This chapter describes how your application or device driver can use the Slot Manager to identify expansion cards and communicate with the firmware on a card.

You need to use the Slot Manager only if you are writing an application or a device driver that must address an expansion card directly. For example, you need to use the Slot Manager if you are writing a driver for a video card, but not if you only want to display information on a monitor for which a device driver already exists.

The ***Slot Manager*** provides functions to help you search through the data structures that expansion cards use to organize the information in their firmware. The meaning of the information in the data structures varies from card to card; you need to know the specifics of a card in order to interpret its data structures. To interpret these data structures, you need to know the information in *Designing Cards and Drivers for the Macintosh Family,* third edition, as well as information specific to the expansion card you're using.

This chapter begins with a brief introduction to Apple's implementation of the ***NuBus expansion interface*** . The NuBus interface provides a 32-bit-wide synchronous, multislot expansion bus for adding expansion cards to Macintosh computers. This introduction explains the firmware data structures of NuBus expansion cards, but does not provide much detail about the information these data structures contain. If you are designing an expansion card, you must read *Designing Cards and Drivers for the Macintosh Family,* third edition. If you are writing a driver for a device on a card, you should also read the chapter "Device Manager" in this book.

After introducing the NuBus architecture and expansion card design, this chapter discusses how you can

- enable and disable NuBus cards
- delete, restore, enable, disable, and find information in an expansion card's firmware
- install and remove slot interrupt handlers

Introduction to Slots and Cards

The Macintosh Operating System provides a standardized interface to expansion cards through the Slot Manager. The Slot Manager supports two types of expansion cards: NuBus and processor-direct slot (PDS). Most Macintosh computers include one or both of these expansion systems. Although the discussion and examples in this chapter use NuBus, the information also applies to PDS expansion cards.

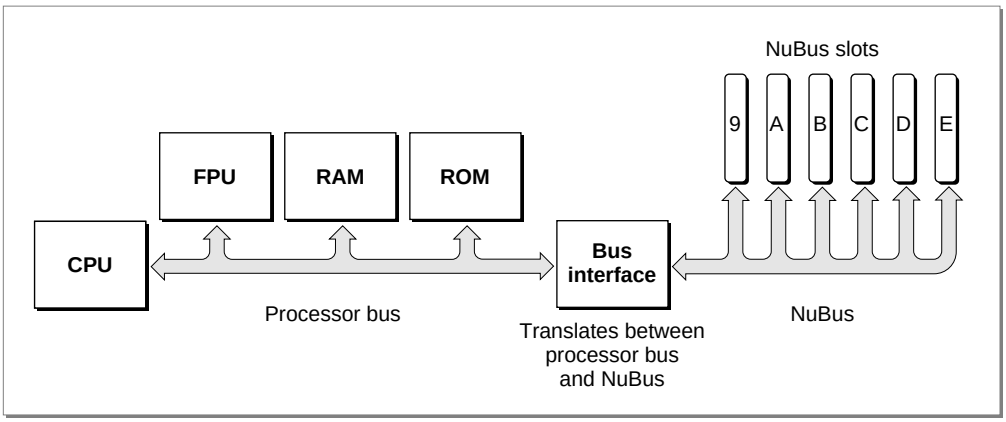
Processor-direct slot expansion cards connect directly to the processor bus, giving them direct access to the microprocessor and therefore a speed advantage over NuBus cards. However, because the PDS expansion interface is an extension of the processor bus, the configuration of the slot depends on which microprocessor is used by the computer. Refer to *Designing Cards and Drivers for the Macintosh Family,* third edition, for information specific to PDS expansion cards.

Macintosh computers that include the NuBus expansion interface contain one or more identical NuBus slots. Each slot is identified by slot a number in the range $1 through $E. (Slot $0 corresponds to the main logic board, and slot $F is reserved for NuBus address translation.)

Note
For convenience, this chapter refers to a NuBus configuration with six slots numbered $9 through $E. Keep in mind that Macintosh computers may have more or fewer slots. Refer to the appropriate Macintosh Developer Note or *Guide to the Macintosh Family Hardware*, second edition, for information about specific models. ◆

In Macintosh computers, the processor bus (which connects the microprocessor to RAM, ROM, and the FPU) and the NuBus (which connects the NuBus slots) are connected by a ***bus interface*** , as shown in Figure 2-1.

Figure 2-1 Simplified processor-bus and NuBus architecture



Both the processor bus and the NuBus are 4 bytes (32 bits) wide. The bus interface transfers data between the buses in byte lanes. A ***byte lane*** is any of the 4 bytes that make up the 32-bit bus. Because the processor bus and the NuBus interpret the significance of bytes within words differently, the bus interface must perform byte-lane swapping between the two buses.

The bus interface also performs some address translation between the two buses. It maps certain address ranges on each bus to different address ranges on the other bus. *Designing Cards and Drivers for the Macintosh Family,* third edition, discusses byte lanes and address translation in more detail.

The next section,“Slot Address Allocations,” discusses the address ranges assigned by the Macintosh architecture to each NuBus slot.

The section “Firmware” on page 2-7 introduces the data structures that cards use to organize information in their firmware.

Slot Address Allocations

The Macintosh architecture assigns certain address ranges to each slot. The microprocessor communicates with an expansion card in a particular slot by reading or writing to memory in the slot's address range. Expansion cards can also communicate with each other in this manner.

The NuBus architecture supports 32-bit addressing, providing 4 gigabytes of address space. All Macintosh computers that use Motorola 68030, 68040, or PowerPC processors support 32-bit addressing under System 7. Macintosh computers that use Motorola 68000 or 68020 processors, and those running System 6, use 24-bit addressing. This section describes address space allocation in both the 32-bit and 24-bit modes.

In 32-bit mode, the Macintosh architecture assigns two address ranges to each NuBus slot: a 256-megabyte super slot space and a 16-megabyte standard slot space.

The 4 gigabytes of 32-bit address space contain 16 regions of 256 megabytes apiece. Each region constitutes the ***super slot space*** for one possible slot ID. Each super slot space spans an address range of $s000 0000 through $sFFF FFFF, where *s* is a hexadecimal digit $1 through $E, corresponding to the slot ID. For example, the address range $9000 0000 through $9FFF FFFF constitutes the super slot space for slot $9.

The ***standard slot spaces*** are 16 megabytes apiece and have address ranges of the form $Fs00 0000 through $FsFF FFFF, where *s* is the slot ID. The standard slot space for slot $9, for example, is $F900 0000 through $F9FF FFFF. Figure 2-2 shows the super slot and standard slot subdivisions of the 32-bit address space.

In 24-bit mode, software can address only a fraction of each card's allocated address range. In this mode, the Operating System assigns each slot a 1-megabyte ***minor slot space***. The bus interface translates 24-bit addresses on the processor bus with the form $sx xxxx (where *s* is a slot ID and *x* is any hexadecimal digit) into 32-bit NuBus addresses of the form $Fs0x xxxx, which is the first megabyte of the slot's standard slot space.

For example, 24-bit addresses in the range $90 0000 through $9F FFFF constitute the minor slot space corresponding to slot $9. The hardware translates these addresses into the NuBus address range $F900 0000 through $F90F FFFF.

Figure 2-2 The NuBus 32-bit address space

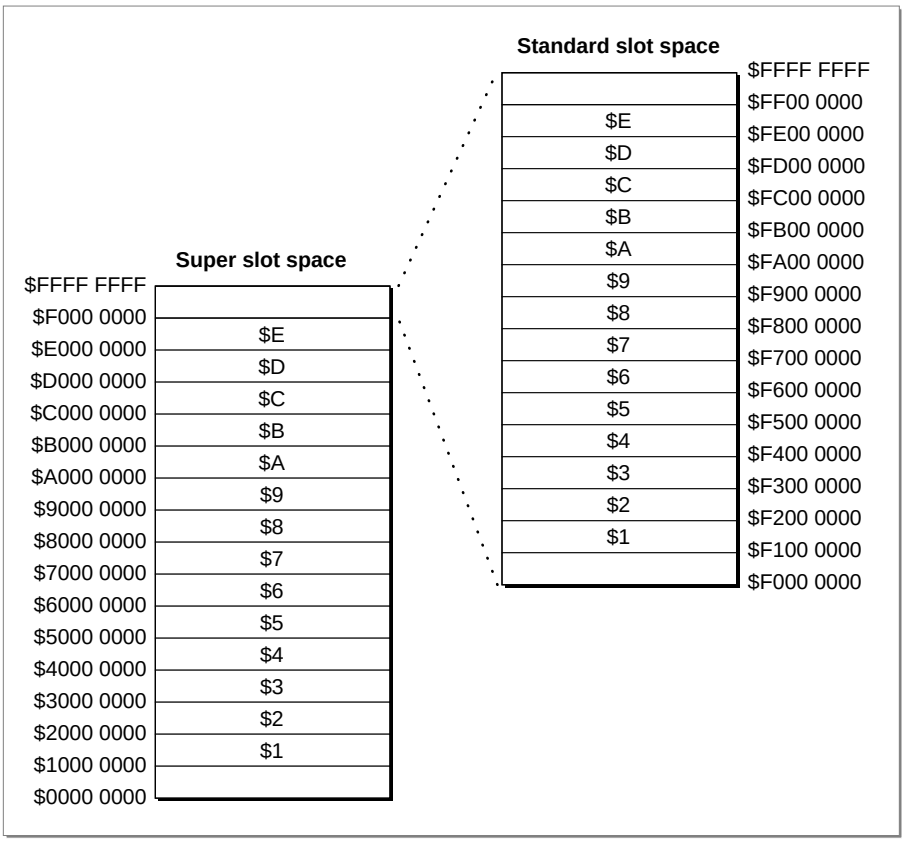


Table 2-1 shows the address allocations for each slot ID.

