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_PBs.

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

PUMA EXHIBIT 2005
PART 9 OF 10

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.

▲ **WARNING**
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 *);
```

CHAPTER 10

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.

CHAPTER 11

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
```

CHAPTER 12

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.

CHAPTER 13

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

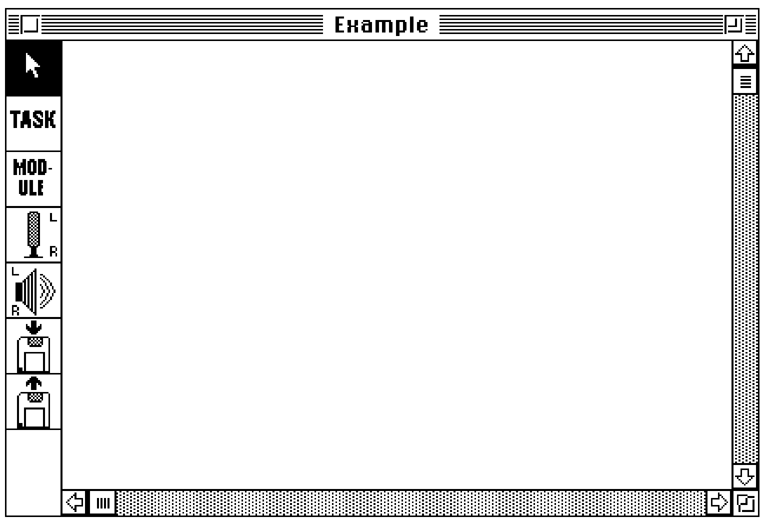
- copy the application to your hard drive
- launch BugLite

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.