACE.

11. FLAT SURFACES TO HAVE FLATNESS TOLERANCE OF 0.10 PER
    25.0, NOT TO EXCEED 1.0 OVER THE ENTIRE SURFACE.

12. MARK APPLE PART NUMBER AND REVISION LETTER WITH 2.0
    MINIMUM HIGH CHARACTERS APPROXIMATELY WHERE SHOWN.
    REVISION LETTER TO BE LOCATED ON AN EJECTOR PIN.

13. MOLD FINISH ON ALL SURFACES TO BE SPI-SPE #3.

14. MOLD DESIGN TO MINIMIZE EJECTION PIN MARKS, GATE BLUSH,
    LINES AND WELD MARKS. MOLD CONSTRUCTION TO CONFORM TO
    GOOD MOLDING INDUSTRY PRACTICE AS STATED IN THE CURRENT
    EDITION OF "STANDARD PRACTICE OF CUSTOM MOLDERS" BY THE
    SOCIETY OF THE PLASTICS INDUSTRY, INC..

15. EJECTOR PIN, PARTING LINE, AND GATE LOCATION MUST BE
    APPROVED BY APPLE COMPUTER PRODUCT DESIGN ENGINEERING
    PRIOR TO MOLD FABRICATION.

16. MOLD TO BE PROPERTY OF APPLE COMPUTER INC, AND SHALL BE
    PERMANENTLY MARKED WITH APPLE'S NAME AND APPROPRIATE
    PART NUMBER.

SECTION B-B

SECTION A-A

SECTION E-E 2X
SCALE: 2/1

SECTION D-D 2X

SECTION C-C
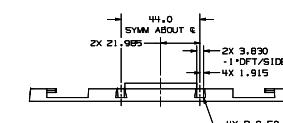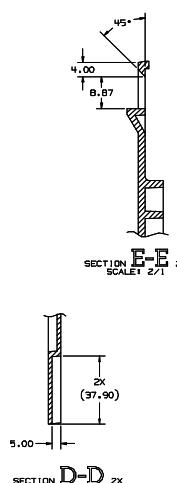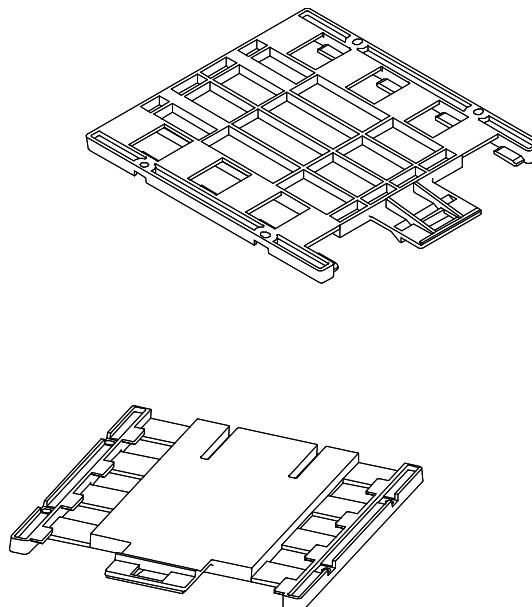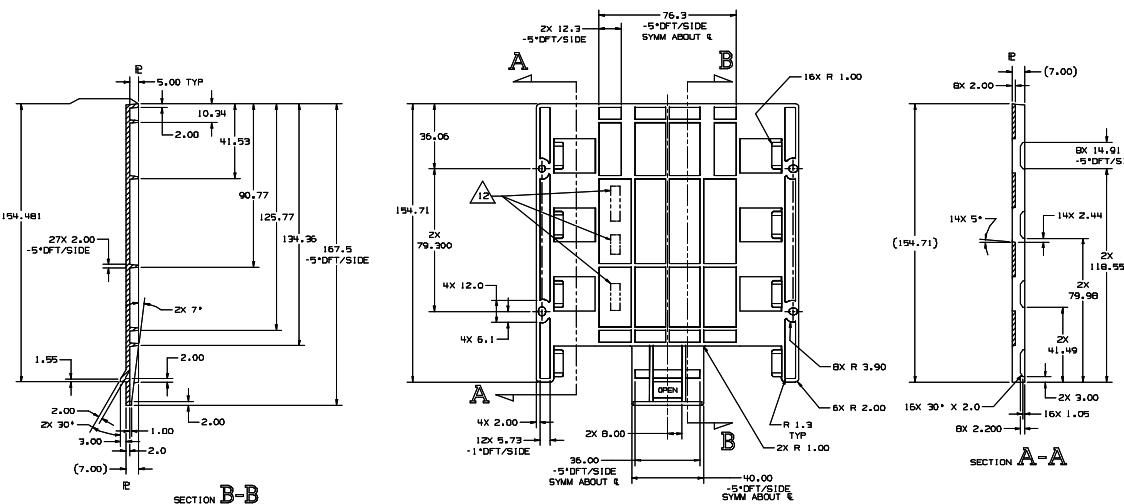
FIGURE D-3

Mounting sled for 5.25-inch SCSI devices

815-1122-03

NOTES (UNLESS SPECIFIED OTHERWISE):

1 MATERIAL: 0.25 (.010") THK DR9 (TIN PRE-PLATED CRS).

2 INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982.

3 ALL DIMENSIONS ARE INCHES AND MILLIMETERS: USE METRIC TOLERANCE BLOCK.

4 ALL UNSPECIFIED INSIDE BEND RADII TO BE 0.3mm.

5 ALL UNSPECIFIED RIGHT ANGLE BENDS TO BE 90°±0.5°.

6 ALL SHARP CORNERS TO BE BROKEN 0.8 X 45°.

7 STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.15mm PER 25.00mm,
   NOT TO EXCEED 0.5mm OVER THE ENTIRE LENGTH.

8 FLAT SURFACES TO HAVE A FLATNESS TOLERANCE OF 0.10mm PER 25.00mm, NOT
   TO EXCEED 0.50mm OVER THE ENTIRE SURFACE.

9 SURFACES TO BE FREE OF CONTAMINATES, METAL FLAKES, AND OIL.

10 MARK APPLE PART NUMBER AND REVISIONS LETTER WITH 3.0mm MINIMUM HIGH
   CHARACTERS APPROXIMATELY WHERE INDICATED.

11 STEEL RULE DIES, SPECIAL PUNCHES, OR OTHER UNIQUE TOOLING REQUIRED TO
   FABRICATE THIS PART TO BE PROPERTY OF APPLE COMPUTER INC. AND SHALL BE
   PERMANENTLY MARKED WITH APPLE'S NAME AND APPROPRIATE APPLE PART NUMBER.

12 MAXIMUM BURR ALLOWANCE IS 15% OF MATERIAL THICKNESS.

13 THIS IS A SUPPLEMENTAL DETAIL DRAWING AND IS TO BE USED IN CONJUNCTION
   WITH PART MODEL FILE shld300.igs TO MANUFACTURE AND TO INSPECT THE PART.