Table 2-1 Slot address allocations by slot ID

Slot ID	24-bit minor slot space (1 MB)	32-bit minor slot space (1 MB)	Standard slot space (16 MB)	Super slot space (256 MB)
$1	$1x xxxx	$F10x xxxx	$F1xx xxxx	$1xxx xxxx
$2	$2x xxxx	$F20x xxxx	$F2xx xxxx	$2xxx xxxx
$3	$3x xxxx	$F30x xxxx	$F3xx xxxx	$3xxx xxxx
$4	$4x xxxx	$F40x xxxx	$F4xx xxxx	$4xxx xxxx
$5	$5x xxxx	$F50x xxxx	$F5xx xxxx	$5xxx xxxx
$6	$6x xxxx	$F60x xxxx	$F6xx xxxx	$6xxx xxxx
$7	$7x xxxx	$F70x xxxx	$F7xx xxxx	$7xxx xxxx
$8	$8x xxxx	$F80x xxxx	$F8xx xxxx	$8xxx xxxx

continued

Table 2-1 Slot address allocations by slot ID (continued)

Slot ID	24-bit minor slot space (1 MB)	32-bit minor slot space (1 MB)	Standard slot space (16 MB)	Super slot space (256 MB)
$9	$9x xxxx	$F90x xxxx	$F9xx xxxx	$9xxx xxxx
$A	$Ax xxxx	$FA0x xxxx	$FAxx xxxx	$Axxx xxxx
$B	$Bx xxxx	$FB0x xxxx	$FBxx xxxx	$Bxxx xxxx
$C	$Cx xxxx	$FC0x xxxx	$FCxx xxxx	$Cxxx xxxx
$D	$Dx xxxx	$FD0x xxxx	$FDxx xxxx	$Dxxx xxxx
$E	$Ex xxxx	$FE0x xxxx	$FExx xxxx	$Exxx xxxx

Firmware

The firmware of a NuBus expansion card contains information that identifies the card and its functions. Your application uses the Slot Manager to communicate with this firmware. This firmware, called the ***declaration ROM***, may also include other information, such as initialization code or code for drivers that communicate with devices on the card. The sole purpose of many Slot Manager routines is to provide access to the information in the declaration ROM.

This section discusses the data structures used to store information in the declaration ROM. You'll need to understand these structures in order to use the Slot Manager routines. To create firmware for an expansion card, you'll need to read *Designing Cards and Drivers for the Macintosh Family*, third edition.

The declaration ROM includes these elements:

- The ***sResources***. An sResource is a data structure in the firmware of an expansion card's declaration ROM that defines a function or capability of the card. An sResource typically contains information about a single function or capability, although some sResources may contain other data—for example, device drivers, icons, fonts, code, or vendor-specific information.
- The ***sResource directory***. The sResource directory is a special sResource that contains offsets to all of the other sResources in the declaration ROM.
- The ***format block***. The format block is a data structure that allows the Slot Manager to find the declaration ROM and to validate it. It contains some identification information and an offset to the sResource directory.

The next few sections discuss these data structures in more detail.

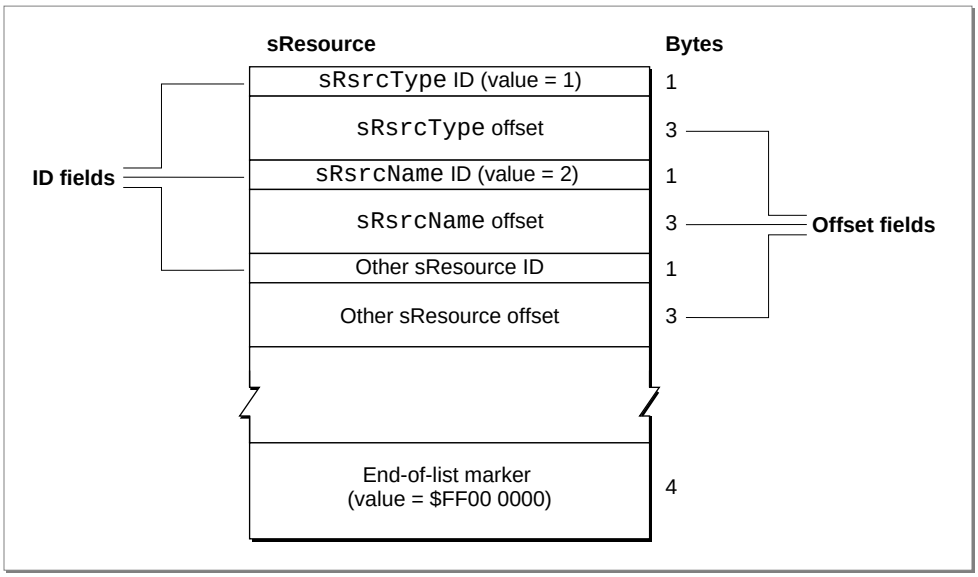
The sResource

An sResource consists of a list of 4-byte entries. The first byte of each entry is an ID field that identifies the type of data contained in the entry. The next 3 bytes contain either data for the sResource or an offset to additional data such as icon definitions, code, or device drivers relating to the sResource.

Note
An sResource is sometimes referred to as a *slot resource*. Note, however, that an sResource is a data structure in the firmware of a NuBus expansion card and not the type of Macintosh resource associated with the Resource Manager (which is described in *Inside Macintosh: More Macintosh Toolbox*). ◆

The last entry in an sResource must contain an end-of-list marker—a 4-byte series with the value $FF 00 00 00. Figure 2-3 shows the format of a typical sResource.

Figure 2-3 The structure of a typical sResource



The ID field of each sResource entry indicates the type of information in the offset field of the entry. Apple reserves the range 0 through 127 for common sResource IDs. *Designing Cards and Drivers for the Macintosh Family,* third edition, includes a complete list of the Apple-defined sResource IDs and their meanings.

The offset field of each entry can contain a byte or word of data, or an offset to a larger block of data. This field takes one of three possible forms:

- two $00 bytes followed by an 8-bit byte of data
- a single $00 byte followed by a 16-bit word of data
- a signed 24-bit offset to a larger data structure; the offset is relative to the address of the preceding ID field

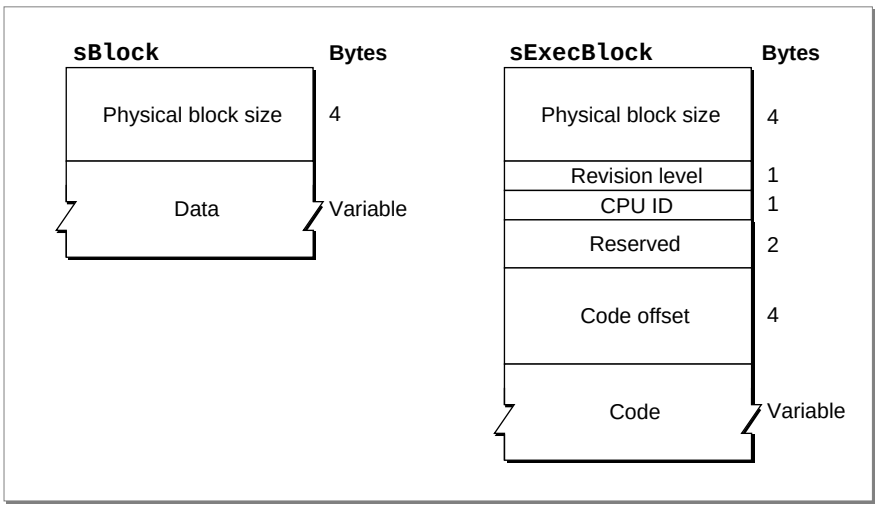
Table 2-2 lists the kinds of large data types commonly used in sResources.

Table 2-2 Large data types used in sResources

Data type	Description
`Long`	32 bits, signed or unsigned
`Pointer`	32 bits, signed or unsigned
`cString`	One-dimensional array of bytes, ending with 0
`sBlock`	A sized block of data (see Figure 2-4)
`sExecBlock`	A sized block of code (see Figure 2-4)

The `sBlock` and `sExecBlock` data structures begin with a `size` field, which contains the physical size of the block (including the `size` field). In the `sBlock` structure, the `size` field is followed by data. The `sExecBlock` structure includes additional fields and a code block. Figure 2-4 shows these structures.

Figure 2-4 The format of the `sBlock` and `sExecBlock` data structures

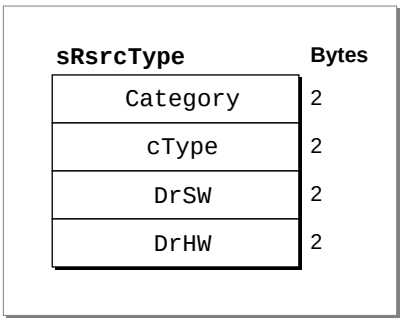


Type and Name Entries

As shown in Figure 2-3, the Slot Manager requires that each sResource contain an `sRsrcType` entry, which identifies the sResource type, and an `sRsrcName` entry, which provides the sResource name.

The `sRsrcType` entry contains an ID value of 1 and an offset to an `sRsrcType` entry. Figure 2-5 shows the format of an `sRsrcType` entry.

Figure 2-5 The sRsrcType entry format



The fields of the `sRsrcType` entry are as follows:

Field	Description
`Category`	The most general classification of card functions. Examples of categories are `catDisplay` and `catNetwork`.
`cType`	The subclass of the category. For example, within the `catDisplay` category there is a `typeVideo` subcategory; within the `catNetwork` category, there is a `typeEtherNet` subcategory.
`DrSW`	The driver software interface to the card. (This provides the calling interface for applications and system software.) For example, under the `catDisplay` category and the `typeVideo` subcategory, there is a `drSwApple` software interface that indicates the Apple-defined interface to work with QuickDraw using Macintosh Operating System frame buffers.
`DrHW`	The identification of the specific hardware device associated with the driver software interface. Generally, only the driver interacts with the hardware specified here.

Every card has a unique `sRsrcType` entry that must be assigned by Apple Computer, Inc. If you are developing a card, refer to *Designing Cards and Drivers for the Macintosh Family*, third edition, for information on obtaining an `sRsrcType` entry.

The `sRsrcName` entry in an sResource contains an ID value of 2 and an offset to a `cString` data structure containing the sResource name. By convention, the `sRsrcName` field is derived by stripping the prefixes from the `sRsrcType` values and separating the fields by underscores. For example, the `sRsrcName` field for an sResource whose `sRsrcType` values are `catDisplay`, `typeVideo`, `DrSwApple`, and `DrHwTFB` becomes `'Display_Video_Apple_TFB'`.

Designing Cards and Drivers for the Macintosh Family, third edition, provides information about these and other sResource entry types.

