SCSI Manager

This chapter describes the original Macintosh SCSI Manager. The ***SCSI Manager*** is the part of the Macintosh Operating System that controls the transfer of data between a Macintosh computer and peripheral devices connected through the Small Computer System Interface (SCSI).

In 1993, Apple Computer introduced SCSI Manager 4.3, an enhanced version of the SCSI Manager that provides new features as well as compatibility with the original version. SCSI Manager 4.3 is described in the chapter "SCSI Manager 4.3" in this book.

SCSI Manager 4.3 Note

Throughout this chapter, notes like this one are used to point out areas where SCSI Manager 4.3 differs from the original SCSI Manager. ◆

You should read this chapter if you are writing a SCSI device driver or other program that needs to be compatible with the original SCSI Manager. To make best use of this chapter, you should understand the Device Manager and how device drivers are implemented in Macintosh computers. You should also be familiar with the SCSI-1 specification established by the American National Standards Institute (ANSI). The SCSI-1 specification appears in ANSI document X3.131-1986, entitled *Small Computer System Interface*. Unless otherwise noted, all mentions of a SCSI specification in this chapter refer to the SCSI-1 specification.

If you are designing a SCSI peripheral device for Macintosh computers, you should read *Designing Cards and Drivers for the Macintosh Family*, third edition, and *Guide to the Macintosh Family Hardware*, second edition.

This chapter provides a brief introduction to SCSI concepts and then explains

- how the SCSI standard is implemented on Macintosh computers
- how data is structured on SCSI disk drives and other block devices
- how you can use SCSI Manager routines and data structures to transfer data to and from SCSI peripheral devices

Introduction to SCSI Concepts

The ***Small Computer System Interface (SCSI)*** is a computer industry standard for connecting computers to peripheral devices such as hard disk drives, CD-ROM drives, printers, scanners, magnetic tape drives, and any other device that needs to transfer large amounts of data quickly.

The SCSI standard specifies the hardware and software interface at a level that minimizes dependencies on any specific hardware implementation. The specification allows a wide variety of peripheral devices to be connected to many types of computers.

A ***SCSI bus*** is a bus that conforms to the physical and electrical specifications of the SCSI standard. A ***SCSI device*** refers to any unit connected to the SCSI bus, either a peripheral device or a computer. Each SCSI device on the bus is assigned a ***SCSI ID***, which is an integer value from 0 to 7 that uniquely identifies the device during SCSI transactions.

The Macintosh computer is always assigned the SCSI ID value of 7, and its internal hard disk drive is normally assigned the SCSI ID value of 0. In general, only one Macintosh computer can be connected to a SCSI bus at a given time, and most Macintosh models support only a single SCSI bus.

SCSI Manager 4.3 Note

Under the original SCSI Manager, the dual SCSI buses in high-performance computers such as the Macintosh Quadra 950 are treated as though they were a single physical bus. SCSI Manager 4.3 supports multiple SCSI buses and treats each bus separately. ◆

When two SCSI devices communicate, one device acts as the ***initiator*** and the other as the ***target***. The initiator begins a transaction by selecting a target device. The target responds to the selection and requests a command. The initiator then sends a SCSI command, and the target carries out the action. After acknowledging the command, the target controls the remainder of the transaction. The role of initiator and target is fixed for each device, and does not usually change. Under the original SCSI Manager, the Macintosh computer always acts as initiator, and peripheral devices are always targets.

SCSI Manager 4.3 Note

SCSI Manager 4.3 allows multiple initiators, meaning that intelligent peripheral devices can initiate SCSI transactions without involving the computer. ◆

SCSI transactions involve interaction between bus signals, bus phases, SCSI commands, and SCSI messages. Although the SCSI Manager masks much of the underlying complexity of SCSI transactions, an understanding of these elements and how they interact will help you understand the role of the SCSI Manager.

The following sections briefly summarize the elements of a SCSI transaction.

SCSI Bus Signals

The SCSI specification defines 50 bus signals, half of which are tied to ground. Table 3-1 describes the 18 SCSI bus signals that are relevant to understanding SCSI transactions. Nine of these signals are used to initiate and control transactions, and nine are used for data transfer (8 data bits plus a parity bit).

Table 3-1 SCSI bus signals

Signal	Name	Description
/BSY	Busy	Indicates that the bus is in use.
/SEL	Select	The initiator uses this signal to select a target.
/C/D	Control/Data	The target uses this signal to indicate whether the information being transferred is control information (signal asserted) or data (signal negated).
/I/O	Input/Output	The target uses this signal to specify the direction of the data movement with respect to the initiator. When the signal is asserted, data flows to the initiator; when negated, data flows to the target.
/MSG	Message	This signal is used by the target during the message phase.
/REQ	Request	The target uses this signal to start a request/ acknowledge handshake.
/ACK	Acknowledge	This signal is used by the initiator to end a request/ acknowledge handshake.
/ATN	Attention	The initiator uses this signal to inform the target that the initiator has a message ready. The target retrieves the message, at its convenience, by transitioning to a message-out bus phase
/RST	Reset	This signal is used to clear all devices and operations from the bus, and force the bus into the bus free phase. The Macintosh computer asserts this signal at startup. SCSI peripheral devices should never assert this signal.
/DB0–/DB7, /DBP	Data	Eight data signals, numbered 0 to 7, and the parity signal. Macintosh computers generate proper SCSI parity, but the original SCSI Manager does not detect parity errors in SCSI transactions.

SCSI Bus Phases

A SCSI bus phase is an interval in time during which, by convention, certain control signals are allowed or expected, and others are not. The SCSI bus can never be in more than one phase at any given time.

For each of the bus phases, there is a set of allowable phases that can follow. For example, the bus free phase can only be followed by the arbitration phase, or by another bus free phase. A data phase can be followed by a command, status, message, or bus free phase.

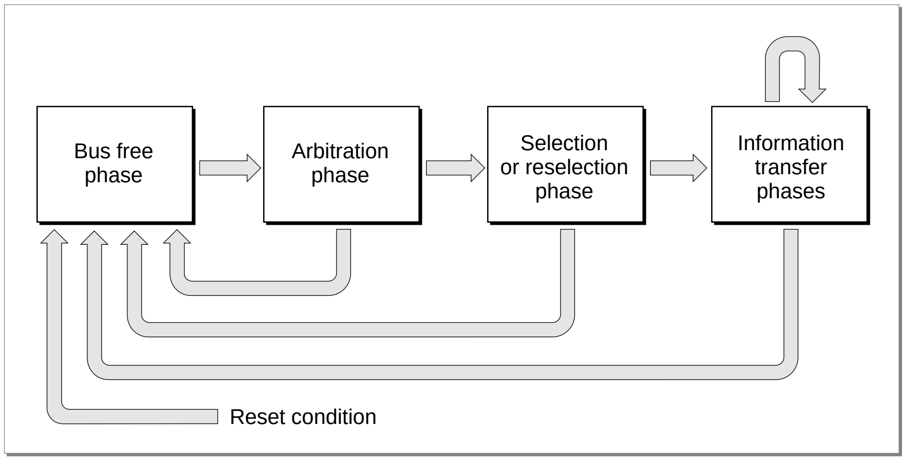
Control signals direct the transition from one phase to another. For example, the reset signal invokes the bus free phase, while the attention signal invokes the message phase.

The SCSI standard specifies eight distinct phases for the SCSI bus:

- ***Bus free*** . This phase means that no SCSI devices are using the bus, and that the bus is available for another SCSI operation.
- ***Arbitration***. This phase is preceded by the bus free phase and permits a SCSI device to gain control of the SCSI bus. During this phase, all devices wishing to use the bus assert the /BSY signal and put their SCSI ID onto the bus (using the data signals). The device with highest SCSI ID wins the arbitration.
- ***Selection*** . This phase follows the arbitration phase. The device that won arbitration uses this phase to select another device to communicate with.
- ***Reselection*** . This optional phase is used by systems that allow peripheral devices to disconnect and reconnect from the bus during lengthy operations. This phase is not supported by the original Macintosh SCSI Manager, but is by SCSI Manager 4.3.
- ***Command*** . During this phase, the target requests a command from the initiator.
- ***Data***. The data phase occurs when the target requests a transfer of data to or from the initiator.
- ***Status***. This phase occurs when the target requests that status information be sent to the initiator.
- ***Message*** . The message phase occurs when the target requests the transfer of a message. Messages are small blocks of data that carry information or requests between the initiator and a target. Multiple messages can be sent during this phase.

Together, the last four phases (command, data, status, and message) are known as the information transfer phases. Figure 3-1 shows the relationship of the SCSI bus phases.

Figure 3-1 SCSI bus phases and allowable transitions



SCSI Commands

A ***SCSI command*** is an instruction from an initiator to a target to conduct an operation, such as reading or writing a block of data. Commands are read by the target when it is ready to do so, as opposed to being sent unrequested by the initiator.

SCSI commands are contained in a data structure called a ***command descriptor block (CDB)***, which can be 6, 10, or 12 bytes in size. The first byte specifies the operation requested, and the remaining bytes are parameters used by that operation.

A single SCSI command may cause a peripheral device to undertake a relatively large amount of work, compared with other device interfaces. For example, the read command can specify multiple blocks of data rather than just one. The primary difference between the SCSI protocol and other interfaces typically used for storage devices is that SCSI commands address a device as a series of logical blocks rather than in terms of heads, tracks, and sectors. It is this abstraction from the physical characteristics of the device that allows the SCSI protocol to be used with a wide variety of devices.

SCSI Messages

The SCSI standard specifies a number of possible messages between initiator and target. ***SCSI messages*** are small blocks of data, often just one byte in size, that indicate the successful completion of an operation (the command complete message), or a variety of other events, requests, and status information. All messages are sent during the message phase.

The command complete message is required in all SCSI implementations. This message is sent from the target to the initiator and indicates that a command (or series of linked commands) has been completed, either successfully or unsuccessfully. Success or failure of the command is indicated by status information sent earlier during the status phase. The importance of the command complete message is more fully discussed in “Using the SCSIComplete Function,” beginning on page 3-21.

Other SCSI messages are optional. During the selection phase, the initiator and target each specify their ability to handle messages other than the command complete message.

SCSI Handshaking

The SCSI standard defines the required sequence of transitions of the control and data signals to ensure reliable communication between SCSI devices. Because the request signal (/REQ) and the acknowledge signal (/ACK) both play a major role, this part of the SCSI protocol is often referred to as request/acknowledge handshaking (usually abbreviated as REQ/ACK handshaking).