SEC A–A

SEC B–B
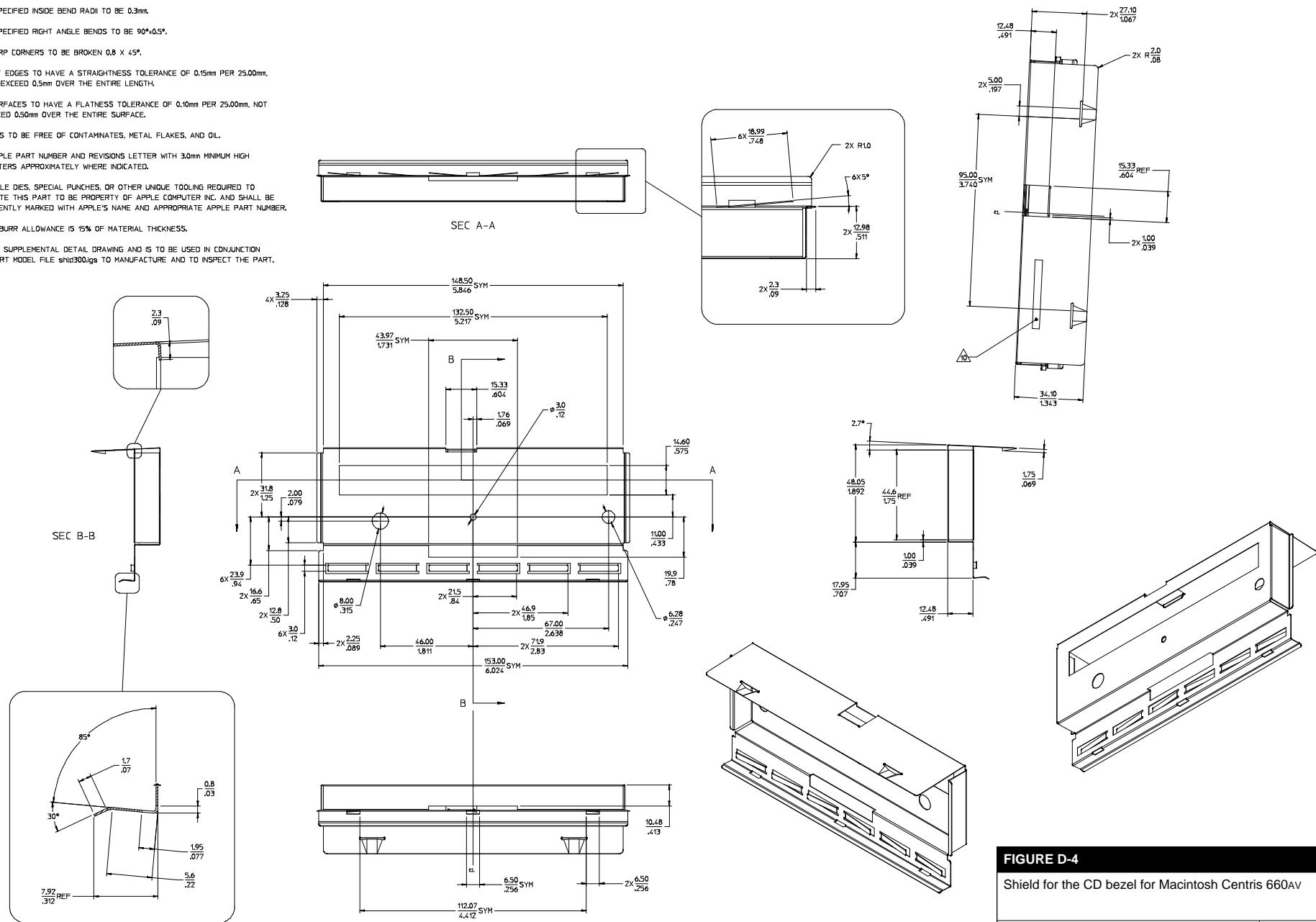
FIGURE D-4

Shield for the CD bezel for Macintosh Centris 660AV

805-0503-01

NOTES: UNLESS OTHERWISE SPECIFIED

1. INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982.

2. MATERIAL: 0.28 (.011") SUPER-ORTHOSIL-4 COATED WITH CARLITE OVER GLASS OR ENGINEERING APPROVED EQUIVALENT.

3. STARRED (*) DIMENSIONS ARE CONTROL DIMENSIONS.

4. ALL UNSPECIFIED INSIDE BEND RADII TO BE 0.3 .

5. ALL UNSPECIFIED RIGHT ANGLE BENDS TO BE 90° ±0.5 .

6. ALL SHARP CORNERS TO BE BROKEN 0.8 X 45°.

7. STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.15 PER 25.00, NOT TO EXCEED 0.50 OVER THE ENTIRE LENGTH.

8. FLAT SURFACES TO HAVE A FLATNESS TOLERANCE OF 0.10 PER 25.00, NOT TO EXCEED 0.50 OVER THE ENTIRE SURFACE.

9. SURFACES TO BE FREE OF CONTAMINATES, METAL FLAKES, AND OIL.

10. MARK APPLE PART NUMBER AND REVISION LETTER WITH 3.0 MINIMUM HIGH CHARACTERS APPROXIMATELY WHERE INDICATED.

11. STEEL RULE DIES, SPECIAL PUNCHES OR OTHER UNIQUE TOOLING REQUIRES TO MAKE THIS PART TO BE PROPERTY OF APPLE COMPUTER, INC., AND SHALL BE PERMANENTLY MARKED WITH APPLE'S NAME AND APPROPRIATE APPLE PART NUMBER.

12. ADHESIVE: SCOTCH 3M P/N 9500 OR ENGINEERING APPROVED EQUIVALENT.

13. ADHESIVE TO BE APPLIED TO SURFACE AS INDICATED AND MUST BE COVERED WITH A REMOVABLE PAPER ELEMENT.
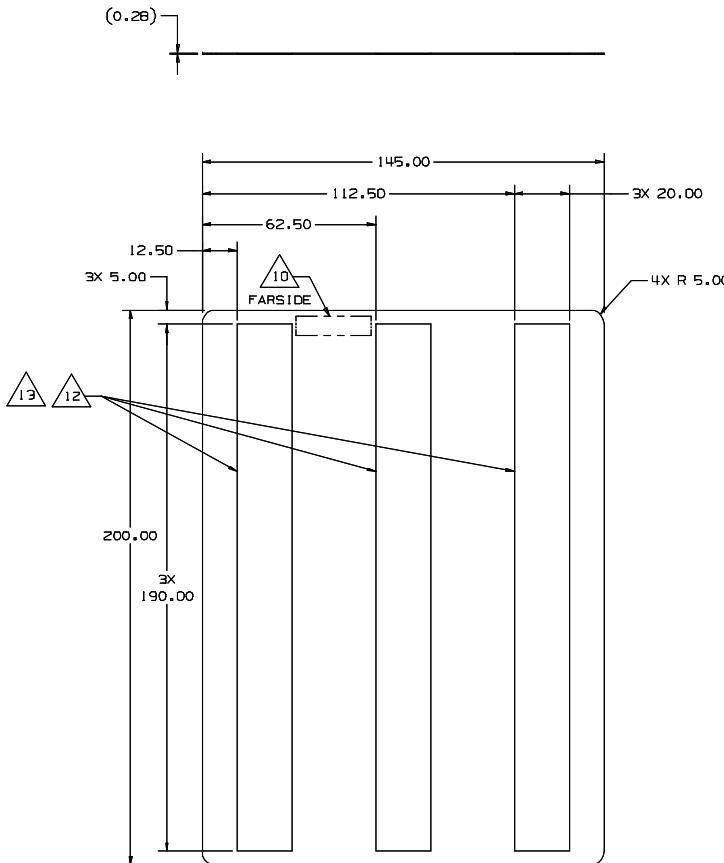
FIGURE D-5
Magnetic shield for CD-ROM drives
805-0517-02

NOTE: UNLESS OTHERWISE SPECIFIED

1. INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982.

2. ALL DIMENSIONS ARE IN MILLIMETERS; REFER TO METRIC TOLERANCE BLOCK.

3. MATERIAL: ABS CYCOLAC KJC 34187-1, COLOR: APPLE PLATINUM PER COLOR CONTROL PANEL 912-0037, TOL PER COLOR TOL SET 912-1037.

4. WALL THICKNESS: 3.00.

5. DRAFT ANGLES: EXTERIOR SURFACES TO BE 0° PER SIDE.
   INTERIOR SURFACES TO BE 1° PER SIDE.

6. ALL EXTERIOR (APPEARANCE SURFACE) EDGES TO BE 0.15 RADIUS, EXCEPT AT PARTING LINE.

7. ALL INTERIOR (NON-APPEARANCE SURFACE) INSIDE RADII (90° CORNERS) TO BE 0.15 OUTSIDE RADII (270° CORNERS) TO BE 0.50, EXCEPT AT PARTING LINE.

8. STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.15 PER 25mm, NOT TO EXCEED 0.5 OVER THE ENTIRE LENGTH.

9. FLAT SURFACES TO HAVE FLATNESS TOLERANCE OF 0.15 PER 25mm, NOT TO EXCEED 0.5 OVER THE ENTIRE SURFACE.

10. APPEARANCE SURFACES ARE THOSE SURFACES TEXTURED PER APPLE SPEC 062-0222, PER NOTES 11, 12 & 14.

11. ALL EXTERIOR (APPEARANCE) SURFACES TO BE RE6642 G2 TEXTURE PER APPLE SPECIFICATION 062-0222 WHERE INDICATED.

12. TEXTURE TO BE SPI-SPE#3 (APPLE SPEC 062-0222) WHERE INDICATED.

13. MOLD FINISH ON INTERIOR (NON-APPEARANCE) SURFACES TO BE SPI-SPE#3.

14. TEXTURE TO BE RE6643 G-3 PER APPLE SPECIFICATION, WHERE INDICATED.

15. GATE TO BE MACHINE TRIMMED 0.15 BELOW SURFACE.

16. REFER TO APPLE SPEC 062-2006 FOR COSMETIC ACCEPTANCE CRITERIA.

17. FLASH NOT TO EXCEED 0.05.

18. PART TO BE FREE OF MOLD RELEASE ON APPEARANCE SIDE OF PART.

19. MOLD DESIGN TO MINIMIZE EJECTION PIN MARKS, GATE BLUSH, LINES AND WELD MARKS. MOLD CONSTRUCTION TO CONFORM TO GOOD MOLDING INDUSTRY PRACTICE AS STATED IN THE CURRENT EDITION OF "STANDARD PRACTICES OF CUSTOM MOLDERS" BY THE SOCIETY OF PLASTIC INDUSTRY, INC.

20. MARK APPLE PART NUMBER AND REVISION LETTER WITH 3.0 MINIMUM HIGH CHARACTERS APPROXIMATELY WHERE SHOWN.

21. EJECTOR PIN, PARTING LINE, AND GATE LOCATIONS MUST BE APPROVED BY APPLE COMPUTER PRODUCT DESIGN ENGINEERING PRIOR TO MOLD FABRICATION.

22. MOLD TO BE PROPERTY OF APPLE COMPUTER, INC. AND SHALL BE PERMANENTLY MARKED WITH APPLE'S NAME AND APPROPRIATE TOOL NUMBER.

23. STARRED (*) DIMENSIONS AND NOTES ARE CRITICAL CONTROL DIMENSIONS AND NOTES FOR QC INSPECTION.

24. THIS DRAWING CONTAINS CRITICAL TO FUNCTION DIMENSIONS ONLY. REFER TO UNIGRAPHICS CAD FILE (815-1189-04) FOR COMPLETE PART INFO.
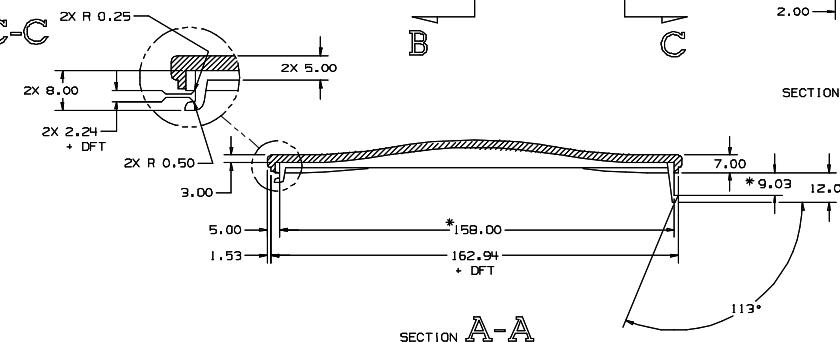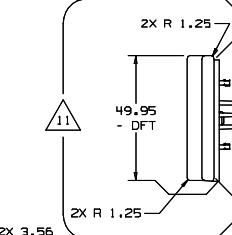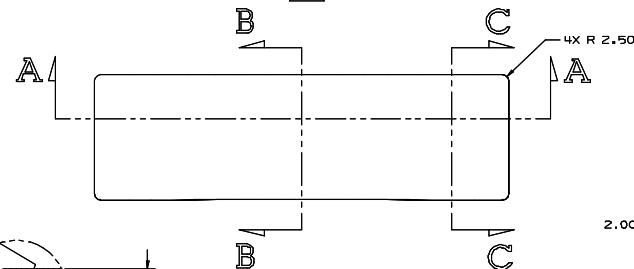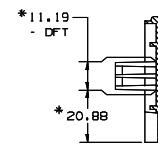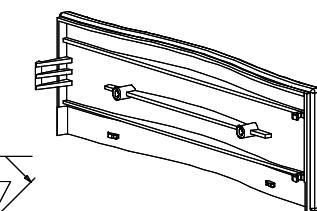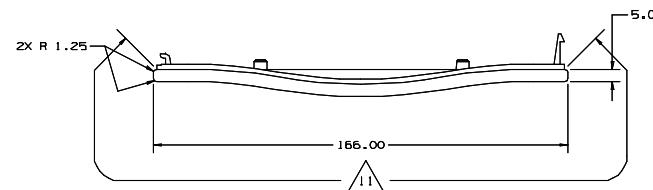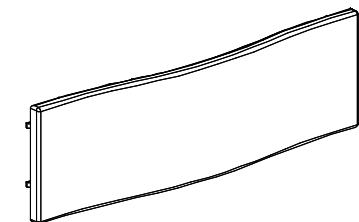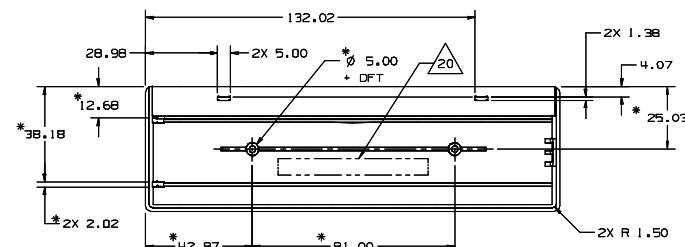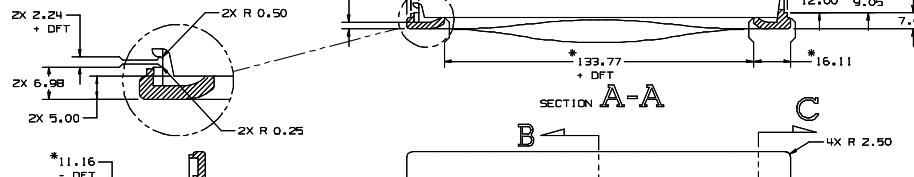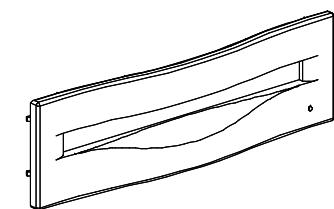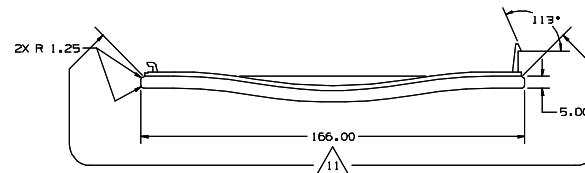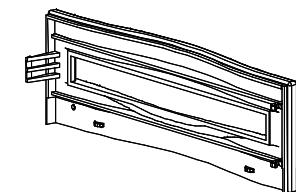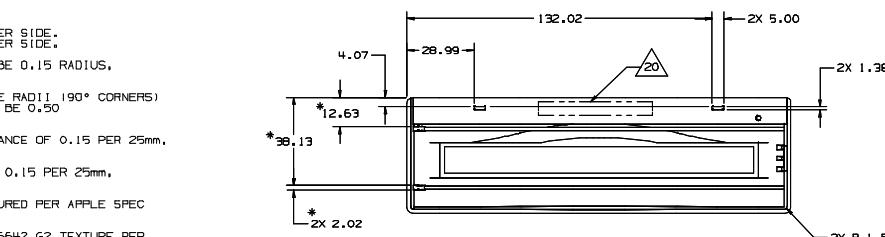
FIGURE D-6

Blank bezel for the Macintosh Quadra 840AV

815-1189-05

1. INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5M-1982.

2. ALL DIMENSIONS ARE IN MILLIMETERS; REFER TO METRIC TOLERANCE BLOCK.

3. MATERIAL: ABS CYCOLAC KJC 34187-1, COLOR: APPLE PLATINUM PER COLOR CONTROL PANEL 912-0037, TOL PER COLOR TOL SET 912-1037.

4. WALL THICKNESS: 3.00.

5. DRAFT ANGLES: EXTERIOR SURFACES TO BE 0° PER SIDE.
   INTERIOR SURFACES TO BE 1° PER SIDE.

6. ALL EXTERIOR (APPEARANCE SURFACE) EDGES TO BE 0.15 RADIUS, EXCEPT AT PARTING LINE.

7. ALL INTERIOR (NON-APPEARANCE SURFACE) INSIDE RADII (90° CORNERS) TO BE 0.15 OUTSIDE RADII, (270° CORNERS) TO BE 0.50 EXCEPT AT PARTING LINE.

8. STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.15 PER 25mm, NOT TO EXCEED 0.5 OVER THE ENTIRE LENGTH.

9. FLAT SURFACES TO HAVE FLATNESS TOLERANCE OF 0.15 PER 25mm, NOT TO EXCEED 0.5 OVER THE ENTIRE SURFACE.

10. APPEARANCE SURFACES ARE THOSE SURFACES TEXTURED PER APPLE SPEC 062-0222, PER NOTES 11, 12 & 14.

11. ALL EXTERIOR (APPEARANCE) SURFACES TO BE RE6642 G2 TEXTURE PER APPLE SPECIFICATION 062-0222 WHERE INDICATED.

12. TEXTURE TO BE SPI-SPE#3 (APPLE SPEC 062-0222) WHERE INDICATED.

13. MOLD FINISH ON INTERIOR (NON-APPEARANCE) SURFACES TO BE SPI-SPE#3.

14. TEXTURE TO BE RE6643 G-3 PER APPLE SPECIFICATION, WHERE INDICATED.

15. GATE TO BE MACHINE TRIMMED 0.15 BELOW SURFACE.

16. REFER TO APPLE SPEC 062-2006 FOR COSMETIC ACCEPTANCE CRITERIA.

17. FLASH NOT TO EXCEED 0.05.

18. PART TO BE FREE OF MOLD RELEASE ON APPEARANCE SIDE OF PART.

19. MOLD DESIGN TO MINIMIZE EJECTION PIN MARKS, GATE BLUSH, LINES AND WELD MARKS. MOLD CONSTRUCTION TO CONFORM TO GOOD MOLDING INDUSTRY PRACTICE AS STATED IN THE CURRENT EDITION OF "STANDARD PRACTICES OF CUSTOM MOLDERS" BY THE SOCIETY OF PLASTIC INDUSTRY, INC.

20. MARK APPLE PART NUMBER AND REVISION LETTER WITH 3.0 MINIMUM HIGH CHARACTERS APPROXIMATELY WHERE SHOWN.

21. EJECTOR PIN, PARTING LINE, AND GATE LOCATIONS MUST BE APPROVED BY APPLE COMPUTER PRODUCT DESIGN ENGINEERING PRIOR TO MOLD FABRICATION.

22. MOLD TO BE PROPERTY OF APPLE COMPUTER, INC. AND SHALL BE PERMANENTLY MARKED WITH APPLE'S NAME AND APPROPRIATE TOOL NUMBER.

23. STARRED (*) DIMENSIONS AND NOTES ARE CRITICAL CONTROL DIMENSIONS AND NOTES FOR QC INSPECTION.

24. THIS DRAWING CONTAINS CRITICAL TO FUNCTION DIMENSIONS ONLY. REFER TO UNIGRAPHICS CAD FILE (815-1186-03) FOR COMPLETE PART INFO.

SECTION D-D

SECTION A-A

SECTION C-C

SECTION B-B

FIGURE D-7

CD bezel for Macintosh Quadra 840AV

815-1186-04

NOTES (UNLESS SPECIFIED OTHERWISE):

1. MATERIAL: CRS 1.00mm THICK
   FINISH: ELECTROGALVANIZED ZINC PRE-PLATE
           (FINGERPRINT-LESS)

2. TYPICAL INSIDE BEND RADIUS: 1.0mm.

3. UNSPECIFIED RADII TO BE 1.0mm.

4. BREAK AND DEBURR ALL SHARP CORNERS AND EDGES. MAXIMUM BURR ALLOWANCE
   IS 15% OF MATERIAL THICKNESS.

5. INTERPRET DIMENSIONS AND TOLERANCES PER ANSI Y14.5-1982.

6. SURFACES TO BE FREE OF CONTAMINANTS, METAL FLAKES AND LUBRICANTS.

7. STRAIGHT EDGES TO HAVE A STRAIGHTNESS TOLERANCE OF 0.20 PER 25.0,
   NOT TO EXCEED 0.40 OVER THE ENTIRE LENGTH.

8. FLAT SURFACES TO HAVE A FLATNESS TOLERANCE OF 0.20 PER 25.0, NOT
   TO EXCEED 0.40 OVER THE ENTIRE SURFACE.

9. STEEL RULE DIES, SPECIAL PUNCHES AND OTHER UNIQUE TOOLING REQUIRED
   TO FABRICATE THIS PART TO BE PROPERTY OF APPLE COMPUTER, INC. AND
   SHALL BE PERMANENTLY MARKED WITH APPLE'S NAME AND APPROPRIATE APPLE
   PART NUMBER.

10. STAMP ASSEMBLY NUMBER 600-0627 AND
    REVISION LEVEL WITH 3.0mm MINIMUM
    HIGH CHARACTERS APPROXIMATELY WHERE
    INDICATED.

11. INSTALL M3 THREADED STANDOFF (PEM
    PART NO. SO-M3-10 OR EQUIV.) WHERE
    INDICATED (2 PLACES).

12. EXTRUDE HOLE AND TAP M3 THREADS
    WHERE INDICATED (2 PLACES). HOLES TO
    HAVE A MINIMUM OF THREE (3) THREADS.

13. THIS IS A SUPPLEMENTAL DETAIL DRAWING
    AND IS TO BE USED IN CONJUNCTION WITH
    PART MODEL FILE 805-0530-06w.igs TO
    MANUFACTURE AND INSPECT THE PART.

SECTION A—A
5 PLACES
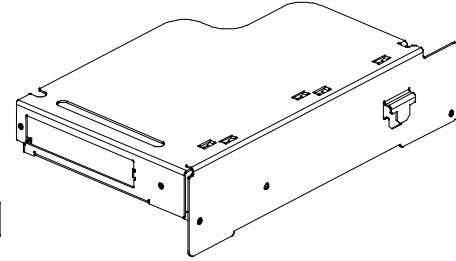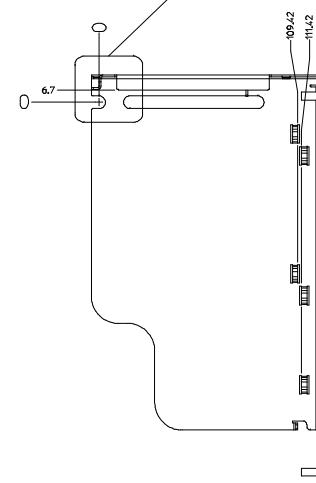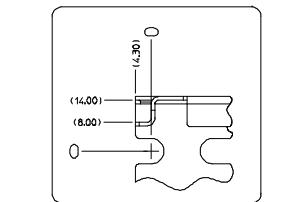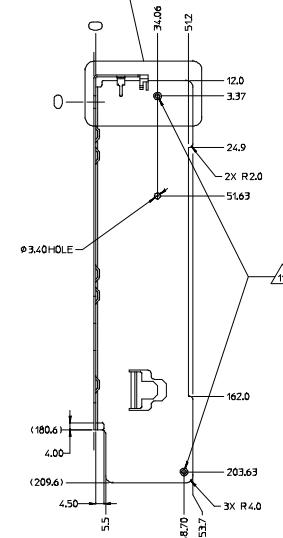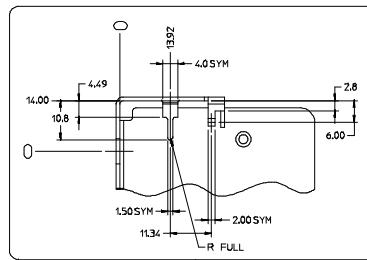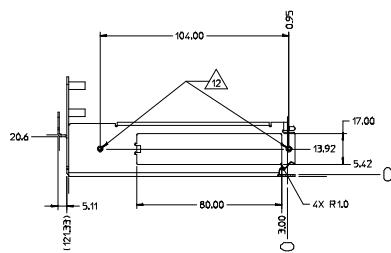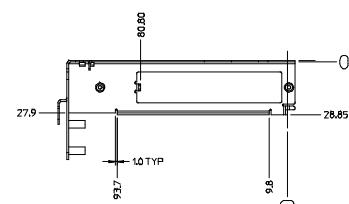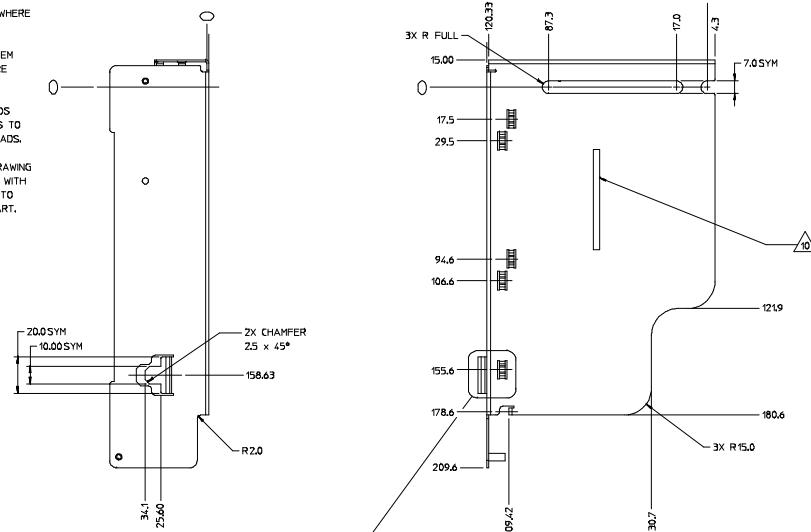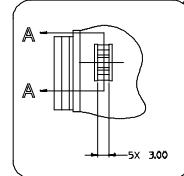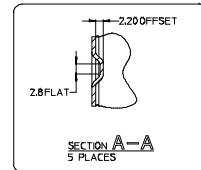
FIGURE D-8