The Board sResource and Functional sResources

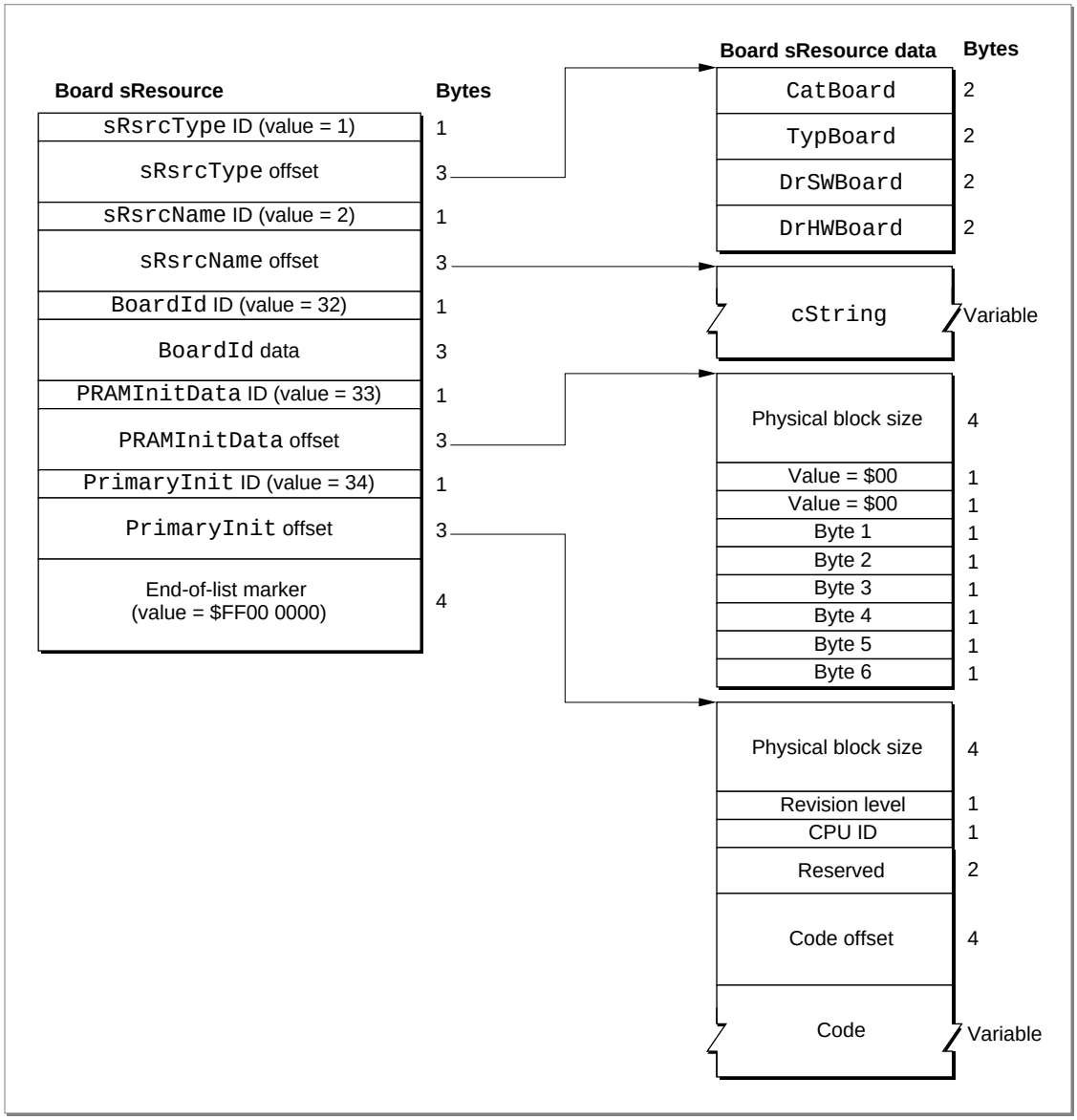
Every card must have a single ***board sResource*** that contains information about the card as a whole. An sResource relating to a specific function is called a ***functional sResource*** , and a card may have as many of them as necessary. For example, a video card may have separate functional sResources for every pixel depth it supports. (See Figure 2-8 on page 2-14 for an example of a functional sResources for a video card, and see *Designing Cards and Drivers for the Macintosh Family,* third edition, for additional examples that include code listings.)

The entries in the board sResource provide the Slot Manager with a card's identification number, vendor information, board flags, and initialization code. Like all sResources, the board sResource must include an `sRsrcType` entry and an `sRsrcName` entry. The board `sRsrcType` entry must contain the constants `CatBoard` ($0001), `TypBoard` ($0000), `DrSWBoard` ($0000), and `DrHWBoard` ($0000). The `sRsrcName` entry for the board sResource name does not follow the same convention as other sResources: the `sRsrcName` entry for the board sResource contains the name of the entire card (for example, `'Macintosh Display Card'`).

The board sResource must also contain a `BoardId` entry, a word that contains the card design identification number assigned by Apple Computer, Inc. *Designing Cards and Drivers for the Macintosh Family,* third edition, describes other Apple-defined entries specifically for board sResources.

Figure 2-6 shows a sample board sResource. It shows an `sRsrcType` entry and an `sRsrcName` entry and also includes three entry types, `BoardID`, `PRAMInitData`, and `PrimaryInit`, which are discussed in *Designing Cards and Drivers for the Macintosh Family,* third edition.

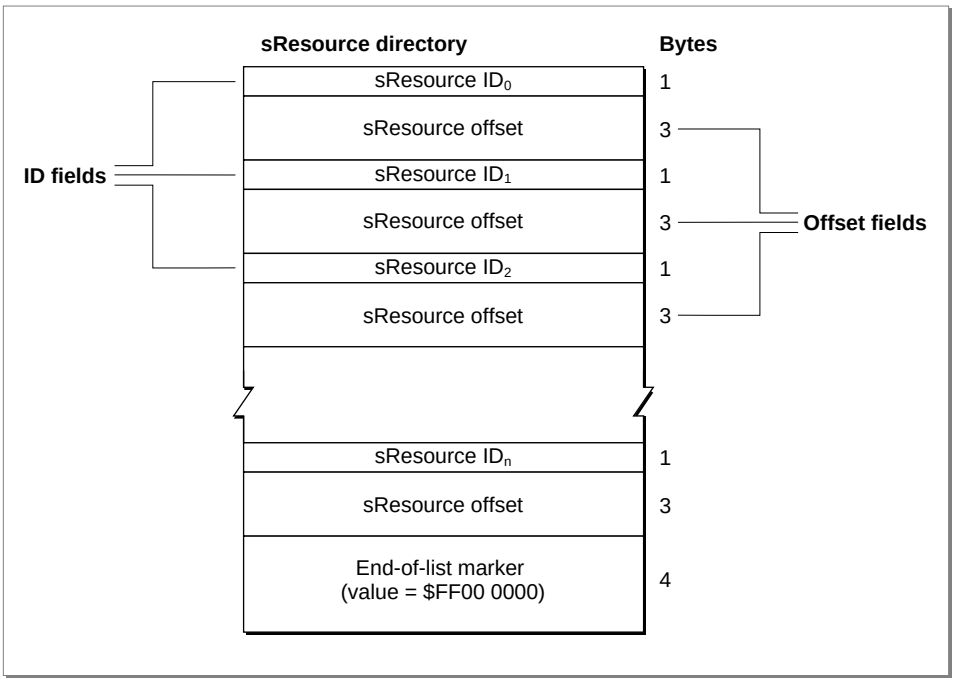
Figure 2-6 A sample board sResource



The sResource Directory

The sResource directory lists all the sResources in the declaration ROM and provides an offset to each one. The sResource directory has the same structure as an sResource—that is, an sResource directory consists of a series of 4-byte entries, where the first byte is an ID field and the next 3 bytes contain an offset to additional data. Figure 2-7 shows the format of the sResource directory.

Figure 2-7 The structure of the sResource directory



The sResource ID field of an entry in the sResource directory always identifies an sResource on the card. Each sResource in the card firmware requires a unique ID defined by the card designer, and the ID must be in the range 1 through 254. For example, an entry for the board sResource must appear first in a card's sResource directory, so card designers typically assign an sResource ID value of 1 to the board sResource. The sResource ID numbers must appear in the sResource directory in ascending order. An sResource directory must conclude with the end-of-list marker ($FF 00 00 00).

The offset field of each entry contains a signed 24-bit offset to the sResource corresponding to the sResource ID field. The offset value counts only those bytes accessible by valid byte lanes, and is relative to the address of the sResource ID field.

The Format Block

The format block always resides at the highest address in the standard slot space of a declaration ROM. At startup, the Slot Manager locates installed cards by searching each slot space for a valid format block. The format block contains information about the declaration ROM and an offset to the sResource directory. The Slot Manager uses the format block to validate the declaration ROM and locate the sResources.

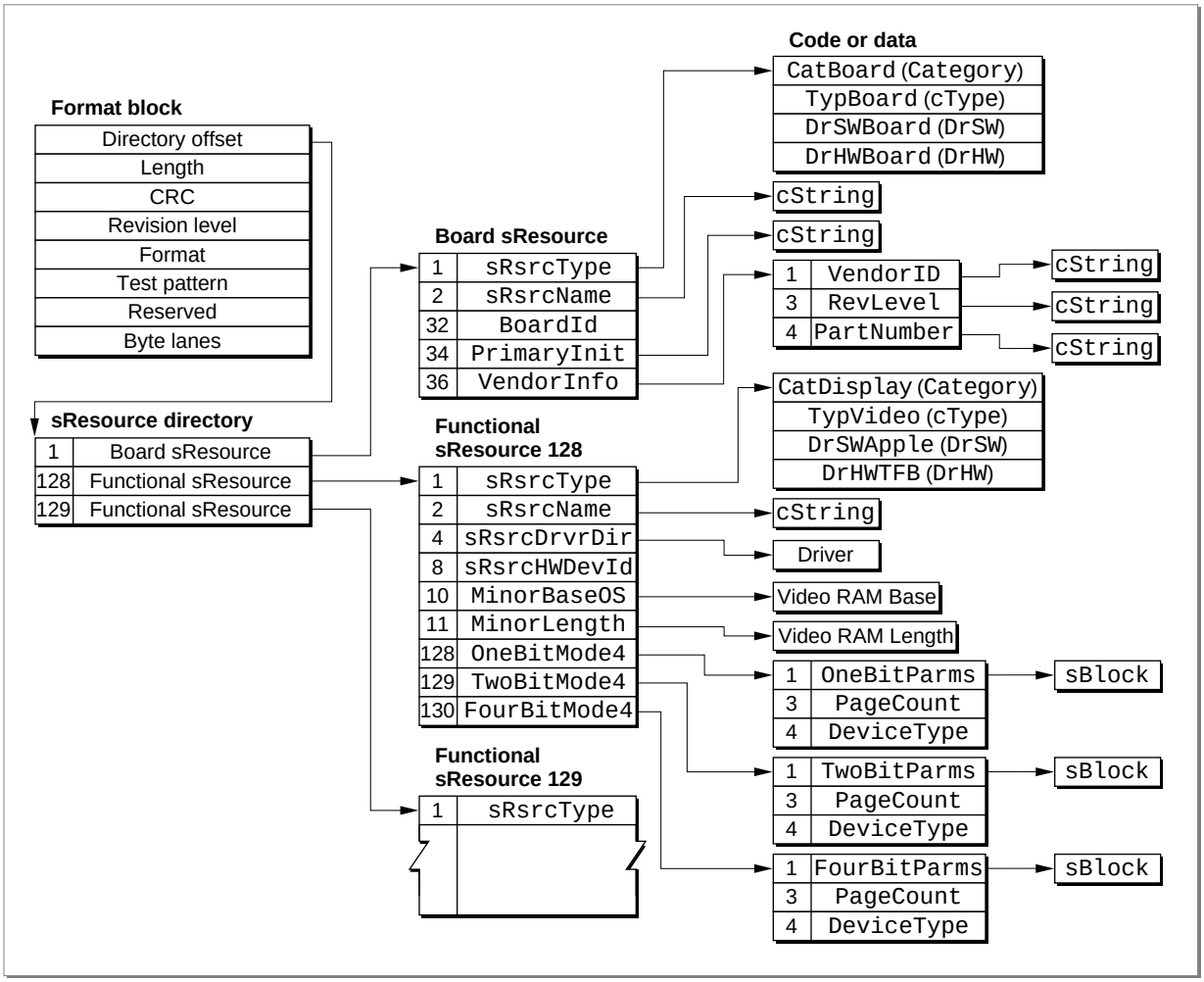
The format block also contains a value that specifies which of the four byte lanes are occupied by the declaration ROM. These byte lanes are called the *valid byte lanes.* Some declaration ROMs do not appear on all four byte lanes, so software cannot read meaningful data at every memory location in the address space for the byte lanes.