The SCSI information transfer phases use REQ/ACK handshaking to transfer data or control information between the initiator and target, in either direction. The direction of the transfer depends on the particular bus phase. The handshaking occurs on every byte transferred, and constitutes the lowest level of the SCSI protocol.

For example, during the data phase, when a target sends data to the initiator, the target places the data on the SCSI bus data lines and then asserts the /REQ signal. The initiator senses the /REQ signal, reads the data lines, then asserts the /ACK signal. When the target senses the /ACK signal, it releases the data lines and negates the /REQ signal. The initiator then senses that the /REQ signal has been negated, and negates the /ACK signal. After the target senses that the /ACK signal has been negated, it can repeat the whole process again, to transfer another byte of data.

Unless you are designing a SCSI device, you do not need any special knowledge of SCSI handshaking to write software that uses the SCSI Manager. However, a general understanding of SCSI handshaking can be helpful when debugging. Refer to the SCSI specification for complete information about SCSI handshaking, bus phases, commands, and messages.

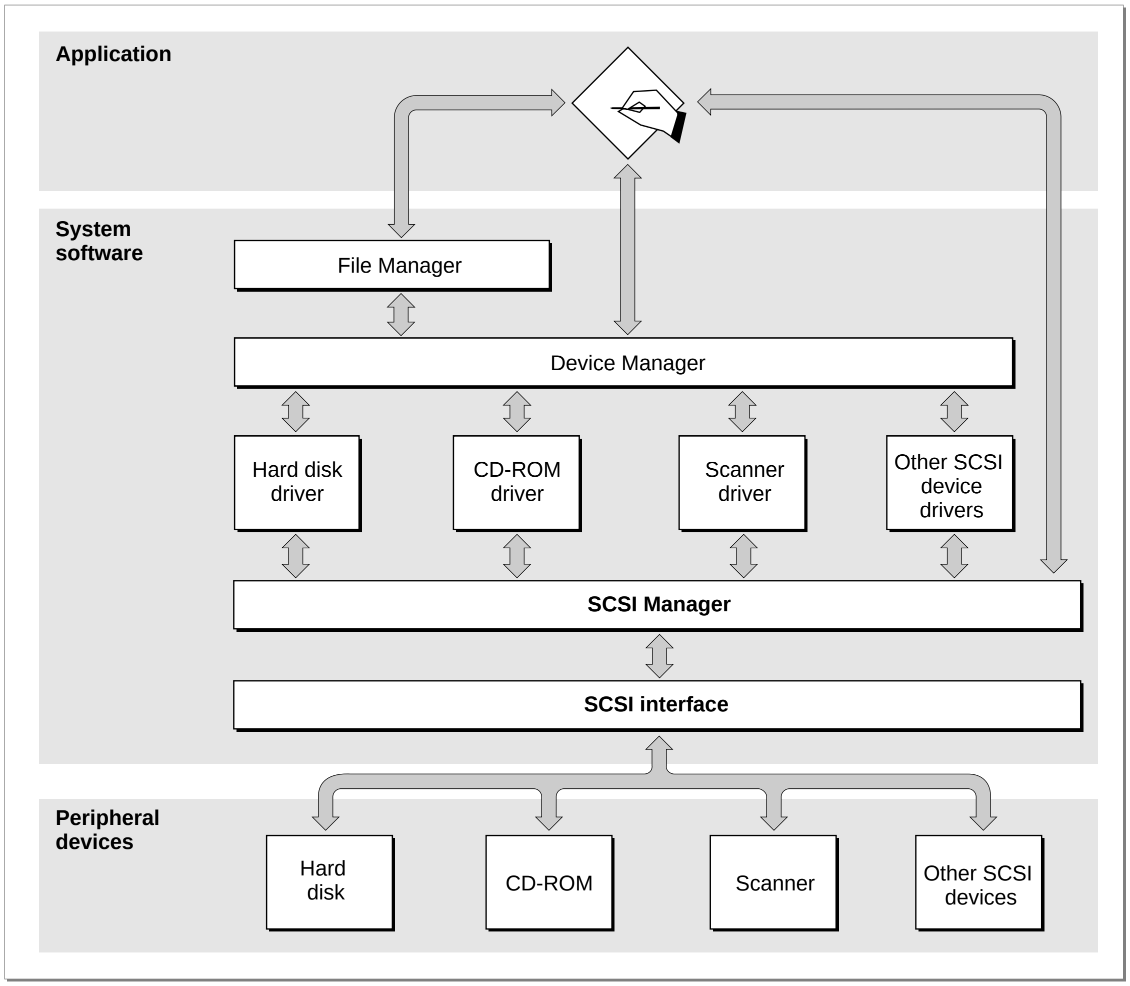
About the SCSI Manager

The SCSI Manager provides routines that allow Macintosh device drivers and other programs to communicate with SCSI peripheral devices using the SCSI protocol.

The SCSI Manager is a software layer that mediates between device drivers or applications and the SCSI controller hardware in the Macintosh computer. In some cases, the amount of mediation is small. For example, the SCSI Manager `SCSIReset` function does little except assert the reset signal on the SCSI bus. In other cases, a single SCSI Manager function may initiate a relatively complex series of actions.

Figure 3-2 shows the relationship of the SCSI Manager to the Macintosh system architecture. The architecture consists of multiple layers: the application layer, the system software layer (which is composed of several subordinate layers), and the hardware layer.

Figure 3-2 The role of the SCSI Manager



Application programs usually rely on high-level services such as those provided by the File Manager, but may also call low-level services directly. The File Manager calls the Device Manager, which calls the appropriate device driver. SCSI device drivers do not control SCSI hardware directly; they use the SCSI Manager to communicate with SCSI devices.

Conformance With the SCSI Specification

The SCSI specification has been revised considerably since the first Macintosh SCSI implementation. For information about the SCSI standard as originally defined, see ANSI document X3.131-1986, *Small Computer System Interface*. Many of the features described in the newer SCSI-2 specification are supported by SCSI Manager 4.3. However, the original SCSI Manager predates these extensions.

Due to hardware variations among Macintosh models, there are minor differences in the behavior of some SCSI Manager routines. These differences lie mostly outside the scope of the SCSI protocol. For information about these differences, see the description of the `SCSIGet` function on page 3-32.

All Macintosh computers support these aspects of the SCSI specification:

- multiple targets
- as many as eight devices on the bus (the computer and up to seven peripherals)
- parity generation

The following optional features of the SCSI specification are not supported by the original SCSI Manager:

- multiple SCSI buses
- multiple initiators on a single bus
- disconnect/reconnect
- parity error detection

SCSI Manager 4.3 Note
These features and other enhancements are supported by SCSI Manager 4.3. ◆

Overview of SCSI Manager Data Structures

The SCSI specification and the Macintosh Operating System define a number of data structures for communicating with SCSI devices. These data structures fall into three categories:

- structures defined by the SCSI specification, such as command descriptor blocks and SCSI messages
- structures specific to the SCSI Manager, such as transfer instruction blocks and the 16-bit status word returned by the `SCSIStat` function
- structures required for the proper operation of SCSI disk drives with the Start Manager and the File Manager; for example, the driver descriptor map and the partition map

The command descriptor block and other data structures defined by the SCSI specification are not discussed in detail in this chapter. Refer to the SCSI specification for complete information about these structures. See "Using CDB and TIB Structures," beginning on page 3-17, for an example of how to send a CDB to a SCSI device.

Although the driver descriptor map and the partition map are not used by the SCSI Manager, they must be present on all block devices compatible with the Macintosh Operating System. These structures are discussed in the following section.

A ***transfer instruction block (TIB)*** is a Macintosh-specific data structure that your program uses to pass instructions to the SCSI Manager. TIB structures are used to control data transfers, and for other purposes such as comparing data on a peripheral device with data in memory. TIB structures are passed as parameters to the SCSI Manager `SCSIRead`, `SCSIRBlind`, `SCSIWrite`, and `SCSIWBlind` functions. For read operations, the TIB specifies a memory location where the data should be stored. For write operations, the TIB specifies the location of the data to be written.

Although a transfer instruction block is data, not machine-executable code, it is analogous to code in that the data is interpreted and executed by the SCSI Manager in a manner similar to executing a program. The `SCSIInstr` data type defines a transfer instruction block.

```
TYPE SCSIInstr =           {transfer instruction block}
RECORD
   scOpcode:   Integer;    {operation code}
   scParam1:   LongInt;    {first parameter}
   scParam2:   LongInt;    {second parameter}
END;
```

The first field of the transfer instruction block contains a transfer operation code. This code is not a command in the SCSI protocol, but rather an instruction to the SCSI Manager that directs the transfer of data across the SCSI bus after a SCSI command has been sent. The instruction set consists of eight operation codes that allow you to transfer data, increment a counter, and form iterative loops. See “SCSI Manager TIB Instructions,” beginning on page 3-27, for details of the TIB instruction set.

A sequence of TIB instructions is also known as a ***TIB pseudoprogram*** . Here is an example of a TIB pseudoprogram:

```
scInc    $67B50   512
scLoop   -10      6
scStop
```

This sample pseudoprogram consists of three TIB instructions that transfer six 512-byte blocks of data to or from address $67B50 (depending on whether these instructions are passed to a `SCSIRead` or a `SCSIWrite` function).

The first TIB instruction transfers a 512-byte block of data from a starting address and then increments that address by the amount of data transferred. The second TIB instruction branches back to the first (by branching back 10 bytes, which is the size of a TIB instruction), and forms a loop that is executed six times (as specified by the second parameter). The third and final TIB instruction terminates the execution sequence and returns to the calling routine.

See “Using CDB and TIB Structures,” beginning on page 3-17, for an example of how to use TIB instructions.

The Structure of Block Devices

This section describes the low-level organization of data on random-access storage devices such as SCSI hard disk drives. Although this information is presented in the context of the SCSI Manager, it applies to any type of block device that can be used by the Macintosh Operating System, regardless of the hardware interface.

There are a number of ways to address data on block-structured storage devices such as disk drives. At the lowest level, a disk drive addresses a block by its cylinder, head, and sector number. The SCSI specification, however, conceals this level of detail. Instead, each block on a SCSI disk is assigned a number, beginning with 0 and extending to the last block on the disk. The SCSI specification describes these addresses as “logical” block numbers, but the SCSI Manager calls them physical block numbers because they correspond to a fixed location on the disk.