Accessory-card bracket for the
Macintosh Centris 660AV

805-0530-06

NOTES: UNLESS OTHERWISE SPECIFIED:

⚠1 MATERIAL POLYCARBONATE SHEET, 0.20 (.008) THICK FLAMMABLILTY
    RATING TO BE LL94-V2 MINIMUM.

⚠2 ADHESIVE MATERIAL 3M 467 ADHESIVE OR EQUIVALENT.

⚠3 ADHESIVE TO BE APPLIED TO SURFACE AS INDICATED, AND MUST BE
    COVERED WITH A REMOVABLE PAPER ELEMENT.

4   ALL REQUIRED TOOLING TO BE PROPERTY OF APPLE COMPUTER, INC AND SHALL
    BE MARKED WITH APPLE'S NAME. APPROPRIATE PART NUMBER, AND DATE.

110.1

R2.03X

176.1

1

2
3
NEARSIDE

R15.02X

58.8

35.1

60.1

75.1
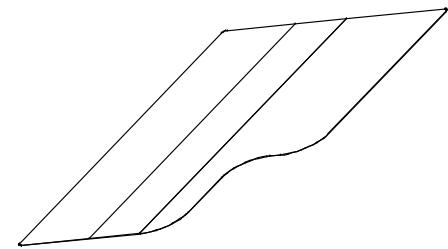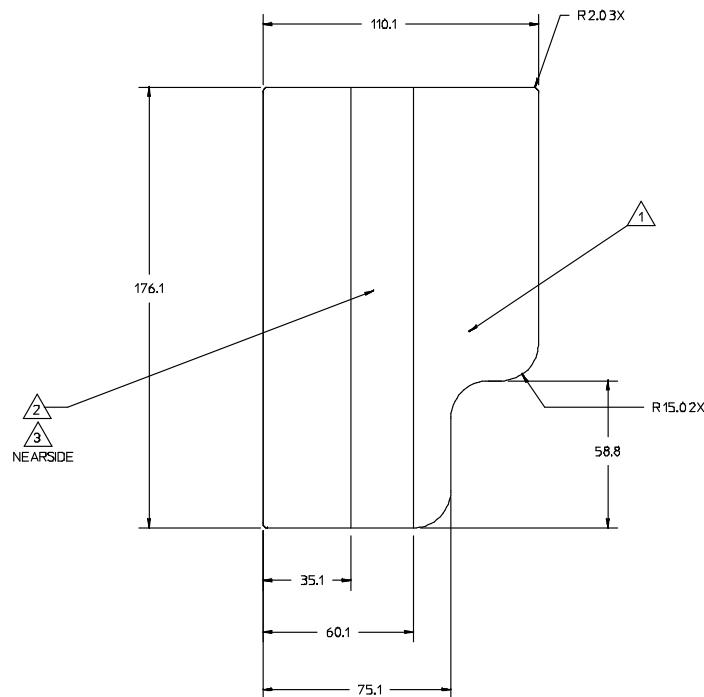
FIGURE D-9

Insulator for the Macintosh Centris 660AV
accessory-card bracket

725-0051-02

NOTES (UNLESS SPECIFIED OTHERWISE):

1 MATERIAL: 0.20 (.008") THK STAINLESS
          STEEL 301 SERIES, HALF-HARD.

2 TYPICAL INSIDE BEND RADIUS: 0.15mm.

3 ALL SHARP CORNERS TO BE RADIUSED 0.5R.

4 BREAK AND DEBURR ALL SHARP CORNERS AND
   EDGES. MAXIMUM BURR ALLOWANCE IS 15%
   OF MATERIAL THICKNESS.

5 INTERPRET DIMENSIONS AND TOLERANCES
   PER ANSI Y14.5-1982.

6 SURFACES TO BE FREE OF CONTAMINANTS,
   METAL FLAKES AND LUBRICANTS.

7 STRAIGHT EDGES TO HAVE A STRAIGHTNESS
   TOLERANCE OF 0.20 PER 25.0, NOT TO
   EXCEED 0.40 OVER THE ENTIRE LENGTH.

8 FLAT SURFACES TO HAVE A FLATNESS
   TOLERANCE OF 0.20 PER 25.0, NOT TO
   EXCEED 0.40 OVER THE ENTIRE SURFACE.

9 STEEL RULE DIES, SPECIAL PUNCHES AND
   OTHER UNIQUE TOOLING REQUIRED TO
   FABRICATE THIS PART TO BE PROPERTY OF
   APPLE COMPUTER, INC. AND SHALL BE
   PERMANENTLY MARKED WITH APPLE'S NAME
   AND APPROPRIATE APPLE PART NUMBER.