IMPORTANT
The format block defines which byte lanes are valid for the declaration ROM only. The valid byte lanes are determined by card design, and may be different for other memory-mapped devices on the card. ▲

Designing Cards and Drivers for the Macintosh Family, third edition, defines the structure of the format block and gives examples of how the valid byte lanes affect communication with a declaration ROM.

Figure 2-8 illustrates the relationship of the format block, the sResource directory, and the sResources for a sample video card. For every entry in the sResource directory and in the sResources, its ID number is shown on the left side of the entry. As shown in this figure, the board sResource is the first sResource listed in the sResource directory. Each functional sResource that follows in turns defines a display capability provided by the card. (To simplify this figure, only one complete functional sResource is shown.)

Figure 2-8 The format block and sResources for a sample video card



About the Slot Manager

The Slot Manager provides three basic services:

- On startup, it examines each slot and initializes any expansion cards it finds.
- It maintains data structures that contain information about each slot and every available sResource.
- It provides functions that allow you to get information about expansion cards and their sResources.

There are two variations of the System 7 Slot Manager: version 1 and version 2. Version 1 of the Slot Manager is RAM based and is installed by the user with the System 7 upgrade kit. Version 2 is included in the ROM of newer Macintosh computers.

At startup, the version of the Slot Manager in ROM searches each slot for a declaration ROM and creates a ***slot information record*** for each slot. See "Slot Information Record" on page 2-24 for the definition of the `SInfoRecord` data type.

As the Slot Manager searches the slots, it identifies all of the sResources in each declaration ROM and creates a table—the ***slot resource table*** (***SRT***)—that lists all of the sResources currently available to the system. The slot resource table is a private data structure maintained by the Slot Manager. Applications and device drivers use Slot Manager routines to get information from the slot resource table.

After building the slot resource table, the Slot Manager initializes the 6 bytes reserved for each slot in parameter RAM. If the slot has an expansion card with a `PRAMInitData` entry in its board sResource, the Slot Manager uses the values in that entry to initialize the parameter RAM; otherwise, it clears those bytes in parameter RAM.

Next, the Slot Manager disables interrupts and executes the code in the `PrimaryInit` entry of the board sResource for each card. Note that at this point in the startup, the keyboard and the mouse are not initialized and that a card's `PrimaryInit` code has only limited control over the functionality of the card itself.

If certain values (defined by the Start Manager) are set in a card's parameter RAM, a card with an `sRsrcBootRec` entry may take over the system startup process. The Start Manager passes control to the code in the `sRsrcBootRec` early in the startup sequence, before system patches are installed. Refer to the chapter "Start Manager" in *Inside Macintosh: Operating System Utilities* for more information about the startup process.

Designing Cards and Drivers for the Macintosh Family, third edition, describes the `PRAMInitData`, `PrimaryInit`, and `sRsrcBootRec` entry types.

If no card takes over, the normal system startup continues. After version 1 of the Slot Manager is loaded, it conducts a second search for declaration ROMs, this time in 32-bit mode. If the Slot Manager finds any additional NuBus cards, it adds their sResources to the slot resource table and executes the code in their `PrimaryInit` entries. (Version 2 of the Slot Manager, which resides in ROM, does not need to conduct a second search.)

Note
Some versions of the Slot Manager prior to System 7 address NuBus cards in 24-bit mode and may not be able to identify all cards. After version 1 of the Slot Manager is loaded, it locates these cards. ◆

After all system patches have been installed, version 1 or later of the Slot Manager executes the code in any `SecondaryInit` entries it finds in the declaration ROMs. It does not reexecute the code from `PrimaryInit` entries, reinitialize parameter RAM, or restore any sResources deleted by the `PrimaryInit` code.

Note
Most versions of the Slot Manager prior to System 7 do not execute code from `SecondaryInit` entries. ◆

After the Slot Manager executes `SecondaryInit` code, it searches for sResources that have an `sRsrcFlags` entry with the `fOpenAtStart` flag set. When the Slot Manager finds an sResource with this flag set, it loads the device driver from the `sRsrcDrvrDir` entry of the sResource, or calls the code in the sResource's `sRsrcLoadRec` entry, which loads the sResource's device driver.

Finally, the system executes initialization resources of type `'INIT'`.

See *Designing Cards and Drivers for the Macintosh Family*, third edition, for details about the `sRsrcFlags`, `sRsrcDrvrDir`, and `sRsrcLoadRec` entry types.

Using the Slot Manager

The Slot Manager allows you to enable and disable NuBus cards, manipulate the slot resource table, get information from slot information records, get status information, and read and change expansion cards' parameter RAM. However, the majority of Slot Manager routines search for sResources in the slot resource table or provide information from these structures.

The Slot Manager provides a variety of methods to find an sResource. These methods include searching for an sResource with a particular sResource ID, searching for an sResource with a particular sResource type, searching through all sResources, searching through only the enabled sResources, and so on.

The Slot Manager also provides a number of routines that return information from sResources. Some of these routines, like the `SReadByte` and `SGetCString` functions, return one particular type of data structure. Others, like the `SFindStruct` function, can return information about any data structure. Functions such as `SGetDriver` and `SExec` not only return information from an sResource, they also perform additional operations like loading the sResource's driver or executing the code of an `sExecBlock` data structure.

You can use the `SVersion` function, described on page 2-30, to determine if the Slot Manager is version 1, version 2, or a version that predates System 7.

Enabling and Disabling NuBus Cards

Version 1 and later of the Slot Manager allows you to temporarily disable your NuBus card. You might want to do this if, for example, you are designing a NuBus card that must be addressed in 32-bit mode or that requires RAM-based system software patches to be loaded into memory before the card is initialized. Your `PrimaryInit` code can disable the card temporarily and the `SecondaryInit` code can reenable it.

To disable a NuBus card temporarily, the initialization routine in your `PrimaryInit` record should return in the `seStatus` field of the `SEBlock` data structure (described in "Slot Execution Parameter Block" on page 2-27) an error code with a value in the range `svTempDisable` ($8000) through `svDisabled` ($8080). The Slot Manager places this code in the `siInitStatusV` field of the slot information record for the slot, and places the fatal error `smInitStatVErr` (–316) in the `siInitStatusA` field of the slot information record. The card and its sResources are then unavailable for use by the Operating System.

After the Operating System loads RAM patches, the Slot Manager checks the value of the `siInitStatusA` field of each slot information record. If this value is greater than or equal to 0, indicating no error, the Slot Manager executes the `SecondaryInit` code for the slot, if any. If the value in the `siInitStatusA` field is `smInitStatVErr`, the Slot Manager checks the `siInitStatusV` field. If the value of the `siInitStatusV` field is in the range `svTempDisable` through `svDisabled`, the Slot Manager sets the `siInitStatusA` field to 0 and runs the `SecondaryInit` code.

For examples of `PrimaryInit` and `SecondaryInit` code, see *Designing Cards and Drivers for the Macintosh Family,* third edition.

Deleting and Restoring sResources

Some NuBus cards have sResources to support a variety of system configurations or modes. The Slot Manager loads all of the sResources during system initialization, and then the card's `PrimaryInit` code can delete from the slot resource table any sResources that are not appropriate for the system as configured. If the user changes the system configuration or selects a different mode of operation, your card can reinstall a deleted sResource. The `SDeleteSRTRec` function deletes sResources; the `InsertSRTRec` function reinstalls them.

Because none of the Slot Manager functions can search for sResources that have been deleted from the slot resource table, you must keep a record of all sResources you delete so that you will have the appropriate parameter values when you want to reinstall one.

When you reinstall an sResource, it may be necessary to update the `dCtlSlotId` and `dCtlDevBase` fields in the slot device driver's device control entry. You need to update the `dCtlSlotId` field if you change the sResource ID. The `dCtlDevBase` field holds the base address of the slot device. For a video card this is the base address for the pixel map in the card's `GDevice` record (which is described in *Inside Macintosh: Imaging With QuickDraw*). The `InsertSRTRec` function updates the `dCtlDevBase` field automatically if you supply a valid driver reference number.

Enabling and Disabling sResources

Under certain circumstances, you might want to disable an sResource while it remains listed in the slot resource table. For example, a NuBus card might provide several modes of operation, only one of which can be active at a given time. Your application might want to disable the sResources associated with all but the active mode, but still list all available modes in a menu. When the user selects a new mode, your application can then disable the currently active sResource and enable the one the user selected.

You use the `SetSRsrcState` function to enable or disable an sResource. Listing 2-1 disables the sResource in slot $A with an sResource ID of 128 and enables the sResource in the same slot with an sResource ID of 131.

Listing 2-1 Disabling and enabling an sResource

```
PROCEDURE MyDisableAndEnableSResource;
VAR
   mySpBlk:    SpBlock;
   myErr:      OSErr;
BEGIN
   WITH mySpBlk DO      {set required values in parameter block}
   BEGIN
      spParamData := 1;       {disable}
      spSlot := $A;           {slot number}
      spID := 128;            {sResource ID}
      spExtDev := 0;          {ID of external device}
   END;
   myErr := SetSRsrcState(@mySpBlk);
   IF myErr = noErr THEN
   BEGIN
      WITH mySpBlk DO
      BEGIN
         spParamData := 0;    {enable}
         spSlot := $A;        {slot number}
         spID := 131;         {sResource ID}
         spExtDev := 0;       {ID of external device}
      END;
      myErr := SetSRsrcState(@mySpBlk);
   END;
END;
```

Searching for sResources

The Slot Manager provides several functions that search for sResources in the slot resource table. These functions allow you to specify which sResources to search, but each function provides slightly different options.

The `SNextSRsrc` and `SNextTypeSRsrc` functions allow you to search for enabled sResources by slot. The `SGetSRsrc` and `SGetTypeSRsrc` functions, available only with the System 7 Slot Manager (that is, version 1 and version 2 of the Slot Manager), allow you to search for disabled sResources as well as enabled ones. Table 2-3 summarizes the Slot Manager search routines and the options available for each.