At an even higher level of abstraction, a device driver can define the mapping of physical addresses on a device to the logical addresses of a file system. This allows file systems to be independent of the characteristics of a particular device.

In the terminology of the SCSI Manager, a ***physical block*** refers to a specific, fixed location defined by the manufacturer of a SCSI device. A ***logical block*** refers to an abstract location defined by software. A ***partition*** is a series of contiguous logical blocks that have been allocated to a particular operating system, file system, or device driver. A disk can be divided into any number of partitions. Locations within these partitions are specified using logical block numbers, which are integer values ranging from 0 to the number of blocks in the partition.

The low-level organization of block devices is defined by two data structures: the driver descriptor record and the partition map. These structures are introduced in the following sections. See “Data Structures,” beginning on page 3-23, for a complete description of the fields within these structures.

The Driver Descriptor Record

The driver descriptor record is a data structure that identifies the device drivers installed on a disk. To support multiple operating systems or other features, a disk can have more than one device driver installed, each in its own partition. The Start Manager reads the driver descriptor record during system startup and uses the information to locate and load the appropriate device driver.

The driver descriptor record is always located at physical block 0, the first block on the disk. The driver descriptor record is defined by the `Block0` data type.

```
TYPE Block0 =
PACKED RECORD
   sbSig:         Integer;    {device signature}
   sbBlkSize:     Integer;    {block size of the device}
   sbBlkCount:    LongInt;    {number of blocks on the device}
   sbDevType:     Integer;    {reserved}
   sbDevId:       Integer;    {reserved}
   sbData:        LongInt;    {reserved}
   sbDrvrCount:   Integer;    {number of driver descriptor entries}
   ddBlock:       LongInt;    {first driver’s starting block}
   ddSize:        Integer;    {size of the driver, in 512-byte blocks}
   ddType:        Integer;    {operating system type (MacOS = 1)}
   ddPad:         ARRAY [0..242] OF Integer; {additional drivers, if any}
END;
```

The driver descriptor record consists of seven fixed fields, followed by a variable amount of driver-specific information. The first field in the driver descriptor record is a signature, which must be set to the value of the `sbSIGWord` constant to indicate that the record is valid (meaning that the disk has been formatted). The second field, `sbBlkSize`, specifies the size of the blocks on the device, in bytes. The `sbBlkCount` field specifies the total number of blocks on the device. The next three fields are reserved. The `sbDrvrCount` field specifies the number of drivers that are installed on the disk. The drivers can be located anywhere on the device and can be as large as necessary.

The `ddBlock`, `ddSize`, and `ddType` fields contain information about the first device driver on the disk. Information about any additional drivers is stored in the `ddPad` field, as an array of consecutive `ddBlock`, `ddSize`, and `ddType` fields.

To select a particular device driver for loading at system startup, you use the Start Manager `SetOSDefault` function and specify a value corresponding to the `ddType` field in the driver descriptor record.

The Partition Map

The partition map is a data structure that describes the partitions present on a block device. The Macintosh Operating System and all other operating systems from Apple use the same partitioning method. This allows a single device to support multiple operating systems.

The partition map always begins at physical block 1, the second block on the disk. With the exception of the driver descriptor record in block 0, every block on a disk must belong to a partition.

Each partition on a disk is described by an entry in the partition map. The partition map is itself a partition, and contains an entry describing itself. The partition map entry for the partition map is not necessarily the first entry in the map. Partition map entries can be in any order, and need not correspond to the physical organization of partitions on the disk.

The number of entries in the partition map is not restricted. However, because the partition map must begin at block 1 and must be contiguous, it cannot easily be expanded once other partitions are created. One way around this limitation is to create a large number of empty partition map entries when the disk is initialized.

To locate a partition, the Start Manager examines the `pmMapBlkCnt` field of the first partition map entry. This field contains the size of the partition map, in blocks. Then, using the block size value from the `sbBlkSize` field of the driver descriptor record, the

Start Manager reads each block in the partition map, looking for a valid signature in the `pmSIG` field of each partition map entry record.

The partition map entry record is defined by the `Partition` data type.

```
TYPE Partition =
RECORD
   pmSig:         Integer;       {partition signature}
   pmSigPad:      Integer;       {reserved}
   pmMapBlkCnt:   LongInt;       {number of blocks in partition map}
   pmPyPartStart: LongInt;       {first physical block of partition}
   pmPartBlkCnt:  LongInt;       {number of blocks in partition}
   pmPartName:    PACKED ARRAY [0..31] OF Char; {partition name}
   pmParType:     PACKED ARRAY [0..31] OF Char; {partition type}
   pmLgDataStart: LongInt;       {first logical block of data area}
   pmDataCnt:     LongInt;       {number of blocks in data area}
   pmPartStatus:  LongInt;       {partition status information}
   pmLgBootStart: LongInt;       {first logical block of boot code}
   pmBootSize:    LongInt;       {size of boot code, in bytes}
   pmBootAddr:    LongInt;       {boot code load address}
   pmBootAddr2:   LongInt;       {reserved}
   pmBootEntry:   LongInt;       {boot code entry point}
   pmBootEntry2:  LongInt;       {reserved}
   pmBootCksum:   LongInt;       {boot code checksum}
   pmProcessor:   PACKED ARRAY [0..15] OF Char; {processor type}
   pmPad:         ARRAY [0..187] OF Integer;    {reserved}
END;
```

The first three fields in a partition map entry record are redundant, in that all entries in the partition map must contain the same values for these fields. The `pmSig` field contains the partition map signature, which is defined by the `pMapSIG` constant. The `pmSigPad` field is currently unused and must be set to 0. The `pmMapBlkCnt` field contains the size in blocks of the entire partition map. Because this value is duplicated in every entry, you can determine the size of the partition map from any entry in the map.

The remaining fields of the partition map entry record contain information about a particular disk partition. The `pmPyPartStart` field contains the physical block number of the first block of the partition. The `pmPartBlkCnt` field contains the number of blocks in the partition. The `pmPartName` field can contain an optional 32-character partition name. If this field contains a string beginning with `Maci` (for Macintosh), the Start Manager will perform checksum verification of the device driver's boot code. Otherwise, this field is ignored.

The `pmParType` field contains a string that identifies the partition type. Strings beginning with `Apple_` are reserved for use by Apple Computer, Inc. The Start Manager uses this information to identify the type of device driver or file system in a partition.

A bootable system disk must contain both an `Apple_Driver` and an `Apple_HFS` partition. See page 3-26 for a list of the standard partition types defined by Apple.

For file systems that do not begin at logical block 0 of the partition, the `pmLgDataStart` field contains the logical block number of the first block of file system data. The `pmDataCnt` field specifies the size of the data area, in blocks. The `pmPartStatus` field is currently used only by the A/UX operating system.

For device driver partitions, the `pmLgBootStart` field specifies the logical block number of the first block containing boot code. The `pmBootSize` field contains the size in bytes of the boot code. The `pmBootAddr` field specifies the memory address where the boot code is to be loaded, while the `pmBootEntry` field specifies the address to which the Start Manager will transfer control after loading the boot code into memory. The `pmBootCksum` field holds the checksum of the boot code, which the Start Manager can compare against the calculated checksum after loading the code. The `pmProcessor` field is a string that identifies the type of processor that will execute the boot code.

For more information about the startup process and SCSI devices, see the chapter "Start Manager" in *Inside Macintosh: Operating System Utilities*.

Using the SCSI Manager

Your device driver or application can use the SCSI Manager routines to transfer data to and from SCSI peripheral devices. This section begins with a simple example that illustrates the basic steps necessary to read data from a SCSI device. Next, the details of using transfer instruction blocks and command descriptor blocks are presented, followed by a complete program that uses these concepts.

Reading Data From a SCSI Device

This section shows you how to use the SCSI Manager routines to read data from a SCSI peripheral device. Your application or device driver follows these steps for reading data from a SCSI device:

1. Create a command descriptor block (CDB) and a transfer instruction block (TIB).
2. Call the `SCSIGet` function to arbitrate for the SCSI bus.
3. Use the `SCSISelect` function to select the SCSI device to read from.
4. Use the `SCSICmd` function to send a command descriptor block (CDB) containing a SCSI read command to the device.
5. Call the `SCSIRead` function to transfer the data.
6. Call the `SCSIComplete` function to get the status and message bytes that mark the end of a transaction over the SCSI bus.

Listing 3-1 shows code illustrating these steps. The example is simplified, in that it excludes the details of setting up the CDB and TIB data structures prior to initiating the read operation. That information is presented in the next section.

Listing 3-1 Reading data from a SCSI device

```
FUNCTION MyReadSCSI : OSErr;
CONST
   kCompletionTimeout = 300;  {value passed to SCSIComplete }
                              { 300 ticks = 5 seconds}
VAR
   CDB:        PACKED ARRAY [0..5] OF Byte;     {command descriptor block}
   CDBLen:     Integer;                         {length of CDB}
   TIB:        PACKED ARRAY [0..1] OF SCSIInstr;{transfer instruction block}
   scsiID:     Integer;                         {SCSI ID of the target}
   compStat:   Integer;                         {status from SCSIComplete}
   compMsg:    Integer;                         {message from SCSIComplete}
   compErr:    OSErr;                           {result from SCSIComplete}
   myErr:      OSErr;                           {cumulative error result}
BEGIN
   {Note: This example assumes the CDB, CDBLen, TIB, and scsiID variables }
   { already contain appropriate values.}
   myErr := SCSIGet;                            {arbitrate for the bus}
   IF myErr = noErr THEN
   BEGIN
      myErr := SCSISelect(scsiID);              {select the target}
      IF myErr = noErr THEN
      BEGIN
         myErr := SCSICmd(@CDB, CDBLen);        {send read command}
         IF myErr = noErr THEN
            myErr := SCSIRead(@TIB);            {polled read}
         {complete the transaction and release the bus}
         compErr := SCSIComplete(compStat, compMsg, kCompletionTimeout);
         {return the most informative error result}
         IF myErr = noErr THEN                  {if no prior errors, then }
            myErr := compErr;                   { return SCSIComplete result}
      END;
   END;
   MyReadSCSI := myErr;                         {return result code}
END;
```

The `MyReadSCSI` function follows the steps presented earlier in this section, starting with calling the `SCSIGet` and `SCSISelect` functions to select the target device, sending a read command using the `SCSICmd` function, and reading the data with the `SCSIRead` function. Finally, the `SCSIComplete` function is called to obtain the status and message bytes from the device and restore the bus to the bus free phase.