10 OPTIONAL STRIP MAY BE ADDED FOR
   SUPPORT DURING FORMING BUT MUST BE
   TRIMMED AWAY WHEN PART IS COMPLETED.

R5.5
100.0
94.3
13.68
27.35
15.0
7.5
Ø 4.00
10.8
92.7
93.7
14.9
17.40
8.70
12.9
MIN RELEIF TYP

[6.00]
8X 24°
2X 2.50
2X 0.6
4X [1.00]
8X 2.5
8X 10.0

91.7
11.8
[9.70]
19.40 REF
30.3
[19.80]
59.4   3 SPACES @ 19.8mm EA.
NON-ACCUM. TOL.
7.20
14.40

R FULL 8X
8X 0.5
8X 2.0
8X 1.0 RELEIF
R FULL 8X

OUTSIDE R0.70 MIN 2X
[14.40]
[27.35]
2X 6.00

OPTIONAL
10
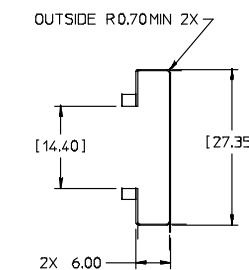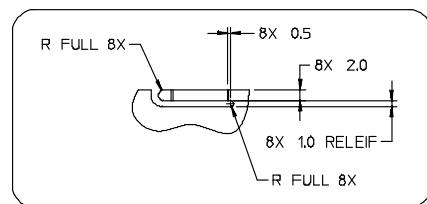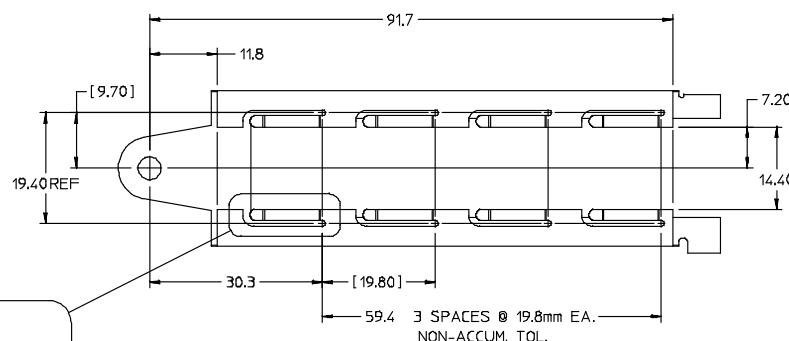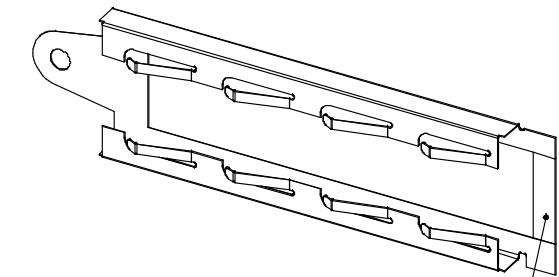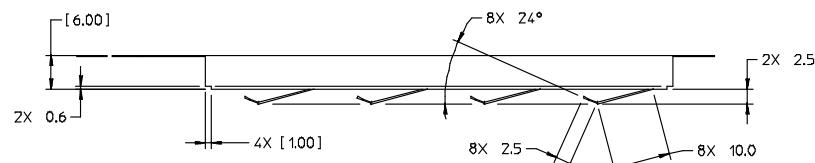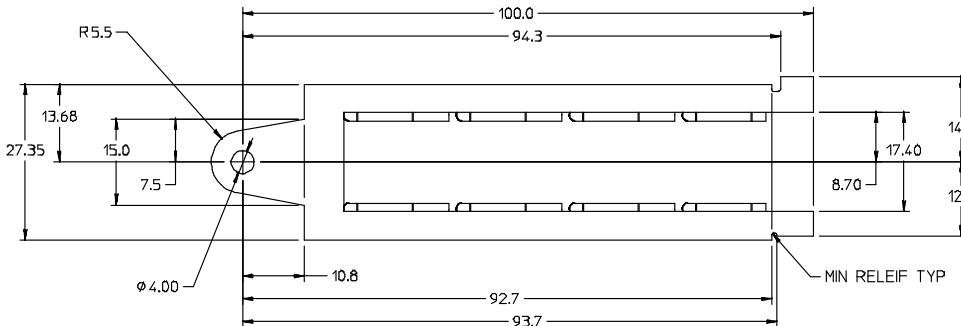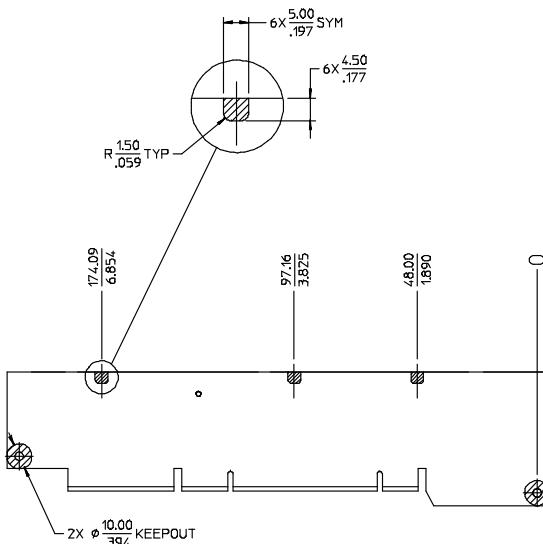
FIGURE D-10

EMI shield for the Macintosh Centris 660AV
accessory-card bracket
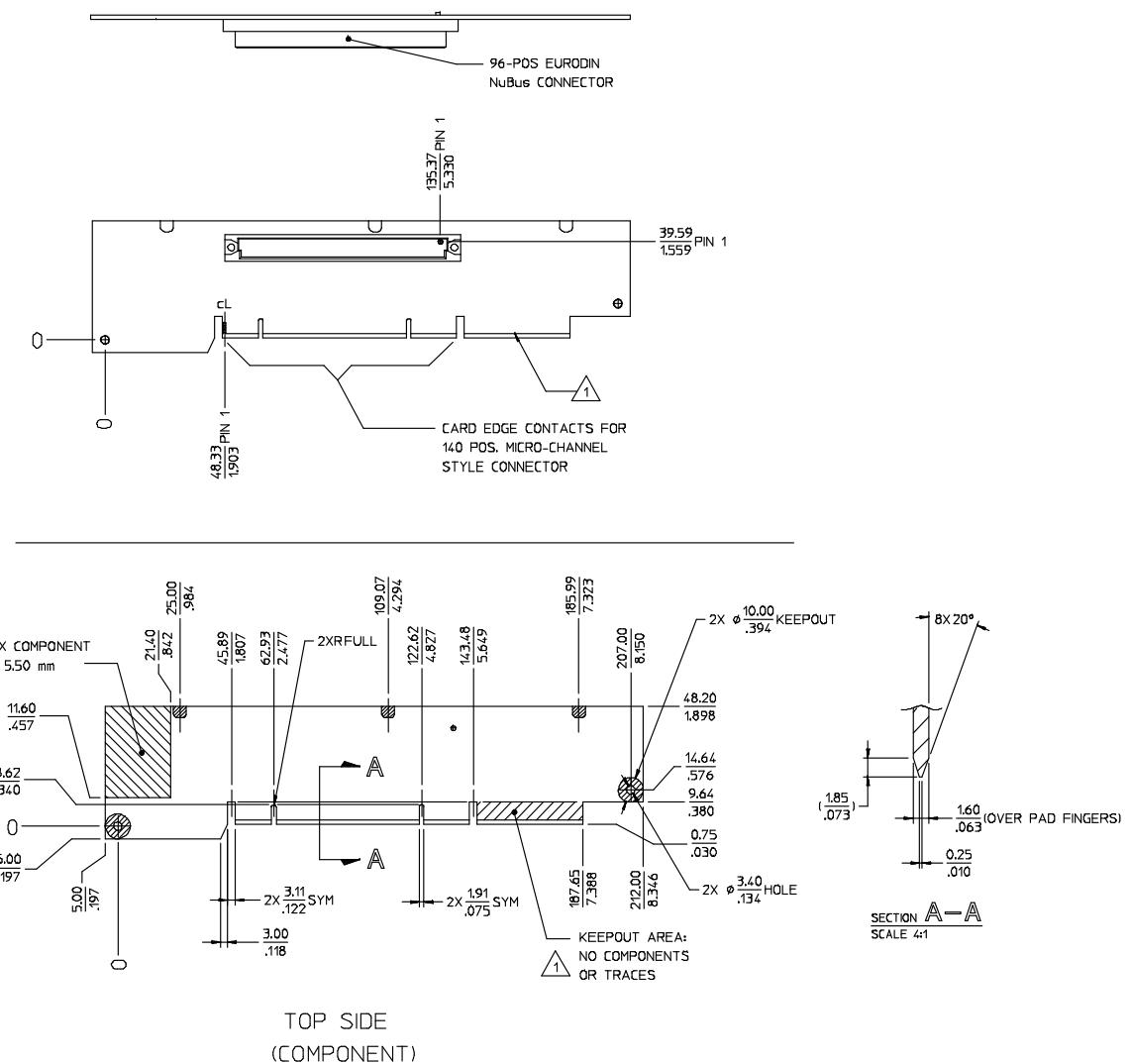
630-0450-10

⚠️1 BOARD GEOMETRY ALLOWS THIS EDGE TO BE PLUGGED INTO 212
POSITION MICRO-CHANNEL STYLE CONNECTOR.