Table 2-3 The Slot Manager search routines

Function	State of sResources for which it searches	Slots it searches	Which sResources it searches for	Type of sResource it searches for
`SNextSRsrc`	Enabled only	Specified slot and higher slots	Next sResource only	Any type
`SGetSRsrc`*	Your choice of enabled only or both enabled and disabled	Your choice of one slot only or specified slot and higher slots	Your choice of specified sResource or next sResource	Any type
`SNextTypeSRsrc`	Enabled only	Specified slot and higher slots	Next sResource only	Specified type only
`SGetTypeSRsrc`*	Your choice of enabled only or both enabled and disabled	Your choice of one slot only or specified slot and higher slots	Next sResource only	Specified type only

* Available only with the System 7 Slot Manager (that is, version 1 and version 2 of the Slot Manager)

Listing 2-2 shows how to use the `SGetTypeSRsrc` function to search all slots for both enabled and disabled sResources with an sResource type category of `catDisplay` and an sResource type subcategory of `typeVideo`.

Listing 2-2 Searching for a specified type of sResource

```
PROCEDURE MySResourceSearch;

VAR
   mySpBlk:    SpBlock;
   myErr:      OSErr;
```

```
BEGIN
   WITH mySpBlk DO                  {set required values in parameter block}
   BEGIN
      spParamData := fAll;          {fAll flag = 1: search all sResources}
      spCategory  := catDisplay;    {search for Category catDisplay}
      spCType     := typeVideo;     {search for cType typeVideo}
      spDrvrSW    := 0;             {this field not being matched}
      spDrvrHW    := 0;             {this field not being matched}
      spTBMask    := 3;             {match only Category and cType fields}
      spSlot      := 1;             {start search from slot 1}
      spID        := 1;             {start search from sResource ID 1}
      spExtDev    := 0;             {external device ID (card-specific)}
   END;
   myErr := noErr;
   WHILE myErr = noErr DO           {loop to search sResources}
   BEGIN
      myErr := SGetTypeSRsrc(@mySpBlk);
      MySRsrcProcess(mySpBlk);      {routine to process results}
   END;
   IF myErr <> smNoMoresRsrcs THEN  {all search functions return this value }
      MyHandleError(myErr);         { when search is complete}
END;
```

Obtaining Information From sResources

If you are writing a driver for a card device, you will most likely want access to the information in an sResource.

The Slot Manager provides many functions that return information from the entries of an sResource. The `SOffsetData`, `SReadByte`, and `SReadWord` functions return information from the offset field of an sResource entry. The `SReadLong`, `SGetCString`, and `SGetBlock` functions return copies of the standard data structures pointed to by the offset field of an sResource entry. The `SFindStruct` and `SReadStruct` functions allow access to other data structures pointed to by sResource entries.

Listing 2-3 shows an example of searching for a board sResource and obtaining its name. This example starts at a particular slot number and then searches for the board sResource in that slot or, if necessary, in higher slots. Once it finds the board sResource, Listing 2-3 calls the `SGetCString` function, which returns a pointer to a buffer containing the name string for the card.

Listing 2-3 Searching for the name of a board sResource

```
PROCEDURE FindBoardsResource  (VAR slotNumber: Integer;
                               VAR finished: Boolean);
VAR
   mySpBlk: SpBlock;
   myErr: OSErr;
BEGIN
   {First, get a pointer to the board sResource for the slot.}
   WITH mySpBlk DO BEGIN
      spSlot      := slotNumber;  {start searching in this slot, }
                                  { and continue until found}
      spID        := 0;
      spCategory  := 1;           {sRsrcType values for a board sResource}
      spCType     := 0;
      spDrvrSw    := 0;
      spDrvrHw    := 0;
   END;
   myErr := SNextTypeSRsrc(@mySpBlk);
   IF myErr <> noErr THEN
      MyHandleError(myErr)        {quit searching if no more sResources}
   ELSE
      gTheSlot := mySpBlk.spSlot; {the slot in which the sResource was found}

   {The spsPointer field of mySpBlock now contains a pointer to the }
   { board sResource list. The SGetCString function uses this field }
   { as one of two input fields.}
   mySpBlk.spID := 2;             {sRsrcName entry}
   myErr := SGetCString(@mySpBlk);
   IF myErr <> noErr THEN
      MyHandleError(myErr)
   ELSE BEGIN
      {The spResult field now points to a copy of the cString.}
      MyProcessCardName(gTheSlot, Ptr(mySpBlk.spResult));
      {Free memory allocated by SGetCString.}
      DisposePtr(Ptr(mySpBlk.spResult));
   END;
END;
```

Because the `SGetCString` function allocates memory for a buffer, your application must dispose of the buffer afterward, using the Memory Manager procedure `DisposePtr` (which is described in *Inside Macintosh: Memory*).

Installing and Removing Slot Interrupt Handlers

If your card generates hardware interrupts, you can install a slot interrupt handler to process interrupts from the card. The Slot Manager maintains an interrupt queue for each slot. You use the `SIntInstall` function, described on page 2-70, to install an interrupt handler in the slot interrupt queue. The `SIntRemove` function, described on page 2-71, removes an interrupt handler from the slot interrupt queue.

The `SlotIntQElement` data type, described on page 2-28, defines a slot interrupt queue element. The queue elements are ordered by priority and contain pointers to interrupt handlers. When a slot interrupt occurs, the Slot Manager calls the highest-priority interrupt handler in the slot's interrupt queue. If the interrupt handler returns without servicing the interrupt, the Slot Manager calls the next interrupt handler in the queue, in order of priority, until the interrupt is serviced. If the interrupt is not serviced by any interrupt handler, a system error dialog box is displayed.

Before returning to the Slot Manager, your interrupt handler should set a result code in register D0 to indicate whether the interrupt was serviced. If the interrupt was not serviced, your interrupt handler must return 0. Any value other than 0 indicates that the interrupt was serviced.

The Slot Manager returns to the interrupted task when your interrupt handler indicates that the interrupt was serviced; otherwise, it calls the next lower-priority interrupt handler for that slot. A system error is generated if the last interrupt handler returns to the Slot Manager without servicing the interrupt.

Slot Manager Reference

This section describes the data structures and routines you use to get information about the Slot Manager, expansion cards, and sResources.

Data Structures

This section describes the Slot Manager parameter block structure, the slot information record, the format header record, the slot parameter RAM record, the slot execution parameter block, and the slot interrupt queue element.

Many Slot Manager routines return information from data structures contained in the firmware of cards. See "Firmware," beginning on page 2-7, for a general discussion of these data structures, and see *Designing Cards and Drivers for the Macintosh Family*, third edition, for more detailed information.

Slot Manager Parameter Block

Every Slot Manager function requires a pointer to a Slot Manager parameter block as a parameter and returns an `OSErr` result code. Each routine uses only a subset of the fields of the parameter block. See the individual routine descriptions for a list of the fields used with each routine. The Slot Manager parameter block is defined by the `SpBlock` data type.

```
TYPE SpBlock =
PACKED RECORD                          {Slot Manager parameter block}
   spResult:         LongInt;          {result}
   spsPointer:       Ptr;              {structure pointer}
   spSize:           LongInt;          {size of structure}
   spOffsetData:     LongInt;          {offset or data}
   spIOFileName:     Ptr;              {reserved for Slot Manager}
   spsExecPBlk:      Ptr;              {pointer to SEBlock data structure}
   spParamData:      LongInt;          {flags}
   spMisc:           LongInt;          {reserved for Slot Manager}
   spReserved:       LongInt;          {reserved for Slot Manager}
   spIOReserved:     Integer;          {ioReserved field from SRT}
   spRefNum:         Integer;          {driver reference number}
   spCategory:       Integer;          {Category field of sRsrcType entry}
   spCType:          Integer;          {cType field of sRsrcType entry}
   spDrvrSW:         Integer;          {DrSW field of sRsrcType entry}
   spDrvrHW:         Integer;          {DrHW field of sRsrcType entry}
   spTBMask:         SignedByte;       {sRsrcType entry bit mask}
   spSlot:           SignedByte;       {slot number}
   spID:             SignedByte;       {sResource ID}
   spExtDev:         SignedByte;       {external device ID}
   spHwDev:          SignedByte;       {hardware device ID}
   spByteLanes:      SignedByte;       {valid byte lanes}
   spFlags:          SignedByte;       {flags used by Slot Manager}
   spKey:            SignedByte;       {reserved for Slot Manager}
END;
```

Field descriptions

`spResult` A general-purpose field used to contain the results returned by several different routines.

`spsPointer` A pointer to a data structure. The field can point to an sResource, a data block, or a declaration ROM, depending on the routine being executed.

`spSize` The size of the data pointed to in the `spsPointer` field.

`spOffsetData` The contents of the offset field of an sResource entry. Some routines use this field for other offsets or data.

`spIOFileName` Reserved for use by the Slot Manager.

`spsExecPBlk`	A pointer to an `SEBlock` data structure, which is described on page 2-27.
`spParamData`	On input, a long word containing flags that determine what sResources the Slot Manager searches. When set, bit 0 (the `fAll` flag) indicates that disabled sResources should be included. When set, bit 1 (the `fOneSlot` flag) restricts the search to sResources on a single card. Bit 2 (the `fNext` flag) indicates when set that the routine finds the next sResource. The rest of the bits must be cleared to 0. On output, this field indicates whether the sResource is enabled or disabled (if 0, the sResource is enabled; if 1, it is disabled).
`spMisc`	Reserved for use by the Slot Manager.
`spReserved`	Reserved for future use.
`spIOReserved`	The value of the `ioReserved` field from the sResource's entry in the slot resource table.
`spRefNum`	The driver reference number of the driver associated with an sResource, if there is one.
`spCategory`	The `Category` field of the `sRsrcType` entry (which is described on page 2-10).
`spCType`	The `cType` field of the `sRsrcType` entry.
`spDrvrSW`	The `DrSW` field of the `sRsrcType` entry.
`spDrvrHW`	The `DrHW` field of the `sRsrcType` entry.
`spTBMask`	A mask that determines which `sRsrcType` fields the Slot Manager examines when searching for sResources.
`spSlot`	The number of the slot with the NuBus card containing the requested, or returned, sResource.
`spID`	The sResource ID of the requested, or returned, sResource.
`spExtDev`	The external device identifier. This field allows you to distinguish between devices on a card.
`spHwDev`	The hardware device identifier from the `sRsrcHWDevID` field of the sResource.
`spByteLanes`	The byte lanes used by a declaration ROM.
`spFlags`	Flags typically used by the Slot Manager.
`spKey`	Reserved for use by the Slot Manager.