The `MyReadSCSI` function assumes these variables have already been set up properly:

- a SCSI command descriptor block (the `CDB` variable)
- an integer specifying the length of the command descriptor block (the `CDBLen` variable)
- a transfer instruction block (the `TIB` variable)
- an integer specifying the SCSI ID of the target device (the `scsiID` variable)

Within its narrowed scope, the `MyReadSCSI` function is correct and complete. You can easily modify it to handle other operations, such as writing data, or conducting blind transfers.

The `MyReadSCSI` function shows one way of handling the error results returned by a series of SCSI Manager functions. The result codes returned by the SCSI Manager functions are put into the `myErr` local variable as each SCSI Manager function is called. Your code should likewise check the result codes and proceed only if there is no error. Calling the `SCSIComplete` function is the last step, and requires special handling. Your code should call the `SCSIComplete` function even if an earlier SCSI Manager routine has returned an error, because the `SCSIComplete` function takes whatever steps are necessary to restore the SCSI bus to the bus free phase. For more information, see “Using the SCSIComplete Function” on page 3-21.

Using CDB and TIB Structures

The command descriptor block (CDB) is a data structure defined by the SCSI specification for communicating commands to SCSI devices. The SCSI Manager does not interpret the commands in a CDB, it simply transfers them to the selected device.

You send a CDB to a SCSI device using the `SCSICmd` function. The size of the CDB structure can be 6, 10, or 12 bytes, depending on the number of parameters required by the command. The first byte specifies the command, and the remaining bytes contain parameters.

The SCSI specification includes a set of standard commands that all SCSI devices must implement, and a wide range of commands for specific device types. In addition, manufacturers can define proprietary command codes for their devices. You should refer to the manufacturer’s documentation for information about the commands supported by a particular device.

You use the transfer instruction block (TIB) data structure to pass instructions to the SCSI Manager `SCSIRead`, `SCSIRBlind`, `SCSIWrite`, and `SCSIWBlind` functions. The TIB structure is defined by the `SCSIInstr` data type. The `scOpcode` field contains a transfer operation code, and the `scParam1` and `scParam2` fields contain parameters to the command. The instruction set consists of eight operation codes that allow you to transfer data, increment a counter, and form iterative loops. See "SCSI Manager TIB Instructions," beginning on page 3-27, for details of the TIB instruction set.

Listing 3-2 shows an example of how you can use CDB and TIB instructions to send a command and read information from a SCSI peripheral device. The `MySCSIInquiry` program uses the SCSI `INQUIRY` command to obtain a 256-byte record of information from a target device. This information includes the target's device type, vendor ID, product ID, revision data, and other vendor-specific information. The `INQUIRY` command is one of the standard commands that all SCSI devices must support.

Listing 3-2 Using TIB and CDB structures

```
PROGRAM MySCSIInquiry;
USES SCSI;

CONST
   kInquiryCmd = $12;         {SCSI command code for the INQUIRY command}
   kVendorIDSize = 8;         {size of the Vendor ID string}
   kProductIDSize = 16;       {size of the Product ID string}
   kRevisionSize = 4;         {size of the Revision string}
   kCompletionTimeout = 300;  {timeout value passed to SCSIComplete}
   kMySCSIID = 0;             {SCSI ID of the target device}

{This structure duplicates the format of the SCSI INQUIRY response record, }
{ as described in the SCSI-2 specification. The first 5 bytes are required }
{ for SCSI-1 devices. The first 36 bytes are required for SCSI-2 devices. }
{ The AdditionalLength field contains the length of the vendor-specific }
{ information, if any, beyond the 5 bytes required for all devices.}
TYPE MyInquiryRecord =
PACKED RECORD
   DeviceType:       Byte;    {SCSI device type code (disk, tape, etc.)}
   DeviceQualifier:  Byte;    {7-bit vendor-specific code}
   Version:          Byte;    {version of ANSI standard (SCSI-1 or SCSI-2)}
   ResponseFormat:   Byte;
   AdditionalLength: Byte;    {length of vendor-specific information}
   VendorUse1:       Byte;
   Reserved1:        Integer;
   VendorID:         PACKED ARRAY [1..kVendorIDSize] OF Char;  {manufacturer}
   ProductID:        PACKED ARRAY [1..kProductIDSize] OF Char; {product code}
   Revision:         PACKED ARRAY [1..kRevisionSize] OF Char;  {firmware rev}
   VendorUse2:       PACKED ARRAY [1..20] OF Byte;
   Reserved2:        PACKED ARRAY [1..42] OF Byte;
   VendorUse3:       PACKED ARRAY [1..158] OF Byte;
END;                          {a total of 256 bytes of data may be returned}
```

```
VAR
   CDB:        PACKED ARRAY [0..5] OF Byte;     {command descriptor block}
   TIB:        PACKED ARRAY [0..1] OF SCSIInstr; {transfer instruction block}
   Response:   MyInquiryRecord;  {holds target’s response}
   compStat:   Integer;          {status information from SCSIComplete}
   compMsg:    Integer;          {message information from SCSIComplete}
   compErr:    OSErr;            {result from SCSIComplete}
   myErr:      OSErr;            {error result}
   i:          Integer;          {loop counter}
BEGIN
   {Set up the command buffer with the SCSI INQUIRY command.}
   CDB[0] := kInquiryCmd;     {SCSI command code for the INQUIRY command}
   CDB[1] := 0;               {unused parameter}
   CDB[2] := 0;               {unused parameter}
   CDB[3] := 0;               {unused parameter}
   CDB[4] := 5;               {maximum number of bytes target should return}
   CDB[5] := 0;               {unused parameter}

   {Set up the two TIB structures; one to read, the other as terminator.}
   TIB[0].scOpcode := scNoInc;            {specify the scNoInc instruction}
   TIB[0].scParam1 := LongInt(@Response); {pointer to buffer}
   TIB[0].scParam2 := 5;                  {number of bytes to move}
   TIB[1].scOpcode := scStop;             {specify the scStop instruction}
   TIB[1].scParam1 := LongInt(NIL);       {unused parameter}
   TIB[1].scParam2 := LongInt(NIL);       {unused parameter}

   WRITELN('SCSI inquiry example. Testing SCSI ID:', kMySCSIID);

   {Send the INQUIRY command twice. The first time to obtain the }
   { AdditionalLength value in the fifth byte of the INQUIRY response }
   { record and the second time to read that additional amount. Notice }
   { that SCSIComplete is always called if SCSISelect was successful.}
   FOR i := 1 to 2 DO
   BEGIN
      myErr := SCSIGet;                   {arbitrate for the bus}
      IF myErr = noErr THEN
         myErr := SCSISelect(kMySCSIID);  {select the target}
      IF myErr <> noErr THEN
      BEGIN
         WRITELN('Error result from SCSIGet or SCSISelect:', myErr);
         EXIT(MySCSIInquiry);
      END;
      myErr := SCSICmd(@CDB, 6);    {send INQUIRY command to the target}
      IF myErr = noErr THEN
      BEGIN
         myErr := SCSIRead(@TIB);   {read the INQUIRY response record}
         IF myErr = noErr THEN      {if there was no error, and }
            IF i = 1 THEN           { if this is the first time through }
            BEGIN                   { the loop, get the AdditionalLength}
               CDB[4] := CDB[4] + Response.AdditionalLength;
               TIB[0].scParam2 := TIB[0].scParam2 +
                                  Response.AdditionalLength;
            END;
      END;

      {Call SCSIComplete to clean up. Results are ignored in this example.}
      compErr := SCSIComplete(compStat, compMsg, kCompletionTimeout);
      IF myErr <> noErr THEN
      BEGIN
         WRITELN('Error result from SCSICmd or SCSIRead:', myErr);
         EXIT(MySCSIInquiry);
      END;
   END;  {FOR loop}

   {Display the information.}
   IF Response.AdditionalLength > 0 THEN
   BEGIN
      WITH Response DO
      BEGIN
         WRITE('VendorID:');
         FOR i := 1 TO kVendorIDSize DO
            WRITE(VendorID[i]);
         WRITELN;
         WRITE('ProductID:');
         FOR i := 1 TO kProductIDSize DO
            WRITE(ProductID[i]);
         WRITELN;
         WRITE('Revision:');
         FOR i := 1 TO kRevisionSize DO
            WRITE(Revision[i]);
         WRITELN;
      END;
   END;
END.
```

The MySCSIInquiry program first defines various constants, including the kInquiryCmd constant, which contains the operation code for the SCSI INQUIRY command. Next the MyInquiryRecord data type is declared, a 256-byte structure that holds the information returned by the target. The fields of this record are based on the SCSI-2 specification. The SCSI-1 specification requires that devices return at least the first 5 bytes of information (DeviceType through AdditionalLength), however, many SCSI-1 devices and all SCSI-2 devices return at least the first 36 bytes (DeviceType through Revision).

In the 6-byte CDB used by the SCSI INQUIRY command, the first byte contains the operation code and the fifth byte specifies the maximum number of bytes the target is allowed to send in response to the inquiry. Restricting the target's response to a specified number of bytes prevents it from overflowing the buffer the initiator has set aside to accept the data.

This program uses two transfer instruction blocks, both of which are relatively simple. The first TIB is an scNoInc instruction, whose parameters specify a data transfer into the Response record. The second TIB is an scStop instruction, which terminates the SCSI Manager processing that occurs inside the SCSIRead function.

The body of the MySCSIInquiry program consists of a loop that performs the arbitrate/select/command/transfer/complete sequence described in "Reading Data From a SCSI Device" on page 3-15. The loop executes this sequence of SCSI Manager functions twice. The first time sends the SCSI INQUIRY command to the target and requests only the standard 5 bytes of information supplied by all SCSI devices. The value of the fifth byte (returned in the AdditionalLength field of the Response record) indicates the amount of additional information the device is capable of returning. Before going through the loop a second time, both the CDB and the TIB are modified to reflect the additional size of the inquiry information.

The program checks for errors at each stage in the SCSI Manager calling sequence. If either the SCSIGet or SCSISelect function returns an error, the program exits. If the SCSICmd function returns an error, SCSIRead is not called. To complete the transaction and release the bus, the SCSIComplete function is always called if SCSISelect was successful.