96-POS EURODIN
NuBus CONNECTOR

$\frac{135.37}{5.330}$ PIN 1

$\frac{39.59}{1.559}$ PIN 1

$\frac{48.33}{1.903}$ PIN 1

CARD EDGE CONTACTS FOR
140 POS. MICRO-CHANNEL
STYLE CONNECTOR

⚠️1

cL

6X $\frac{5.00}{.197}$ SYM

6X $\frac{4.50}{.177}$

R $\frac{1.50}{.059}$ TYP

$\frac{174.09}{6.854}$

$\frac{97.16}{3.825}$

$\frac{48.00}{1.890}$

2X ⌀ $\frac{10.00}{.394}$ KEEPOUT

BACKSIDE

MAX COMPONENT
HT 5.50 mm

$\frac{25.00}{.984}$

$\frac{21.40}{.842}$

$\frac{45.89}{1.807}$

$\frac{62.93}{2.477}$

2X R FULL

$\frac{109.07}{4.294}$

$\frac{122.62}{4.827}$

$\frac{143.48}{5.649}$

$\frac{185.99}{7.323}$

$\frac{207.00}{8.150}$

2X $\frac{10.00}{.394}$ KEEPOUT

8X 20°

$\frac{11.60}{.457}$

$\frac{48.20}{1.898}$

$\frac{8.62}{.340}$

$\frac{14.64}{.576}$

A

A

$\frac{9.64}{.380}$

$\frac{0.75}{.030}$

2X ⌀ $\frac{3.40}{.134}$ HOLE

$(\frac{1.85}{.073})$

$\frac{1.60}{.063}$ (OVER PAD FINGERS)

$\frac{0.25}{.010}$

$\frac{5.00}{.197}$

$\frac{5.00}{.197}$

2X $\frac{3.11}{.122}$ SYM

$\frac{3.00}{.118}$

2X $\frac{1.91}{.075}$ SYM

$\frac{187.65}{7.388}$

$\frac{212.00}{8.346}$

KEEPOUT AREA:
NO COMPONENTS
OR TRACES

⚠️1

SECTION A—A
SCALE 4:1

TOP SIDE
(COMPONENT)

FIGURE D-11

NuBus adaptor card for the Macintosh Centris 660AV

630-0450-12

Glossary

ADB See **Apple Desktop Bus.**

AIAO See **all-in/all-out buffer.**

all-in/all-out buffer (AIAO) A buffer that is completely emptied each time it is read.

ANSI American National Standards Institute.

APDA Apple's worldwide direct distribution channel for Apple and third-party development tools and documentation products.

API See **application programming interface.**

Apple Desktop Bus (ADB) An asynchronous bus used to connect relatively slow user-input devices to Apple computers.

Apple SuperDrive Apple's disk drive for high-density floppy disks.

AppleTalk Apple's local area networking protocol.

Apple Telecom External Clock Synchronizer (ATECS) A chip that synchronizes the DSP and sound subsystems to external clock signals received through a serial port.

application programming interface (API) A set of calls, instructions, and data structures in system software or a processor instruction set that application software can use to program the computer.

arbitration The process of determining which of several contending subsystems gains control of a bus at any given time.

ATA See **average timeshare available.**

ATECS See **Apple Telecom External Clock Synchronizer.**

ATT See **average total timeshare.**

ATU See **average timeshare used.**

AutoCache In digital signal processing, a visible caching model in which the DSP operating system performs all load and save functions automatically.

average timeshare available (ATA) The average amount of time per frame that the DSP is in sleep mode. This time can be used for timesharing tasks.

average timeshare used (ATU) The average amount of time per frame that the DSP spends executing timeshare tasks.

average total timeshare (ATT) The sum of **average timeshare available** and **average timeshare used.**

baud The maximum number of signal changes per second on a transmission line.

block transfer Data transfers of more than one longword at a time.

cache load In digital signal processing, the process of moving data from local memory to cache memory.

cache save In digital signal processing, the process of moving data from cache memory to local memory.

CAM See **Common Access Method.**

CAS See **column address strobe.**

Casper The code name for Apple's speech recognition human interface and technology.

CCIR Comité Consultatif International Radio.

CD-ROM See **compact disc ROM.**

CIVIC See **Cyclone Integrated Video Interfaces Controller.**

client In DSP programming, an application or system toolbox routine that uses the DSP.

Clifton Plus A functional equivalent of the Endeavor chip, used in the Macintosh Centris 660AV.

CMOS See **complementary metal-oxide semiconductor.**

codec A digital encoder and decoder.

color depth   The number of bits required to encode the color of each pixel in a display.

column address strobe (CAS)   A signal that captures the column component of a matrix addressing scheme from a bus that carries both row and column addresses.

command item   A user-selectable button in a dialog box that can be operated by voice control—for example, the OK or Cancel button.

Common Access Method   A specification for SCSI operation embodied in ANSI Standard X3T9.

compact disc ROM (CD-ROM)   A read-only data storage disk 120 mm in diameter that can hold up to 550 MB of data.

complementary metal-oxide semiconductor (CMOS)   A chip material and fabrication technology that features low power requirements and high noise immunity.

composite video   A video signal that includes both picture information (with chroma and luminance combined) and the timing and other signals needed to display it. It is the standard signal form for communication between video cassette recorders, television sets, and other common video equipment.

container   In DSP programming, a memory location occupied by a section.

convolution   The process of smoothing alternate lines of a video signal to be shown in succeeding frames for a line-interlaced display.

CPU bus   The bus connected directly to the main processor.

Cuda   A microcontroller chip that manages the ADB and real-time clock, maintains parameter RAM, manages power on and reset, and performs other general system functions.

Curio   An I/O chip that supports Ethernet, SCSI, SCC, and LocalTalk.

Cyclone Integrated Video Interfaces Controller (CIVIC)   A video control chip that manages VRAM, generates video timing signals, and performs convolution where needed.

DAC   See **digital-to-analog converter.**

data burst   Multiple longwords of data sent over a bus in a single, uninterrupted stream.

delimiter   A character or character pair used to set off embedded speech commands in speech synthesis.

DemandCache   In digital signal processing, a visible caching model in which the program explicitly moves code and data blocks between on-chip memory and off-chip memory.

digital audio/video (DAV) expansion connector   A connector in line with a NuBus slot that lets a plug-in card access digital sound and unscaled YUV video data directly.

Digital Multistandard Decoder (DMSD)   A video chip that decodes the color information in NTSC, PAL, and SECAM video signals.

digital signal processor (DSP)   A chip that performs fast real-time data processing tasks, such as speech recognition and audio compression.

digital-to-analog converter (DAC)   Circuitry that produces analog electrical levels in response to digital data.

direct memory access (DMA)   A process of transferring data rapidly into or out of RAM without passing it through a processor or buffer.

DMA   See **direct memory access.**

DMSD   See **Digital Multistandard Decoder.**

DRAM   See **dynamic random-access memory.**

DSP   See **digital signal processor.**

DSP map   A data structure used by the Real Time Manager to hold intertask buffer information.

DSP operating system   Software built into the DSP chip (independent of the Macintosh Operating System) that supports DSP programming and operation.

dumb lumpy algorithm   A DSP operation that varies in running time and for which the program cannot determine before a frame how long it will take to run. See also **smart lumpy algorithm.**

duration control   A control code in synthesized speech that determines the duration of one or more previous allophones.

dynamic random-access memory (DRAM) Random-access memory in which each storage address must be periodically interrogated ("refreshed") to maintain its value.

embedded speech command   In speech synthesis, an instruction placed in text being spoken to indicate the rhythm, phrasing, modulation, or tone of delivery.

Endeavor   A chip that generates video clock signals for a variety of different monitors.

ending prosody   The modulation that distinguishes the end of a sentence or statement in normal speech.

Ethernet   A high-speed local area network technology that includes both cable standards and a series of communications protocols.

exception vector table (EVT)   A data structure in which the DSP operating system places system information, such as run-time variables and routine addresses.

facsimile (fax)   A data format and transmission protocol for sending graphic images over telephone lines.

fax   See **facsimile.**

FIFO   See **first-in, first-out.**

first-in, first-out (FIFO)   A data-buffering technique in which bytes are read out in the same order in which they were received.

floating-point format   A data format that encodes real numbers, including decimals.

floating-point unit (FPU)   A part of the MC68040 processor that calculates numbers in floating-point format.

frame   In DSP programming, the repeating time period during which DSP code runs.

frame-based processing   The DSP processing technique in which data is processed during a fixed time interval (a frame).

GCR   See **Group Code Recording.**

GeoPort   Apple's versatile, high-performance serial interface that communicates with most telephone systems worldwide by means of external pods.

GPB   See **guaranteed processing bandwidth.**

Group Code Recording (GCR)   The Apple recording format for floppy disks.

guaranteed processing bandwidth (GPB)   A concept in DSP programming that lets the programmer make sure that the DSP will be able to complete its required tasks during every frame.

HAL   See **hardware abstract layer.**

hardware abstract layer (HAL)   An API layer in the DMA Serial Driver that makes the driver hardware independent.