Listing 2-1 on page 2-18 illustrates how to set values in an `SpBlock` record to disable and enable an sResource. Listing 2-2 on page 2-19 illustrates how to use the values in an `SpBlock` record for searching for sResources.

Slot Information Record

The Slot Manager creates a slot information record for each slot. This structure is defined by the `SInfoRecord` data type.

```
TYPE SInfoRecord =              {slot information record}
PACKED RECORD
   siDirPtr:       Ptr;         {pointer to sResource directory}
   siInitStatusA:  Integer;     {initialization status}
   siInitStatusV:  Integer;     {status returned by vendor }
                                { initialization routine}
   siState:        SignedByte;  {initialization state}
   siCPUByteLanes: SignedByte;  {byte lanes used}
   siTopOfROM:     SignedByte;  {highest valid address in ROM}
   siStatusFlags:  SignedByte;  {status flags}
   siTOConstant:   Integer;     {timeout constant for bus error}
   siReserved:     PACKED ARRAY [0..1] OF SignedByte;
                                {reserved}
   siROMAddr:      Ptr;         {address of top of ROM}
   siSlot:         Char;        {slot number}
   siPadding:      PACKED ARRAY [0..2] OF SignedByte; {reserved}
END;
```

Field descriptions

`siDirPtr` A pointer to the sResource directory (described in "The sResource Directory" on page 2-12).

`siInitStatusA` The initialization status code set by the Slot Manager. A value of 0 indicates the card is installed and operational. Any other value is a Slot Manager error code indicating why the initialization failed.

`siInitStatusV` The initialization status code returned by the card's `PrimaryInit` routine in the `seStatus` field of the `SEBlock` parameter block (described on page 2-27). Negative values cause the card initialization to fail. Values in the range `svTempDisable` ($8000) through `svDisabled` ($8080) are used to temporarily disable a card. See "Enabling and Disabling NuBus Cards" on page 2-17 for more information.

`siState` Reserved for use by the Slot Manager.

`siCPUByteLanes` The byte lanes used by the declaration ROM.

`siTopOfROM` The least significant byte of the address stored in `siROMAddr`.

`siStatusFlags` Slot status flag field set by the Slot Manager. If the `fCardIsChanged` flag (bit 1) is set, the board ID of the installed card does not match the board ID stored in parameter RAM. Other flag bits are reserved.

`siTOConstant` The number of retries that will be performed when a bus error occurs while accessing the declaration ROM. The default is 100.

`siReserved` Reserved for use by the Slot Manager.

`siROMAddr` The highest address in the declaration ROM.

`siSlot` The slot number.

`siPadding` Reserved for use by the Slot Manager.

Format Header Record

The Slot Manager uses a format header record to describe the structure of a card's format block, which is located at the highest address in the slot's NuBus address space. By reading information from the format header record, the Slot Manager can locate and validate the card's declaration ROM. The format header record is defined by the `FHeaderRec` data type.

Note

For more information about the format block, see *Designing Cards and Drivers for the Macintosh Family,* third edition. ◆

```
TYPE FHeaderRec =                {format header record}
PACKED RECORD
   fhDirOffset:   LongInt;       {offset to sResource directory}
   fhLength:      LongInt;       {length in bytes of declaration ROM}
   fhCRC:         LongInt;       {cyclic redundancy check}
   fhROMRev:      SignedByte;    {declaration ROM revision}
   fhFormat:      SignedByte;    {declaration ROM format}
   fhTstPat:      LongInt;       {test pattern}
   fhReserved:    SignedByte;    {reserved; must be 0}
   fhByteLanes:   SignedByte;    {byte lanes used by declaration ROM}
END;
```

Field descriptions

`fhDirOffset` A self-relative signed offset to the sResource directory. This field specifies only bytes accessible by valid byte lanes; as a result, the value in this field might not be the absolute address difference.

`fhLength` The number of valid bytes in the declaration ROM. The Slot Manager uses this value when computing the checksum.

`fhCRC` A checksum that allows the Slot Manager to validate the entire declaration ROM.

`fhROMRev` The current ROM revision level. This field should contain a value in the range 1–9; values greater than 9 cause the Slot Manager to generate the error `smRevisionErr`.

`fhFormat` The format of the declaration ROM. A value of 1 designates the Apple format.

`fhTstPat` A test pattern. This field must contain the value $5A932BC7.

`fhReserved` Reserved. This field must be 0.

`fhByteLanes` A signed byte that specifies which of the four byte lanes to use when communicating with the declaration ROM. Refer to *Designing Cards and Drivers for the Macintosh Family,* third edition, for a list of valid values.

Slot Parameter RAM Record

The Macintosh Operating System reserves eight bytes of parameter RAM for each slot. Six of these bytes are available for card designers to store information. The `SPRAMRecord` data type defines the organization of these bytes of data in parameter RAM. This data structure includes the Apple-defined `BoardID` and six bytes of vendor-specific information.

```
TYPE SPRAMRecord =              {slot parameter RAM record}
PACKED RECORD
   boardID:       Integer;      {Apple-defined board ID}
   vendorUse1:    SignedByte;   {available for vendor use}
   vendorUse2:    SignedByte;   {available for vendor use}
   vendorUse3:    SignedByte;   {available for vendor use}
   vendorUse4:    SignedByte;   {available for vendor use}
   vendorUse5:    SignedByte;   {available for vendor use}
   vendorUse6:    SignedByte;   {available for vendor use}
END;
```

Field descriptions

`boardID` The card identification number assigned by Apple Computer, Inc.

`vendorUse` General-purpose fields that may be used by the card designer.

Slot Execution Parameter Block

The `SGetDriver` and `SExec` functions load and execute code from an sResource. These routines use the slot execution parameter block to exchange information with this code. The slot execution parameter block is defined by the `SEBlock` data type.

```
TYPE SEBlock =                  {slot execution parameter block}
PACKED RECORD
   seSlot:        SignedByte;   {slot number}
   sesRsrcID:     SignedByte;   {sResource ID}
   seStatus:      Integer;      {status of sExecBlock code}
   seFlags:       SignedByte;   {flags}
   seFiller0:     SignedByte;   {filler for word alignment}
   seFiller1:     SignedByte;   {filler}
   seFiller2:     SignedByte;   {filler}
   seResult:      LongInt;      {result of SLoadDriver}
   seIOFileName:  LongInt;      {pointer to driver name}
   seDevice:      SignedByte;   {device to read from}
   sePartition:   SignedByte;   {the partition}
   seOSType:      SignedByte;   {type of OS}
   seReserved:    SignedByte;   {reserved}
   seRefNum:      SignedByte;   {driver reference number}
```

```
   seNumDevices:  SignedByte;    {number of devices to load}
   seBootState:   SignedByte;    {state of StartBoot code}
END;
```

Field descriptions

`seSlot` The slot number containing the code to be executed.

`sesRsrcID` The sResource containing the code to be executed.

`seStatus` The status returned by the executed code. A card's `PrimaryInit` routine returns its initialization status in this field, and the value is stored in the `siInitStatusV` field of the slot information record.

`seFlags` Flags passed to or returned by the executed code.

Name	Bit	Meaning
`fWarmStart`	2	Set if a restart is being performed.
`dRAMBased`	6	Set if the `seResult` field contains a handle to a device driver.

`seFiller0–2` Reserved.

`seResult` A result value returned by the executed code. Normally used to return a pointer or handle to a device driver.

`seIOFileName` An optional pointer to a device driver name.

`seDevice` The device number containing the code to be executed. This field is used when loading code from a device attached to a card.

`sePartition` The partition number containing the code to be executed. This field is used when loading code from a device attached to a card.

`seOSType` The operating system type identifier obtained from parameter RAM. This field is used when loading code from a device attached to a card.

`seReserved` Additional information from parameter RAM, used when loading code from a device attached to a card.

`seRefNum` The driver reference number returned by the loaded device driver.

`seNumDevices` Unused.

`seBootState` A value indicating the relative state of the boot process. During initialization, the Slot Manager passes one of the following constant values in this field:

Name	Value	Meaning
`sbState0`	0	State 0 of the boot process.
`sbState1`	1	State 1 the boot process.

Slot Interrupt Queue Element

The Slot Manager maintains a queue of interrupt handlers for each slot. You use the `SIntInstall` and `SIntRemove` functions (described on page 2-70 and page 2-71, respectively) to install and remove routines in the queue. The `SlotIntQElement` data type defines a slot interrupt queue element.

```
TYPE SlotIntQElement =          {slot interrupt queue element}
RECORD
   sqLink:        Ptr;          {pointer to next queue element}
   sqType:        Integer;      {queue type ID; must be sIQType}
   sqPrio:        Integer;      {priority value in low byte}
   sqAddr:        ProcPtr;      {interrupt handler}
   sqParm:        LongInt;      {optional A1 parameter}
END;
```

Field descriptions

`sqLink`	A pointer to the next queue element. This field is maintained by the Slot Manager.
`sqType`	The queue type identifier, which you set to the defined type `sIQType`.
`sqPrio`	The relative priority level of the interrupt handler. Only the low-order byte of this field is used. The high-order byte must be set to 0. Valid priority levels are 0 through 199. Priority levels 200 through 255 are reserved for Apple devices.
`sqAddr`	A pointer to the interrupt handler.
`sqParm`	An optional value that the Slot Manager places in register A1 before calling the interrupt handler. This field is typically used to store a handle to a driver's device control entry.

Slot Manager Routines

This section describes the routines provided by the Slot Manager. Most of the routines in this section are used to locate sResources or read information from an entry in an sResource. Some of the routines allow you to read and set information about expansion cards, such as their parameter RAM values, and others allow you to manipulate Slot Manager data structures, like the slot resource table.

Because the `SGetCString`, `SGetBlock`, `SGetDriver`, `SExec`, `InitSDeclMgr`, `SInitPRAMRecs`, `SInitSRsrcTable`, and `SPrimaryInit` functions may allocate memory, your application should not call them at interrupt time; however, your can call any other Slot Manager function at interrupt time.

Because each routine uses a subset of the Slot Manager parameter block fields, each routine reference section includes a list of pertinent fields and how they are used.

Parameter block

→	`fieldName`	`FieldType`	Input field.
←	`fieldName`	`FieldType`	Output field.
↔	`fieldName`	`FieldType`	Input/output field.
✕	`fieldName`	`FieldType`	Affected field.

The arrows show whether you provide a value in the field, the routine returns a value in the field, or both. The ✕ symbol designates fields that may be affected by the execution

of the routine. Any value you store in one of these affected fields may be lost. Also, the meaning of these fields upon completion of the routine is undefined; your application should not depend on these values.