Using the SCSIComplete Function

The SCSIComplete function completes a SCSI transaction and restores the bus to the bus free phase. You must call this function at the end of every transaction that proceeds past the selection phase, even if the transaction does not complete successfully.

The SCSIComplete function waits a specified number of ticks for the current transaction to complete, and then returns one byte of status information and one byte of message information from the target device. The function returns one of the following result codes:

- noErr. The SCSIComplete function was able to obtain both the status and message bytes successfully. This result code indicates that the information is valid.

- `scComplPhaseErr`. Upon entry, the `SCSIComplete` function detected that the target was ready to transfer information (that is, the /REQ signal was asserted) but the SCSI bus was not in the status phase. The SCSI Manager performed corrective action to bring the bus into the status phase. For example, accepting bytes from the target without passing them to your program ("bit-bucketing"), or sending an arbitrary number of bytes to the target. Once in status phase, the `SCSIComplete` function was able to transfer the status and message bytes successfully, and this information is valid.
- `scPhaseErr`. The `SCSIComplete` function could not force the SCSI bus into the status phase. The status and message bytes should be considered invalid. You may need to reset the bus to restore proper operation.
- `scCommErr`. This result code covers any other error conditions encountered by the `SCSIComplete` function, such as the timeout that occurs if the transaction does not complete within the specified number of ticks.

Choosing Polled or Blind Transfers

The SCSI Manager supports two data transfer methods: polled and blind. During a ***polled transfer***, the SCSI Manager senses the state of the Macintosh SCSI controller hardware to determine when the controller is ready to transfer another byte. In a ***blind transfer***, the SCSI Manager assumes that the SCSI controller (and the target device) can keep up with a specified transfer rate, and does not explicitly sense whether the hardware is ready.

Note
These transfer modes are specific to the Macintosh SCSI interface hardware implementation and are not part of the SCSI protocol. ◆

When the SCSI Manager retrieves data from the SCSI controller, it can explicitly verify that a byte was received by the controller and is ready for transfer. The SCSI Manager does this by polling a status register in the controller. Alternatively, the SCSI Manager can assume that a byte is available and can attempt to read it without checking first. As long as a SCSI device can supply data to the SCSI controller faster than the SCSI Manager can retrieve it, blind transfers work reliably. If the SCSI device cannot keep up, timeout errors and other problems can occur.

For example, in the Macintosh Plus (the first model to include a SCSI interface), if the SCSI Manager reads a byte from the SCSI controller chip before the chip receives a byte from the target, the read operation completes but the data is invalid. The `SCSIComplete` function does not always return an error result in this case.

Newer Macintosh models include hardware support for handshaking, allowing blind transfers to be both fast and reliable. This handshaking allows the SCSI controller to defer the CPU if no data is available to transfer. If the data doesn't arrive within a specified period, the SCSI Manager returns the `scBusTOErr` result. The timeout period varies for each Macintosh model. This type of timeout error does not occur when using polled transfers.

Polled transfers work reliably with all SCSI peripheral devices, and are a good choice for slow or unpredictable devices such as printers and scanners. You should also use polled transfers if you are unfamiliar with the characteristics of a particular device. You use the `SCSIRead` and `SCSIWrite` functions to initiate polled transfers.

For disk drives and other high-speed devices, blind transfers can significantly increase data throughput. As long as the device does not incur any delays during a transfer, or the delays occur at predictable times, blind transfers are a good choice. You use the `SCSIRBlind` and `SCSIWBlind` functions to initiate blind transfers.

Because the first byte transferred by each TIB instruction is always polled, even in blind mode, you can work around predictable delays using an appropriate sequence of TIB instructions. For example, if a peripheral device always pauses at a specific byte within a transfer, you can divide the transfer into blocks so that the delayed byte is located at the start of a TIB instruction. The SCSI Manager polls the controller before the first byte, then reads the remaining bytes using a blind transfer. For disk drives, predictable delays generally occur at sector boundaries, so you can compensate by dividing your transfers into sector-sized blocks.

SCSI Manager Reference

This section describes the data structures and routines that constitute the SCSI Manager, and also includes the data structures that describe the low-level structure of block devices.

The section "SCSI Manager TIB Instructions," beginning on page 3-27, contains descriptions of transfer instruction block (TIB) instructions. These structures are used to control data transfers conducted by the SCSI Manager. Although TIB instructions are data structures, not machine-executable code, they are analogous to code in that TIB instructions are interpreted and executed by the SCSI Manager. Because of this dual nature, TIB instructions are presented in their own section.

Data Structures

This section describes the driver descriptor record and the partition map entry record. These data structures are not used by the SCSI Manager, but represent the way data is structured on random access storage devices such as hard disk drives. The Start Manager uses this information to locate partitions and device drivers on SCSI disks.

Driver Descriptor Record

The driver descriptor record contains information about the device drivers resident on a SCSI peripheral device. The driver descriptor record is defined by the `Block0` data type.

```
TYPE Block0 =
PACKED RECORD
   sbSig:         Integer;    {device signature}
   sbBlkSize:     Integer;    {block size of the device}
   sbBlkCount:    LongInt;    {number of blocks on the device}
   sbDevType:     Integer;    {reserved}
   sbDevId:       Integer;    {reserved}
   sbData:        LongInt;    {reserved}
   sbDrvrCount:   Integer;    {number of driver descriptor entries}
   ddBlock:       LongInt;    {first driver’s starting block}
   ddSize:        Integer;    {size of the driver, in 512-byte blocks}
   ddType:        Integer;    {operating system type (MacOS = 1)}
   ddPad:         ARRAY [0..242] OF Integer; {additional drivers, if any}
END;
```

Field descriptions

`sbSig`	The device signature. This field should contain the value of the `sbSIGWord` constant ($4552) to indicate that the driver descriptor record is valid (meaning that the disk has been formatted).
`sbBlkSize`	The size of the blocks on the device, in bytes.
`sbBlkCount`	The number of blocks on the device.
`sbDevType`	Reserved.
`sbDevId`	Reserved.
`sbData`	Reserved.
`sbDrvrCount`	The number of drivers installed on the disk. More than one driver may be included when multiple operating systems or processors are supported. The drivers can be located anywhere on the device and can be as large as necessary.
`ddBlock`	The physical block number of the first block of the first device driver on the disk.
`ddSize`	The size of the device driver, in 512-byte blocks.
`ddType`	The operating system or processor supported by the driver. A value of 1 specifies the Macintosh Operating System. The values 0 through 15 are reserved for use by Apple Computer, Inc.
`ddPad`	Additional `ddBlock`, `ddSize`, and `ddType` entries for other device drivers on the disk.

If multiple device drivers exist on the device, you can use the Start Manager `SetOSDefault` function to control which operating system is loaded at startup by specifying a value that corresponds to the `ddType` field of the appropriate device driver. For more information on the startup process, see the chapter “Start Manager” in *Inside Macintosh: Operating System Utilities*.

See “The Structure of Block Devices,” beginning on page 3-12, for more information about this data structure.

Partition Map Entry Record

The partition map entry record contains information about how data is stored on a block device, usually a SCSI disk drive. The partition map entry record is defined by the `Partition` data type.

```
TYPE Partition =
RECORD
   pmSig:         Integer;       {partition signature}
   pmSigPad:      Integer;       {reserved}
   pmMapBlkCnt:   LongInt;       {number of blocks in partition map}
   pmPyPartStart: LongInt;       {first physical block of partition}
   pmPartBlkCnt:  LongInt;       {number of blocks in partition}
   pmPartName:    PACKED ARRAY [0..31] OF Char; {partition name}
   pmParType:     PACKED ARRAY [0..31] OF Char; {partition type}
   pmLgDataStart: LongInt;       {first logical block of data area}
   pmDataCnt:     LongInt;       {number of blocks in data area}
   pmPartStatus:  LongInt;       {partition status information}
   pmLgBootStart: LongInt;       {first logical block of boot code}
   pmBootSize:    LongInt;       {size of boot code, in bytes}
   pmBootAddr:    LongInt;       {boot code load address}
   pmBootAddr2:   LongInt;       {reserved}
   pmBootEntry:   LongInt;       {boot code entry point}
   pmBootEntry2:  LongInt;       {reserved}
   pmBootCksum:   LongInt;       {boot code checksum}
   pmProcessor:   PACKED ARRAY [0..15] OF Char; {processor type}
   pmPad:         ARRAY [0..187] OF Integer;    {reserved}
END;
```

Field descriptions

`pmSig`	The partition signature. This field should contain the value of the `pMapSIG` constant ($504D). An earlier but still supported version uses the value $5453.
`pmSigPad`	Reserved.
`pmMapBlkCnt`	The size of the partition map, in blocks.
`pmPyPartStart`	The physical block number of the first block of the partition.
`pmPartBlkCnt`	The size of the partition, in blocks.
`pmPartName`	An optional partition name, up to 32 bytes in length. If the string is less than 32 bytes, it must be terminated with the ASCII NUL character (a byte with a value of 0). If the partition name begins with `Maci` (for Macintosh), the Start Manager will perform checksum verification of the device driver's boot code. Otherwise, this field is ignored.

`pmParType` A string that identifies the partition type. Names that begin with `Apple_` are reserved for use by Apple Computer, Inc. Names shorter than 32 characters must be terminated with the NUL character. The following standard partition types are defined for the `pmParType` field:

String	Meaning
`Apple_partition_map`	Partition contains a partition map
`Apple_Driver`	Partition contains a device driver
`Apple_Driver43`	Partition contains a SCSI Manager 4.3 device driver
`Apple_MFS`	Partition uses the original Macintosh File System (64K ROM version)
`Apple_HFS`	Partition uses the Hierarchical File System implemented in 128K and later ROM versions
`Apple_Unix_SVR2`	Partition uses the Unix file system
`Apple_PRODOS`	Partition uses the ProDOS file system
`Apple_Free`	Partition is unused
`Apple_Scratch`	Partition is empty

`pmLgDataStart` The logical block number of the first block containing file system data. This is for use by operating systems, such as A/UX, in which the file system does not begin at logical block 0 of the partition.

`pmDataCnt` The size of the file system data area, in blocks. This is used in conjunction with the `pmLgDataStart` field, for those operating systems in which the file system does not begin at logical block 0 of the partition.