HBA   See **host bus adapter.**

host   A Macintosh application from the viewpoint of a DSP program.

host bus adapter (HBA)   The hardware associated with a specific SCSI bus adapter.

IEEE   Institute of Electrical and Electronics Engineers.

I/O   See **input/output.**

input/output (I/O)   Parts of a computer system that transfer data to or from peripheral devices.

Integrated Services Digital Network (ISDN)   A series of protocols that integrate voice and data transmission over telephone lines.

intermodule buffer   A buffer used to pass data between DSP modules.

interrupt latency   The maximum time that a program can delay responding to an interrupt without affecting the performance of the operating system or peripheral devices.

intertask buffer (ITB)   A buffer used to pass data between DSP tasks.

ISDN   See **Integrated Services Digital Network.**

ITB   See **intertask buffer.**

LocalTalk   The cable terminations and other hardware that Apple supplies for local area networking from Macintosh serial ports.

LocalTalk Patch Chip (LTPC) A chip that processes LocalTalk signals to and from the printer port.

logical unit number (LUN) A logical ID that identifies a SCSI device for the SCSI Manager.

LTPC See **LocalTalk Patch Chip.**

lumpy algorithm A DSP operation whose running time may vary from frame to frame. See also **smooth algorithm.**

LUN See **logical unit number.**

MACE See **Media Access Controller for Ethernet.**

Macintosh Universal NuBus Interface (MUNI) A control and interface chip between NuBus and the MC68040 processor.

MCA See **Memory Controller and Arbiter.**

MC68040 The model number of the Motorola processor used in the Macintosh Quadra 840AV and Macintosh Centris 660AV.

Media Access Controller for Ethernet (MACE) Circuitry within Curio that supports Ethernet I/O.

Memory Controller and Arbiter (MCA) A memory manager chip that controls access to ROM and RAM and performs arbitration for the CPU bus.

MFM See **Modified Frequency Modulation.**

Mickey A video encoder that produces composite and S-video outputs in NTSC and PAL formats.

mini-DIN An international standard form of cable connector for peripheral devices.

Modified Frequency Modulation (MFM) A recording format for floppy disks used by DOS computers.

module The basic unit of DSP programming. A module always includes DSP code and may also include data, I/O buffers, and parameter blocks.

MUNI See **Macintosh Universal NuBus Interface.**

New Age A controller chip for Apple floppy disk drives.

NTSC An acronym for National Television Standards Committee, the television signal format common in North America, Japan, parts of South America, and other regions.

NuBus A bus architecture in Apple computers that supports plug-in accessory cards. The Macintosh Quadra 840AV contains three NuBus slots.

Open Scripting Architecture (OSA) A standard for the operation of scripting systems (such as AppleScript and QuicKeys).

option item A radio button or checkbox in a dialog box, which may or may not be voice controlled.

OSA See **Open Scripting Architecture.**

PAL An acronym for Phased Alternate Lines, the television signal format common in Western Europe (except France), Australia, parts of South America, most of Africa, and Southern Asia.

parameter RAM Random-access memory in an Apple computer that retains data when the computer is turned off.

PBX See **Private Branch Exchange.**

PDS See **processor-direct slot.**

Peripheral Subsystem Controller (PSC) A control chip that manages DMA, handles system interrupts, and performs other tasks.

phoneme A single sound element of synthesized speech.

pitch In synthesized speech, the dominant frequency of an utterance.

pixel A single dot on a screen display.

Private Branch Exchange (PBX) The traditional transmission standard for voice telephone.

processor-direct slot (PDS) A connector in the Macintosh Centris 660AV only that lets a plug-in card access the CPU bus directly. The same connector also accepts a NuBus adapter card.

prosody The rhythm, modulation, and stress patterns of speech.

PSC See **Peripheral Subsystem Controller.**

RAS See **row address strobe.**

Real Time Manager   A part of the system software for the Macintosh Quadra 840AV and Macintosh Centris 660AV that lets applications control the DSP.

real-time processing   Data processing that occurs within the time constraints of another process, such as manipulating a digital video stream.

relative pitch control   A control code in synthesized speech that determines the pitch relative to the pitch range for the current voice.

RGB   Abbreviation for *red-green-blue.* A data format for each pixel of a color display in which the red, green, and blue values are separately encoded.

row address strobe (RAS)   A signal that captures the row component of a matrix addressing scheme from a bus that carries both row and column addresses.

RS-232, RS-422   Standard communications protocols established by the Electronics Industries Association for serial data transmission.

scatter/gather (S/G) list   A list of discontiguous locations in memory where a single run of data is located.

SCC   See **Serial Communications Controller.**

SCSI   See **Small Computer System Interface.**

SCSI Interface Module (SIM)   A lower layer of the SCSI Manager 4.3, which interfaces with host bus adapters.

Sebastian   A video color manager and digital-to-analog converter on one chip.

SECAM   A French acronym for the television signal format used in France, Eastern Europe, the former Soviet Union, and many former French colonies.

section   In DSP programming, a part of a module that is stored in a locked contiguous memory block.

section table   A data structure maintained by the DSP operating system to keep track of active containers.

Serial Communications Controller (SCC)   Circuitry on the Curio chip that provides an interface to the serial data ports.

S/G list   See **scatter/gather list.**

SIM   See **SCSI Interface Module.**

SIMM   See **Single Inline Memory Module.**

Singer   A digital encoder and decoder (codec) for analog sound data, including speech.

Single Inline Memory Module (SIMM)   A plug-in card for expanding RAM that contains several RAM chips and their interconnections.

sleep mode   The idle state of the DSP during the remainder of a frame after all required processing tasks have been completed.

Small Computer System Interface (SCSI)   An industry standard parallel bus protocol for connecting computers with peripheral devices such as hard disk drives.

smart lumpy algorithm   A DSP operation that varies in running time but for which the program can determine before each frame how long it will take to run. See also **dumb lumpy algorithm.**

SME   See **Speech Macro Editor.**

smooth algorithm   A DSP operation that always takes substantially the same time to run. See also **lumpy algorithm.**

speech macro   A user-defined routine that specifies an utterance to be recognized plus a set of instructions to be followed when it is recognized.

Speech Macro Editor (SME)   An application shipped with the Macintosh Quadra 840AV and Macintosh Centris 660AV that lets users edit speech macros.

Speech Monitor   A background application that supports speech recognition.

speech rule   An instruction to the Speech Monitor for recognizing and acting on certain words and phrases. Speech rules are kept in files in a special folder in the System Folder.

speech rules file   A file in the System Folder or Extensions folder that contains speech rules.

Speech Setup control panel   A control panel, accessible through the Apple menu, that lets users customize the computer's speech recognition behavior.

Standard Sound   The DSP Sound Driver and a set of common sound-manipulation tasks, all of which are part of the Real Time Manager software.

S-video   A video format in which chroma and luminance are transmitted on separate lines. It provides higher image quality than composite video.

task   A group of DSP modules that always run together.

TIB   See **transfer information block.**

timeshare processing   Data processing that uses DSP facilities after real-time tasks are done, such as file compression.

transfer information block (TIB)   A SCSI Manager data structure that communicates instructions about the transferring of data through the SCSI port.

transport (XTP) layer   The upper level of the SCSI Manager 4.3, which interfaces with old and new SCSI drivers.

Truecolor   A color range encoded by 24 bits.

VDC   See **Video Data Path Chip.**

Versatile Interface Adapter (VIA)   The interface for system interrupts that is standard on most Apple computers.

VIA   See **Versatile Interface Adapter.**

Video Data Path Chip (VDC)   A chip that converts video in YUV format to RGB format and performs video window scaling.

video frame buffer   Memory that stores one or more frames of video information until they are displayed on a screen.

video RAM (VRAM)   Random-access memory used to store both static graphics and video frames.

virtual memory (VM)   A system of memory storage that translates addresses used by software into physical addresses that may be different.

visible caching   A DSP programming technique in which off-chip code is stored on-chip in a cache accessible to the application.

VM   See **virtual memory.**

voice   A particular style of utterance in speech synthesis, such as male adult English.

voice synthesizer   A utility that cooperates with the Speech Manager to generate speech of a particular kind.

volume control   A control code in synthesized speech that determines the loudness of an utterance.

VRAM   See **video RAM.**

XTP   See **transport layer.**

YUV   A data format for each pixel of a color display in which color is encoded by values calculated from its native red, green, and blue components.

Index