Assembly-Language Note

You can call Slot Manager routines using either the `_SlotManager` trap macro with a selector or an individual macro name consisting of the routine name preceded by an underscore. For example, you can call the `SVersion` function using the `_SVersion` macro. Because every routine name macro is equivalent to the `_SlotManager` trap macro that specifies the corresponding routine selector, you will need to know the routine selectors to trace your code in MacsBug. The `_SlotManager` trap macro selector for each routine is included in the routine description and summarized in "Trap Macros," beginning on page 2-99. ◆

Determining the Version of the Slot Manager

Unlike other system software managers, which use the `Gestalt` function to return version information, the Slot Manager includes its own function for providing this information.

SVersion

You can use the `SVersion` function to determine which version of the Slot Manager is in use by the Macintosh Operating System.

```
FUNCTION SVersion (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`LongInt`	The Slot Manager version number.
←	`spsPointer`	`Ptr`	A pointer to additional information.

DESCRIPTION

The `SVersion` function returns the version number of the Slot Manager in the `spResult` field of the Slot Manager parameter block that you point to in the `spBlkPtr` parameter. Version number 1 corresponds to the RAM-based Slot Manager and version number 2 corresponds to the ROM-based Slot Manager. Versions of the Slot Manager prior to System 7 do not recognize the `SVersion` function and return the result code `smSlOOBErr`. The `spsPointer` field is reserved for future use as a pointer to additional information.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SVersion` function are

Trap macro	Selector
`_SlotManager`	$0008

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0008

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smSelOOBErr`	–338	Selector out of bounds or function not implemented

SEE ALSO

For more information on the different versions of the Slot Manager, see "About the Slot Manager" on page 2-15.

Finding sResources

The functions in this section locate sResources in the slot resource table and return pointers to them and additional information about them. The `SRsrcInfo` function is useful for finding the driver reference number of an SResource. The `SGetSRsrc` and `SGetTypeSRsrc` functions are the preferred routines for searching sResources. You can use these functions to step through the sResources and to find disabled as well as enabled sResources. Use the `SNextSRsrc` and `SNextTypeSRsrc` functions with System 6 and earlier versions of the Slot Manager.

SRsrcInfo

You can use the `SRsrcInfo` function to find an sResource. This function also provides additional information about the sResource, such as the driver reference number of the slot device driver.

```
FUNCTION SRsrcInfo (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
←	`spIOReserved`	`Integer`	The value of the slot resource table `ioReserved` field.
←	`spRefNum`	`Integer`	The device driver reference number.
←	`spCategory`	`Integer`	The `Category` field of the `sRsrcType` entry (described on page 2-10).
←	`spCType`	`Integer`	The `cType` field of the `sRsrcType` entry.
←	`spDrvrSW`	`Integer`	The `DrSW` field of the `sRsrcType` entry.
←	`spDrvrHW`	`Integer`	The `DrHW` field of the `sRsrcType` entry.
→	`spSlot`	`SignedByte`	The slot number of the requested sResource.
→	`spId`	`SignedByte`	The sResource ID of the requested sResource.
→	`spExtDev`	`SignedByte`	The external device identifier.
←	`spHwDev`	`SignedByte`	The hardware device identifier.

DESCRIPTION

The `SRsrcInfo` function allows you to find an sResource from the slot resource table and provides additional information, including its driver reference number and the values contained in its `sRsrcType` entry.

You specify an sResource with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter.

The `SRsrcInfo` function returns a pointer to the sResource in the `spsPointer` field and returns information about the sResource type in the `spRefNum`, `spCType`, `spDrvrSW`, `spDrvrHW` fields. The function returns other information about the sResource in the `spIOReserved`, `spRefNum`, and `spHwDev` fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SRsrcInfo` function are

Trap macro	Selector
`_SlotManager`	$0016

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0016

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
smNoMoresRsrcs	–344	Requested sResource not found

SEE ALSO

For more control in finding sResources, you can use the SGetSRsrc function, described next, and the SGetTypeSRsrc function, described on page 2-35.

SGetSRsrc

You can use the SGetSRsrc function to find any sResource, even one that has been disabled.

```
FUNCTION SGetSRsrc (spBlkPtr: SpBlockPtr): OSErr;
```

spBlkPtr A pointer to a Slot Manager parameter block.

Parameter block

←	spsPointer	Ptr	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
↔	spParamData	LongInt	On input: parameter flags. On output: 0 if the sResource is enabled or 1 if disabled.
←	spRefNum	Integer	The slot resource table reference number.
←	spCategory	Integer	The Category field of the sRsrcType entry (described on page 2-10).
←	spCType	Integer	The cType field of the sRsrcType entry.
←	spDrvrSW	Integer	The DrSW field of the sRsrcType entry.
←	spDrvrHW	Integer	The DrHW field of the sRsrcType entry.
↔	spSlot	SignedByte	The slot number.
↔	spId	SignedByte	The sResource ID.
↔	spExtDev	SignedByte	The external device identifier.
←	spHWDev	SignedByte	The hardware device identifier.

DESCRIPTION

The SGetSRsrc function allows you to specify whether the function should include disabled sResources, whether it should continue looking for sResources in higher-numbered slots, and whether it should return information about the specified sResource or the one that follows it.

You specify an sResource with the spSlot, spID, and spExtDev fields of the Slot Manager parameter block you point to in the spBlkPtr parameter. You must also include flags in bits 0, 1, and 2 of the spParamData field as follows:

- Set the fAll flag (bit 0) to search both enabled and disabled sResources. Clear this flag to search only enabled sResources.

- Set the `fOneSlot` flag (bit 1) to search only the specified slot. Clear this flag to search all slots.
- Set the `fNext` flag (bit 2) to return information about the sResource with the next higher sResource ID than the specified sResource (or the first one on the next card if the `fAll` flag is set). Clear this flag to return data about the specified sResource.

The `SGetSRsrc` function returns values in the `spSlot`, `spID`, and `spExtDev` fields corresponding to the sResource that it found. If you cleared the `fNext` flag, these fields retain the values you specified when calling the function. In addition, the function returns 0 in the `spParamData` field if the sResource is enabled or 1 if it is disabled. If you cleared the `fAll` bit, the `spParamData` field always returns the value 0.

The `SGetSRsrc` function also returns a pointer to the sResource in the `spsPointer` field and returns other information about the sResource in the `spRefNum`, `spCategory`, `spCType`, `spDrvrSW`, `spDrvrHW`, and `spHwDev` fields.

SPECIAL CONSIDERATIONS

The `SGetSRsrc` function is available only with version 1 or later of the Slot Manager. You can use the `SVersion` function, described on page 2-30, to determine whether the Slot Manager is version 1 or later.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SGetSRsrc` function are

Trap macro	Selector
`_SlotManager`	$000B

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$000B

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more control in finding sResources, you can also use the `SGetTypeSRsrc` function, described next.

SGetTypeSRsrc

You can use the `SGetTypeSRsrc` function to step through sResources of one type, including disabled ones.

```
FUNCTION SGetTypeSRsrc (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
↔	`spParamData`	`LongInt`	On input: parameter flags. On output: 0 if the sResource is enabled or 1 if disabled.
←	`spRefNum`	`Integer`	The slot resource table reference number.
↔	`spCategory`	`Integer`	The `Category` field of the `sRsrcType` entry (described on page 2-10).
↔	`spCType`	`Integer`	The `cType` field of the `sRsrcType` entry.
↔	`spDrvrSW`	`Integer`	The `DrSW` field of the `sRsrcType` entry.
↔	`spDrvrHW`	`Integer`	The `DrHW` field of the `sRsrcType` entry.
→	`spTBMask`	`SignedByte`	The type bit mask for `sRsrcType` fields.
↔	`spSlot`	`SignedByte`	The slot number.
↔	`spId`	`SignedByte`	The sResource ID.
↔	`spExtDev`	`SignedByte`	The external device identifier.
←	`spHWDev`	`SignedByte`	The hardware device identifier.

DESCRIPTION

The `SGetTypeSRsrc` function allows you to find the next sResource of a certain type, as does the `SNextTypeSRsrc` function, but the `SGetTypeSRsrc` function also allows you to find disabled sResources and to limit searching to a single slot.

You specify an sResource with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter, and you specify the type of the sResource with the `spCategory`, `spCType`, `spDrvrSW`, and `spDrvrHW` fields. You must also use the `spTBMask` field to specify which of these `sRsrcType` fields should not be included in the search:

- Set bit 0 to ignore the `DrHW` field.
- Set bit 1 to ignore the `DrSW` field.
- Set bit 2 to ignore the `cType` field.
- Set bit 3 to ignore the `Category` field.

You must also set the `fAll` flag of the `spParamData` field (bit 0) to search both enabled and disabled sResources or clear this flag to search only enabled ones. Set the `fOneSlot` flag (bit 1) to search only the specified slot, or clear this flag to search all slots. The

SGetTypeSRsrc function does not use the fNext flag (bit 2) because it always searches for the next sResource of the given type.

The SGetTypeSRsrc function returns values in the spSlot, spID, and spExtDev fields corresponding to the sResource that it found, and it returns 0 in the spParamData field if that sResource is enabled or 1 if it is disabled.

The SGetTypeSRsrc function also returns a pointer to the sResource in the spsPointer field and returns other information about the sResource in the spRefNum, spCategory, spCType, spDrvrSW, spDrvrHW, and spHwDev fields.

SPECIAL CONSIDERATIONS

The SGetTypeSRsrc function is available only with version 1 or later of the Slot Manager. You can use the SVersion function, described on page 2-30, to determine whether the Slot Manager is version 1 or later.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SGetTypeSRsrc function are

Trap macro	Selector
_SlotManager	$000C

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When _SlotManager returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $000C

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
smNoMoresRsrcs	–344	Requested sResource not found

SEE ALSO

For information on enabling and disabling sResources, see "Enabling and Disabling sResources" on page 2-18 and the description of the SetSRsrcState function in the next section.

SNextSRsrc

You can use the `SNextSRsrc` function to step through the sResources on a card or from one card to the next.

```
FUNCTION SNextSRsrc (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
←	`spIOReserved`	`Integer`	The value of the slot resource table `ioReserved` field.
←	`spRefNum`	`Integer`	The driver reference number.
←	`spCategory`	`Integer`	The `Category` field of the `sRsrcType` entry (described on page 2-10).
←	`spCType`	`Integer`	The `cType` field of the `sRsrcType` entry.
←	`spDrvrSW`	`Integer`	The `DrSW` field of the `sRsrcType` entry.
←	`spDrvrHW`	`Integer`	The `DrHW` field of the `sRsrcType` entry.
↔	`spSlot`	`SignedByte`	The slot number.
↔	`spId`	`SignedByte`	The sResource ID.
↔	`spExtDev`	`SignedByte`	The external device identifier.
←	`spHWDev`	`SignedByte`	The hardware device identifier.