`pmPartStatus` Two words of status information about the partition. The low-order byte of the low-order word contains status information used only by the A/UX operating system:

Bit	Meaning
0	Set if a valid partition map entry
1	Set if partition is already allocated; clear if available
2	Set if partition is in use; may be cleared after a system reset
3	Set if partition contains valid boot information
4	Set if partition allows reading
5	Set if partition allows writing
6	Set if boot code is position-independent
7	Unused

The remaining bytes of the `pmPartStatus` field are reserved.

`pmLgBootStart` The logical block number of the first block containing boot code.

`pmBootSize` The size of the boot code, in bytes.

`pmBootAddr`	The memory address where the boot code is to be loaded.
`pmBootAddr2`	Reserved.
`pmBootEntry`	The memory address to which the Start Manager will transfer control after loading the boot code into memory.
`pmBootEntry2`	Reserved.
`pmBootCksum`	The boot code checksum. The Start Manager can compare this value against the calculated checksum after loading the code.
`pmProcessor`	An optional string that identifies the type of processor that will execute the boot code. Strings shorter than 16 bytes must be terminated with the ASCII NUL character. The following processor types are defined: `68000`, `68020`, `68030`, and `68040`.
`pmPad`	Reserved.

See "The Structure of Block Devices," beginning on page 3-12, for more information about this data structure.

SCSI Manager TIB Instructions

The transfer instruction block (TIB) is a data structure that you use to control the data transfer process. TIB structures are passed as parameters to the `SCSIRead`, `SCSIRBlind`, `SCSIWrite`, and `SCSIWBlind` functions. The transfer instruction block is defined by the `SCSIInstr` data type.

```
TYPE SCSIInstr =
RECORD
   scOpcode:         Integer;    {operation code}
   scParam1:         LongInt;    {first parameter}
   scParam2:         LongInt;    {second parameter}
END;
```

The `scOpcode` field contains a value that specifies the operation to be performed. There are eight possible operations, known as TIB instructions, which carry out tasks such as moving data, looping, and address arithmetic. These instructions are described in this section. The operation codes for the TIB instructions are:

```
CONST
   scInc      = 1;     {transfer data, increment buffer pointer}
   scNoInc    = 2;     {transfer data, don’t increment pointer}
   scAdd      = 3;     {add long to address}
   scMove     = 4;     {move long to address}
   scLoop     = 5;     {decrement counter and loop if > 0}
   scNop      = 6;     {no operation}
   scStop     = 7;     {stop TIB execution}
   scComp     = 8;     {compare SCSI data with memory}
```

To transfer data, you create a variable-length array of TIB instructions and pass a pointer to this array to any of the SCSI Manager data transfer functions (`SCSIRead`, `SCSIRBlind`, `SCSIWrite`, `SCSIWBlind`). These SCSI Manager functions interpret the TIB instructions and carry out the requested operations.

For an example of how to use TIB instructions, see "Using CDB and TIB Structures," beginning on page 3-17.

IMPORTANT

Before you call any of the SCSI Manager data transfer functions (`SCSIRead`, `SCSIRBlind`, `SCSIWrite`, or `SCSIWBlind`), you must first send a SCSI read or write command to the target using the `SCSICmd` function. ▲

scInc

You can use the `scInc` TIB instruction to transfer data and increment the buffer pointer.

Parameter block

→	`scParam1`	`Ptr`	A pointer to a data buffer.
→	`scParam2`	`LongInt`	The number of bytes to be transferred.

DESCRIPTION

The `scInc` instruction moves data to or from the buffer pointed to by `scParam1`. You specify the number of bytes to be transferred in `scParam2`. The buffer pointer in `scParam1` is incremented by the number of bytes transferred (for use by a subsequent iteration of this instruction).

scNoInc

You can use the `scNoInc` TIB instruction to transfer data without incrementing the buffer pointer.

Parameter block

→	`scParam1`	`Ptr`	A pointer to a data buffer.
→	`scParam2`	`LongInt`	The number of bytes to be transferred.

DESCRIPTION

The `scNoInc` instruction moves data to or from the buffer pointed to by `scParam1`. You specify the number of bytes to be transferred in `scParam2`. The buffer pointer in `scParam1` is unmodified by this instruction.

scAdd

You can use the `scAdd` TIB instruction to add a value to an address.

Parameter block

→	`scParam1`	`Ptr`	An address.
→	`scParam2`	`LongInt`	The number to add to the address.

DESCRIPTION

The `scAdd` instruction adds the long value in `scParam2` to the address in `scParam1`.

scMove

You can use the `scMove` TIB instruction to copy a long value from one memory location to another.

Parameter block

→	`scParam1`	`Ptr`	The source address.
→	`scParam2`	`Ptr`	The destination address.

DESCRIPTION

The `scMove` TIB instruction copies the 32-bit value pointed to by the `scParam1` parameter to the memory location specified by the `scParam2` parameter.

scLoop

You can use the `scLoop` TIB instruction to repeat a sequence of TIB instructions a specified number of times.

Parameter block

→	`scParam1`	`LongInt`	The relative offset of the TIB instruction to branch to.
→	`scParam2`	`LongInt`	The number of times to loop.

DESCRIPTION

The `scLoop` TIB instruction decrements the value in `scParam2` by 1. If the result is greater than 0, the flow of control branches to the TIB instruction whose relative offset is the current instruction plus the value in `scParam1`. If the result is 0, control passes to the instruction following the `scLoop` instruction. The offset in `scParam1` is a signed value, and must be a multiple of 10 bytes (the size of the `SCSIInstr` data type). For example, to branch to the instruction immediately preceding the current one, you would specify a relative offset of –10. To jump ahead three instructions, you would specify a relative offset of 30.

scNop

The `scNop` TIB instruction does nothing.

DESCRIPTION

The `scNop` TIB instruction is analogous to an assembly-language `NOP` instruction. The two parameters are ignored.

scStop

You use the `scStop` TIB instruction to end a sequence of TIB instructions.

DESCRIPTION

The `scStop` TIB instruction stops execution of a sequence of TIB instructions and returns control to the calling SCSI Manager function. At least one `scStop` instruction is required in any TIB instruction sequence, usually at the end. The two parameters are ignored.

scComp

You can use the `scComp` TIB instruction to compare data on a SCSI device with data in memory.

Parameter block

→	`scParam1`	`Ptr`	A pointer to a data buffer.
→	`scParam2`	`LongInt`	The number of bytes to be compared.

DESCRIPTION

The `scComp` TIB instruction is used in conjunction with the `SCSIRead` function to compare data in memory with incoming data from a SCSI device. The SCSI Manager compares the result of the read command with the contents of the data buffer pointed to by `scParam1`. The `scParam2` parameter specifies the number of bytes to read and compare. If all bytes do not compare, the `SCSIRead` function returns the result code `scCompareErr`.

SCSI Manager 4.3 Note

You should avoid using the `scComp` TIB instruction because it is not supported by SCSI Manager 4.3. ◆

SCSI Manager Routines

This section describes the SCSI Manager routines you use to

- reset the SCSI bus
- arbitrate for the SCSI bus
- select a SCSI device
- send SCSI commands and messages
- read or write data to SCSI devices
- obtain the status of the SCSI bus
- complete the processing of a SCSI transaction

SCSIReset

You can use the `SCSIReset` function to reset all devices on the SCSI bus.

```
FUNCTION SCSIReset: OSErr;
```

DESCRIPTION

The `SCSIReset` function directs the SCSI controller chip (or equivalent hardware) in the Macintosh computer to assert the SCSI bus reset signal. The reset signal causes all devices on the bus to clear pending I/O and forces the bus into the bus free phase.

▲ **WARNING**

The `SCSIReset` function interrupts SCSI communications and can cause data loss. Use this function only in exceptional circumstances. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIReset` are

Trap macro	Selector
`_SCSIDispatch`	$0000

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout

SEE ALSO

See “SCSI Bus Signals,” beginning on page 3-4, and “SCSI Bus Phases,” beginning on page 3-5, for more information about the reset signal and the bus free phase.

SCSIGet

You use the `SCSIGet` function to arbitrate for control of the SCSI bus.

```
FUNCTION SCSIGet: OSErr;
```

DESCRIPTION

The `SCSIGet` function prepares the SCSI Manager to initiate the arbitration sequence. If the SCSI Manager is busy with another operation, this function returns the `scMgrBusyErr` result. If arbitration failed because the bus was busy, the function returns the `scArbNBErr` result.

IMPORTANT

The operation of the `SCSIGet` function varies on different Macintosh models and does not necessarily initiate the SCSI bus arbitration phase. In some Macintosh models, the arbitration phase does not occur until your program calls the `SCSISelect` function. However, your program must always call the `SCSIGet` function before calling `SCSISelect`. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIGet` are

Trap macro	Selector
`_SCSIDispatch`	$0001

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scArbNBErr`	3	Bus busy, arbitration timeout
`scMgrBusyErr`	7	SCSI Manager busy

SEE ALSO

See “SCSI Bus Phases,” beginning on page 3-5, for a description of the arbitration phase.

SCSISelect

You use `SCSISelect` function to select a SCSI device for a subsequent operation.

```
FUNCTION SCSISelect (targetID: Integer): OSErr;
```

`targetID` The SCSI ID of the target device, with a value from 0 to 7.

DESCRIPTION

The `SCSISelect` function selects the SCSI device identified by the `targetID` value.

IMPORTANT
You must call the `SCSIGet` function before calling `SCSISelect`. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSISelect` are

Trap macro	Selector
`_SCSIDispatch`	$0002

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scArbNBErr`	3	Bus busy, arbitration timeout
`scSequenceErr`	8	Attempted operation is out of sequence

SEE ALSO

See “SCSI Bus Phases,” beginning on page 3-5, for a description of the selection phase.

SCSISelAtn

You can use the `SCSISelAtn` function to select a SCSI device and at the same time to assert the attention (/ATN) bus signal.

```
FUNCTION SCSISelAtn (targetID: Integer): OSErr;
```

`targetID` The SCSI ID of the target device, with a value from 0 to 6.

DESCRIPTION

The `SCSISelAtn` function is identical to the `SCSISelect` function except that this function asserts the /ATN signal during selection. The /ATN signal informs the target that the initiator wants to send a message. The `SCSISelAtn` function must be followed by a call to the `SCSIMsgOut` function to send the message to the target device.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSISelAtn` are

Trap macro	**Selector**
`_SCSIDispatch`	$000B

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout

SEE ALSO

See "SCSI Bus Signals," beginning on page 3-4, and "SCSI Bus Phases," beginning on page 3-5, for more information about the attention signal and the selection phase.

SCSICmd

You use the `SCSICmd` function to send a SCSI command to a SCSI device.

```
FUNCTION SCSICmd (buffer: Ptr; count: Integer): OSErr;
```

`buffer`	A pointer to a buffer containing the SCSI command descriptor block.
`count`	The size of the command descriptor block, in bytes.

DESCRIPTION

The `SCSICmd` function sends a SCSI command to the previously selected target device. The command code and other parameters are contained in a command descriptor block (CDB) data structure pointed to by the `buffer` parameter. The `count` parameter specifies the size of the CDB structure, which can be 6, 10, or 12 bytes.

The SCSI specification describes the CDB data structure and lists the standard SCSI commands that all devices must support. Devices may support additional commands not defined by the SCSI specification.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSICmd` are

Trap macro	**Selector**
`_SCSIDispatch`	$0003

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scPhaseErr`	5	Phase error on the SCSI bus

SEE ALSO

See "SCSI Commands," beginning on page 3-7, for an overview of SCSI commands. Refer to the SCSI specification for detailed information about SCSI commands.

SCSIMsgIn

You can use the `SCSIMsgIn` function to receive a message from a SCSI device.

```
FUNCTION SCSIMsgIn (VAR message: Integer): OSErr;
```

`message`	The low-order byte contains the message from the target device.

DESCRIPTION

The `SCSIMsgIn` function receives a SCSI message from the previously selected target device. The message is returned in the low-order byte of the `message` parameter. See the SCSI specification for information about the types of messages that can be sent from a target to an initiator.

The `SCSIMsgIn` function leaves the attention bus signal undisturbed if it is already asserted.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIMsgIn` are

Trap macro	**Selector**
`_SCSIDispatch`	$000C

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scPhaseErr`	5	Phase error on the SCSI bus

SEE ALSO

See "SCSI Messages," beginning on page 3-7, for an overview of SCSI messages. Refer to the SCSI specification for detailed information about SCSI messages.

SCSIMsgOut

You can use the `SCSIMsgOut` function to send a message to a SCSI device.

```
FUNCTION SCSIMsgOut (message: Integer): OSErr;
```

`message`	The low-order byte contains the message to be sent to the target device.

DESCRIPTION

The `SCSIMsgOut` function sends a SCSI message to the previously selected target device. The message is contained in the low-order byte of the `message` parameter. See the SCSI specification for information about the types of messages that can be sent from an initiator to a target.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIMsgOut` are

Trap macro	**Selector**
`_SCSIDispatch`	$000D

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scPhaseErr`	5	Phase error on the SCSI bus

SEE ALSO

See “SCSI Messages,” beginning on page 3-7, for an overview of SCSI messages. Refer to the SCSI specification for detailed information about SCSI messages.

SCSIRead

You can use the `SCSIRead` function to read data from a SCSI device using a polled transfer.

```
FUNCTION SCSIRead (tibPtr: Ptr): OSErr;
```

`tibPtr`	A pointer to an array of TIB instructions.

DESCRIPTION

The `SCSIRead` function reads data from the previously selected target device. The data transfer instructions are specified by the TIB array pointed to by the `tibPtr` parameter.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIRead` are

Trap macro	Selector
`_SCSIDispatch`	$0005

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scBadParmsErr`	4	Unrecognized TIB instruction
`scPhaseErr`	5	Phase error on the SCSI bus
`scCompareErr`	6	Comparison error from `scComp` instruction

SEE ALSO

See "Using CDB and TIB Structures," beginning on page 3-17, for information about using TIB instructions. See "SCSI Manager TIB Instructions," beginning on page 3-27, for details of the TIB instruction set.

SCSIRBlind

You can use the `SCSIRBlind` function to read data from a SCSI device using a blind transfer.

```
FUNCTION SCSIRBlind (tibPtr: Ptr): OSErr;
```

`tibPtr`	A pointer to an array of TIB instructions.

DESCRIPTION

The `SCSIRBlind` function is identical to the `SCSIRead` function but does not poll the SCSI controller before transferring each byte of data. The SCSI controller is polled only for the first byte transferred by each `scInc`, `scNoInc`, or `scComp` TIB instruction.

SPECIAL CONSIDERATIONS

You should use this function only if the device you are reading from is capable of transferring data fast enough to avoid timeout errors from the SCSI controller.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIRBlind` are

Trap macro	Selector
`_SCSIDispatch`	$0008

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scBadParmsErr`	4	Unrecognized TIB instruction
`scPhaseErr`	5	Phase error on the SCSI bus
`scCompareErr`	6	Comparison error from `scComp` instruction
`scBusTOErr`	9	Bus timeout during blind transfer

SEE ALSO

See the description of the `SCSIRead` function on page 3-36 for information about performing a polled transfer. See "Choosing Polled or Blind Transfers," beginning on page 3-22, for additional information.

SCSIWrite

You can use the `SCSIWrite` function to write to a SCSI device using a polled transfer.

```
FUNCTION SCSIWrite (tibPtr: Ptr): OSErr;
```

`tibPtr`	A pointer to an array of TIB instructions.

DESCRIPTION

The `SCSIWrite` function transfers data to the previously selected target device. The data transfer instructions are specified by the TIB array pointed to by the `tibPtr` parameter.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIWrite` are

Trap macro	**Selector**
`_SCSIDispatch`	$0006

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scBadParmsErr`	4	Unrecognized TIB instruction
`scPhaseErr`	5	Phase error on the SCSI bus

SEE ALSO

See "Using CDB and TIB Structures," beginning on page 3-17, for information about using TIB instructions. See "SCSI Manager TIB Instructions," beginning on page 3-27, for details of the TIB instruction set.

SCSIWBlind

You can use the `SCSIWBlind` function to write to a SCSI device using a blind transfer.

```
FUNCTION SCSIWBlind (tibPtr: Ptr): OSErr;
```

`tibPtr`	A pointer to an array of TIB instructions.

DESCRIPTION

The `SCSIWBlind` function is identical to the `SCSIWrite` function but does not poll the SCSI controller before transferring each byte of data. The SCSI controller is polled only for the first byte transferred by each `scInc`, `scNoInc`, or `scComp` TIB instruction.

SPECIAL CONSIDERATIONS

You should use this function only if the device you are writing to is capable of accepting data fast enough to avoid timeout errors from the SCSI controller.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIWBlind` are

Trap macro	Selector
`_SCSIDispatch`	$0009

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scBadParmsErr`	4	Unrecognized TIB instruction
`scPhaseErr`	5	Phase error on the SCSI bus
`scBusTOErr`	9	Bus timeout during blind transfer

SEE ALSO

See the description of the `SCSIWrite` function on page 3-38 for information about performing a polled transfer. See “Choosing Polled or Blind Transfers,” beginning on page 3-22, for additional information.

SCSIComplete

You use the `SCSIComplete` function to complete a SCSI transaction.

```
FUNCTION SCSIComplete(VAR stat: Integer; VAR message: Integer;
                      wait: LongInt): OSErr;
```

`stat`	The low-order byte contains the status byte from the target device.
`message`	The low-order byte contains the message byte from the target device.
`wait`	The number of ticks to wait for the command to complete.

DESCRIPTION

The `SCSIComplete` function performs the tasks necessary to properly complete the current SCSI transaction and leave the bus in the bus free phase. This function must be called at the end of each SCSI transaction, even if the transaction does not complete successfully.

The `SCSIComplete` function waits for the transaction to complete, and then returns one byte of status information and one byte of message information. If the transaction fails to complete within the number of ticks specified by the `wait` parameter, the `scCommErr` result is returned.

The `SCSIComplete` function uses a number of strategies to correct anomalous conditions on the SCSI bus and restore the bus into a known state. These include accepting arbitrary amounts of data sent by the target (and throwing this data away), and sending arbitrary data (bytes with the value of $EE) as requested by the target. The function returns the `scComplPhaseErr` result if either of these steps were necessary.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIComplete` are

Trap macro	**Selector**
`_SCSIDispatch`	$0004

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scPhaseErr`	5	Phase error on the SCSI bus
`scComplPhaseErr`	10	SCSI bus was not in status phase on entry to `SCSIComplete`

SEE ALSO

See “Using the SCSIComplete Function,” beginning on page 3-21, for more information about this function.

SCSIStat

You can use the `SCSIStat` function to obtain status information from the SCSI Manager.

```
FUNCTION SCSIStat: Integer;
```

DESCRIPTION

The `SCSIStat` function returns a 16-bit value containing status information. This information includes the state of all SCSI bus control signals as well as the status of the NCR 5380 SCSI controller chip (or equivalent hardware). In Macintosh models that use other SCSI controller hardware, the status information conforms to the 5380 format, but may not represent the actual state of the hardware.

IMPORTANT
Because hardware differences make it difficult to accurately interpret the status information, use of this function is not recommended. ▲

Bits 0 through 9 represent the state of the SCSI bus signals, and bits 10 through 15 report status information from the SCSI controller hardware. The status bits have these meanings:

Bit	Name	Meaning
0	DBP	Data parity signal
1	/SEL	Select signal
2	/I/O	I/O signal
3	/C/D	Command/Data signal
4	/MSG	Message signal
5	/REQ	Request signal
6	/BSY	Busy signal
7	/RST	Reset signal
8	/ACK	Acknowledge signal
9	/ATN	Attention signal
10	BSY ERR	Busy error
11	PHS MAT	Phase match
12	INT REQ	Interrupt request
13	PTY ERR	Parity error
14	DMA REQ	Direct memory access request
15	END DMA	Direct memory access complete

Note
The SCSI bus control signals are active low; therefore, the status bits represent the complement of the bus signals. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SCSIStat` are

Trap macro	**Selector**
`_SCSIDispatch`	$000A

RESULT CODES

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scPhaseErr`	5	Phase error on the SCSI bus

SEE ALSO

See “SCSI Bus Signals,” beginning on page 3-4, for an overview of SCSI bus signals. Refer to the SCSI specification for detailed information about SCSI bus signals. Refer to the NCR 5380 SCSI controller specification for information about that device.

Summary of the SCSI Manager

Pascal Summary

Constants

```
CONST
   scInc       = 1;     {transfer data, increment buffer pointer}
   scNoInc     = 2;     {transfer data, don’t increment pointer}
   scAdd       = 3;     {add long to address}
   scMove      = 4;     {move long to address}
   scLoop      = 5;     {decrement counter and loop if > 0}
   scNop       = 6;     {no operation}
   scStop      = 7;     {stop TIB execution}
   scComp      = 8;     {compare SCSI data with memory}

   {signature values}
   sbSIGWord   = $4552;    {driver descriptor map signature}
   pMapSIG     = $504D;    {partition map signature}
```

Data Types

```
TYPE SCSIInstr =
   RECORD
      scOpcode:         Integer;    {operation code}
      scParam1:         LongInt;    {first parameter}
      scParam2:         LongInt;    {second parameter}
   END;

   Block0 =
   PACKED RECORD
      sbSig:         Integer;    {device signature}
      sbBlkSize:     Integer;    {block size of the device}
      sbBlkCount:    LongInt;    {number of blocks on the device}
      sbDevType:     Integer;    {reserved}
      sbDevId:       Integer;    {reserved}
      sbData:        LongInt;    {reserved}
      sbDrvrCount:   Integer;    {number of driver descriptor entries}
      ddBlock:       LongInt;    {first driver’s starting block}
      ddSize:        Integer;    {size of the driver, in 512-byte blocks}
      ddType:         Integer;    {operating system type (MacOS = 1)}
      ddPad:          ARRAY [0..242] OF Integer; {additional drivers, if any}
   END;

   Partition =
   RECORD
      pmSig:          Integer;    {partition signature}
      pmSigPad:       Integer;    {reserved}
      pmMapBlkCnt:    LongInt;    {number of blocks in partition map}
      pmPyPartStart: LongInt;     {first physical block of partition}
      pmPartBlkCnt:  LongInt;     {number of blocks in partition}
      pmPartName:     PACKED ARRAY [0..31] OF Char; {partition name}
      pmParType:      PACKED ARRAY [0..31] OF Char; {partition type}
      pmLgDataStart: LongInt;     {first logical block of data area}
      pmDataCnt:      LongInt;    {number of blocks in data area}
      pmPartStatus:  LongInt;     {partition status information}
      pmLgBootStart: LongInt;     {first logical block of boot code}
      pmBootSize:     LongInt;    {size of boot code, in bytes}
      pmBootAddr:     LongInt;    {boot code load address}
      pmBootAddr2:    LongInt;    {reserved}
      pmBootEntry:    LongInt;    {boot code entry point}
      pmBootEntry2:  LongInt;     {reserved}
      pmBootCksum:    LongInt;    {boot code checksum}
      pmProcessor:    PACKED ARRAY [0..15] OF Char; {processor type}
      pmPad:          ARRAY [0..187] OF Integer;    {reserved}
   END;
```

Routines

```
FUNCTION SCSIReset           : OSErr;
FUNCTION SCSIGet             : OSErr;
FUNCTION SCSISelect          (targetID: Integer): OSErr;
FUNCTION SCSISelAtn          (targetID: Integer): OSErr;
FUNCTION SCSICmd             (buffer: Ptr; count: Integer): OSErr;
FUNCTION SCSIMsgIn           (VAR message: Integer): OSErr;
FUNCTION SCSIMsgOut          (message: Integer): OSErr;
FUNCTION SCSIRead            (tibPtr: Ptr): OSErr;
FUNCTION SCSIRBlind          (tibPtr: Ptr): OSErr;
FUNCTION SCSIWrite           (tibPtr: Ptr): OSErr;
FUNCTION SCSIWBlind          (tibPtr: Ptr): OSErr;
FUNCTION SCSIComplete        (VAR stat: Integer; VAR message: Integer;
                              wait: LongInt): OSErr;
```

```
FUNCTION SCSIStat           : Integer;
```

C Summary

Constants

```
enum {
   /* TIB instruction opcodes */
   scInc       = 1,        /* transfer data, increment buffer pointer */
   scNoInc     = 2,        /* transfer data, don’t increment pointer */
   scAdd       = 3,        /* add long to address */
   scMove      = 4,        /* move long to address */
   scLoop      = 5,        /* decrement counter and loop if > 0 */
   scNop       = 6,        /* no operation */
   scStop      = 7,        /* stop TIB execution */
   scComp      = 8,        /* compare SCSI data with memory */

   /* signature values */
   sbSIGWord   = 0x4552,   /* driver descriptor map signature */
   pMapSIG     = 0x504D    /* partition map signature */
};
```

Data Types

```
struct SCSIInstr {
   unsigned short    scOpcode;      /* operation code */
   unsigned long     scParam1;      /* first parameter */
   unsigned long     scParam2;      /* second parameter */
};
typedef struct SCSIInstr SCSIInstr;

struct Block0 {
   unsigned short    sbSig;         /* device signature */
   unsigned short    sbBlkSize;     /* block size of the device*/
   unsigned long     sbBlkCount;    /* number of blocks on the device*/
   unsigned short    sbDevType;     /* reserved */
   unsigned short    sbDevId;       /* reserved */
   unsigned long     sbData;        /* reserved */
   unsigned short    sbDrvrCount;   /* number of driver descriptor entries */
   unsigned long     ddBlock;       /* first driver’s starting block */
   unsigned short    ddSize;        /* driver’s size, in 512-byte blocks */
   unsigned short    ddType;         /* operating system type (MacOS = 1) */
   unsigned short    ddPad[243];     /* additional drivers, if any */
};
typedef struct Block0 Block0;

Partition {
   unsigned short    pmSig;          /* partition signature */
   unsigned short    pmSigPad;       /* reserved */
   unsigned long     pmMapBlkCnt;    /* number of blocks in partition map */
   unsigned long     pmPyPartStart;  /* first physical block of partition */
   unsigned long     pmPartBlkCnt;   /* number of blocks in partition */
   unsigned char     pmPartName[32];/* partition name */
   unsigned char     pmParType[32];  /* partition type */
   unsigned long     pmLgDataStart;  /* first logical block of data area */
   unsigned long     pmDataCnt;      /* number of blocks in data area */
   unsigned long     pmPartStatus;   /* partition status information */
   unsigned long     pmLgBootStart;  /* first logical block of boot code */
   unsigned long     pmBootSize;     /* size of boot code, in bytes */
   unsigned long     pmBootAddr;     /* boot code load address */
   unsigned long     pmBootAddr2;    /* reserved */
   unsigned long     pmBootEntry;    /* boot code entry point */
   unsigned long     pmBootEntry2;   /* reserved */
   unsigned long     pmBootCksum;    /* boot code checksum */
   unsigned char     pmProcessor[16];  /* processor type */
   unsigned short    pmPad[188];     /* reserved */
};
typedef struct Partition Partition;
```

Functions

```
pascal OSErr SCSIReset      (void);
pascal OSErr SCSIGet        (void);
pascal OSErr SCSISelect     (short targetID);
pascal OSErr SCSISelAtn     (short targetID);
pascal OSErr SCSICmd        (Ptr buffer, short count);
pascal OSErr SCSIMsgIn      (short *message);
pascal OSErr SCSIMsgOut     (short message);
pascal OSErr SCSIRead       (Ptr tibPtr);
pascal OSErr SCSIRBlind     (Ptr tibPtr);
pascal OSErr SCSIWrite      (Ptr tibPtr);
pascal OSErr SCSIWBlind     (Ptr tibPtr);
pascal OSErr SCSIComplete   (short *stat, short *message,
                             unsigned long wait);
pascal short SCSIStat       (void);
```

Assembly-Language Summary

Data Structures

Transfer Instruction Block

0	`scOpcode`	word	operation code
2	`scParam1`	long	first parameter
6	`scParam2`	long	second parameter

Driver Descriptor Record

0	`sbSig`	word	device signature
2	`sbBlkSize`	word	block size of the device
4	`sbBlkCount`	long	number of blocks on the device
8	`sbDevType`	word	reserved
10	`sbDevId`	word	reserved
12	`sbData`	long	reserved
16	`sbDrvrCount`	word	number of driver descriptor entries
18	`ddBlock`	long	first driver's starting block
22	`ddSize`	word	driver's size, in 512-byte blocks
24	`ddType`	word	operating system type (MacOS = 1)
26	`ddPad`	486 bytes	additional drivers, if any

Partition Map Entry Record

0	`pmSig`	word	partition signature
2	`pmSigPad`	word	reserved
4	`pmMapBlkCnt`	long	number of blocks in partition map
8	`pmPyPartStart`	long	first physical block of partition
12	`pmPartBlkCnt`	long	number of blocks in partition
16	`pmPartName`	32 bytes	partition name
48	`PmParType`	32 bytes	partition type
80	`pmLgDataStart`	long	first logical block of data area
84	`pmDataCnt`	long	number of blocks in data area
88	`pmPartStatus`	long	partition status information
92	`pmLgBootStart`	long	first logical block of boot code
96	`pmBootSize`	long	size of boot code, in bytes
100	`pmBootAddr`	long	boot code load address
104	`pmBootAddr2`	long	reserved
108	`pmBootEntry`	long	boot code entry point
112	`pmBootEntry2`	long	reserved

116	`pmBootCksum`	long	boot code checksum
120	`pmProcessor`	16 bytes	processor type
136	`pmPad`	376 bytes	reserved

Trap Macros

Trap Macros Requiring Routine Selectors

`_SCSIDispatch`

Selector	Routine
$00	`SCSIReset`
$01	`SCSIGet`
$02	`SCSISelect`
$03	`SCSICmd`
$04	`SCSIComplete`
$05	`SCSIRead`
$06	`SCSIWrite`
$08	`SCSIRBlind`
$09	`SCSIWBlind`
$0A	`SCSIStat`
$0B	`SCSISelAtn`
$0C	`SCSIMsgIn`
$0D	`SCSIMsgOut`

Result Codes

`noErr`	0	No error
`scCommErr`	2	Communications error, operation timeout
`scArbNBErr`	3	Bus busy, arbitration timeout
`scBadParmsErr`	4	Bad parameter or unrecognized TIB instruction
`scPhaseErr`	5	Phase error on the SCSI bus
`scCompareErr`	6	Comparison error from `scComp` instruction
`scMgrBusyErr`	7	SCSI Manager busy
`scSequenceErr`	8	Attempted operation is out of sequence
`scBusTOErr`	9	Bus timeout during blind transfer
`scComplPhaseErr`	10	SCSI bus was not in status phase on entry to `SCSIComplete`