DESCRIPTION

The `SNextSRsrc` function is similar to the `SRsrcInfo` function, except the `SNextSRsrc` function returns information about the sResource that follows the requested one—that is, the one with the next entry in the sResource directory or the first sResource on the next card. The `SNextSRsrc` function skips disabled sResources.

You specify a particular sResource with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter. The `SNextSRsrc` function finds the next sResource, returns a pointer to it in the `spsPointer` field, and updates the `spSlot`, `spID`, and `spExtDev` fields to correspond to the sResource it found. If there are no more sResources, the `SNextSRsrc` function returns the `smNoMoresRsrcs` result code.

The `SNextSRsrc` function returns other information about the sResource in the `spRefNum`, `spCategory`, `spCType`, `spDrvrSW`, and `spDrvrHW` fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SNextSRsrc` function are

Trap macro	Selector
`_SlotManager`	$0014

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0014

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more control in finding sResources, you can use the `SGetSRsrc` function, described on page 2-33, and the `SGetTypeSRsrc` function, described on page 2-35.

SNextTypeSRsrc

You can use the `SNextTypeSRsrc` function to step through sResources of one type.

```
FUNCTION SNextTypeSRsrc (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
←	`spRefNum`	`Integer`	The slot resource table reference number.
↔	`spCategory`	`Integer`	The `Category` field of the `sRsrcType` entry (described on page 2-10).
↔	`spCType`	`Integer`	The `cType` field of the `sRsrcType` entry.
↔	`spDrvrSW`	`Integer`	The `DrSW` field of the `sRsrcType` entry.
↔	`spDrvrHW`	`Integer`	The `DrHW` field of the `sRsrcType` entry.
→	`spTBMask`	`SignedByte`	The type bit mask for `sRsrcType` fields.
↔	`spSlot`	`SignedByte`	The slot number.
↔	`spId`	`SignedByte`	The sResource ID.
↔	`spExtDev`	`SignedByte`	The external device identifier.
←	`spHWDev`	`SignedByte`	The hardware device identifier.

DESCRIPTION

The `SNextTypeSRsrc` function allows you to find the next sResource, as does the `SNextSRsrc` function, but the `SNextTypeSRsrc` function skips disabled sResources.

You indicate the sResource you want returned by identifying the slot number, sResource ID, and device ID in the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter. You specify the type of the sResource with the `spCategory`, `spCType`, `spDrvrSW`, and `spDrvrHW` fields. You must also use the `spTBMask` to specify which of these `sRsrcType` entry fields should not be included in the search:

- Set bit 0 to ignore the `DrHW` field.
- Set bit 1 to ignore the `DrSW` field.
- Set bit 2 to ignore the `cType` field.
- Set bit 3 to ignore the `Category` field.

The `SNextTypeSRsrc` function returns values in the `spSlot`, `spID`, and `spExtDev` fields corresponding to the sResource that it found.

The `SNextTypeSRsrc` function also returns a pointer to the sResource in the `spsPointer` field and returns other information about the sResource in the `spIOReserved`, `spRefNum`, `spCategory`, `spCType`, `spDrvrSW`, and `spDrvrHW` fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SNextTypeSRsrc` function are

Trap macro	Selector
`_SlotManager`	$0015

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0015

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information on enabling and disabling sResources, see "Enabling and Disabling sResources" on page 2-18 and the description of the `SetSRsrcState` function on page 2-51.

Getting Information From sResources

The Slot Manager provides a number of routines that simplify access to the information in sResources. Most of these routines simply return the value of an sResource entry.

The `SReadDrvrName` function returns the name of an sResource, formatted as a Pascal string and prefixed with a period. You can pass this string to the Device Manager's `OpenSlot` function to open the driver.

The `SReadByte`, `SReadWord`, and `SReadLong` functions return byte, word, or long values from an sResource entry. The `SGetCString`, `SGetBlock`, `SReadStruct`, and `SFindStruct` functions return pointers to larger data types.

SReadDrvrName

You can use the `SReadDrvrName` function to read the name of an sResource in a format you can use to open the driver with Device Manager routines.

```
FUNCTION SReadDrvrName (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spSlot`	`SignedByte`	The slot number.
→	`spID`	`SignedByte`	The sResource ID.
→	`spResult`	`Ptr`	A pointer to the driver name.
✕	`spSize`	`LongInt`	
✕	`spsPointer`	`Ptr`	

DESCRIPTION

The `SReadDrvrName` function reads the name of an sResource, prefixes a period to the value, and converts it to type `Str255`. The final driver name is compatible with the Device Manager's `OpenDriver` function.

You indicate an sResource by identifying the slot number and sResource ID in the `spSlot` and `spID` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter. In your program, you should declare a Pascal string variable and pass a pointer to it in the `spResult` field.

The `SReadDrvrName` function returns the driver name by copying it into the string pointed to by the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the values of the `spSize` and `spsPointer` fields of the parameter block. Your application should not depend on the values returned in these fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadDrvrName` function are

Trap macro	Selector
`_SlotManager`	$0019

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0019

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more information about the device control entry and device driver reference numbers, see the chapter "Device Manager" in this book.

SReadByte

You can use the `SReadByte` function to determine the value of the low-order byte of an sResource entry.

```
FUNCTION SReadByte (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	spResult	LongInt	The contents of the entry byte.
→	spsPointer	Ptr	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	spID	SignedByte	The ID of the sResource entry.
X	spOffsetData	LongInt	
X	spByteLanes	SignedByte	

DESCRIPTION

The `SReadByte` function returns the low-order byte of the offset field of an entry in an sResource. You provide a pointer to the sResource in the `spsPointer` field and the ID of the entry in the `spID` field. The `SReadByte` function returns the value in the low-order byte of the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the values of the `spOffsetData` and `spByteLanes` fields of the parameter block. Your application should not depend on the values returned in these fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadByte` function are

Trap macro	Selector
`_SlotManager`	$0000

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0000

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SReadWord

You can use the `SReadWord` function to determine the value of the low-order word of an sResource entry.

```
FUNCTION SReadWord (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`LongInt`	The contents of the entry word.
→	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	`spID`	`SignedByte`	The ID of the sResource entry.
X	`spOffsetData`	`LongInt`	
X	`spByteLanes`	`SignedByte`	

DESCRIPTION

The `SReadWord` function returns the low-order word of the offset field of an entry in an sResource. You provide a pointer to the sResource in the `spsPointer` field of the Slot Manager parameter block you point to in the `spBlkPtr` parameter, and you provide the ID of the entry in the `spID` field. The `SReadWord` function returns the value in the low-order word of the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the values of the `spOffsetData` and `spByteLanes` fields of the parameter block. Your application should not depend on the values returned in these fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadWord` function are

Trap macro	Selector
`_SlotManager`	$0001

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0001

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
smNoMoresRsrcs	–344	Requested sResource not found

SReadLong

You can use the SReadLong function to determine the value of a long word pointed to by the offset field of an sResource entry.

```
FUNCTION SReadLong (spBlkPtr: SpBlockPtr): OSErr;
```

spBlkPtr A pointer to a Slot Manager parameter block.

Parameter block

←	spResult	LongInt	The contents of the long word.
→	spsPointer	Ptr	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	spID	SignedByte	The ID of the sResource entry.
✕	spSize	LongInt	
✕	spOffsetData	LongInt	
✕	spByteLanes	SignedByte	

DESCRIPTION

The SReadLong function returns the 32-bit value pointed to by the offset field of an sResource entry. In the Slot Manager parameter block you point to in the spBlkPtr parameter, you provide a pointer to the sResource in the spsPointer field and specify the ID of the entry in the spID field. The SReadLong function returns the long word value in the spResult field.

SPECIAL CONSIDERATIONS

This function may alter the values of the spSize, spOffsetData, and spByteLanes fields of the parameter block. Your application should not depend on the values returned in these fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SReadLong function are

Trap macro	Selector
_SlotManager	$0002

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0002

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SGetCString

You can use the `SGetCString` function to determine the value of a string pointed to by the offset field of an sResource entry.

```
FUNCTION SGetCString (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`Ptr`	A pointer to a copy of the `cString` data structure.
→	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	`spID`	`SignedByte`	The ID of the sResource entry.
✕	`spSize`	`LongInt`	
✕	`spOffsetData`	`LongInt`	
✕	`spByteLanes`	`SignedByte`	
✕	`spFlags`	`SignedByte`	

DESCRIPTION

The `SGetCString` function returns a copy of the `cString` data structure pointed to by the offset field of an sResource entry.

You provide a pointer to the sResource in the `spsPointer` field and specify the ID of the entry in the `spID` field.

The `SGetCString` function allocates a memory buffer, copies the value of the `cString` data structure into it, and returns a pointer to it in the `spResult` field. You should dispose of this pointer by using the Memory Manager procedure `DisposePtr`.

SPECIAL CONSIDERATIONS

The `SGetCString` function may alter the values of the `spSize`, `spOffsetData`, `spByteLanes`, and `spFlags` fields of the parameter block. Your application should not depend on the values returned in these fields.

SPECIAL CONSIDERATIONS

The `SGetCString` function allocates memory; your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SGetCString` function are

Trap macro	Selector
`_SlotManager`	$0003

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$0003

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more information about the `cString` data structure, see "Firmware," beginning on page 2-7.

SGetBlock

You can use the `SGetBlock` function to obtain a copy of an `sBlock` data structure pointed to by the offset field of an sResource entry.

```
FUNCTION SGetBlock (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`Ptr`	A pointer to a copy of an `sBlock` data structure (described on page 2-9).
→	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	`spID`	`SignedByte`	The ID of the sResource entry.
✕	`spSize`	`LongInt`	
✕	`spOffsetData`	`LongInt`	
✕	`spByteLanes`	`SignedByte`	
✕	`spFlags`	`SignedByte`	

DESCRIPTION

The `SGetBlock` function returns a copy of the `sBlock` data structure pointed to by the offset field of an sResource entry.

In the parameter block you point to in the `spBlkPtr` parameter, you provide a pointer to the sResource in the `spsPointer` field and specify the ID of the entry in the `spID` field.

The `SGetBlock` function allocates a memory buffer, copies the contents of the `sBlock` data structure into it, and returns a pointer to it in the `spResult` field. You should dispose of this pointer by using the Memory Manager procedure `DisposePtr`.

SPECIAL CONSIDERATIONS

The `SGetBlock` function may alter the values of the `spSize`, `spOffsetData`, `spByteLanes`, and `spFlags` fields of the parameter block. Your application should not depend on the values returned in these fields.

The `SGetBlock` function allocates memory; your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SGetBlock` function are

Trap macro	Selector
`_SlotManager`	$0005

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0005

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SFindStruct

You can use the `SFindStruct` function to obtain a pointer to any data structure pointed to by the offset field of an sResource entry. You might want to use this function, for example, when the data structure type is defined by the card designer.

```
FUNCTION SFindStruct (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

↔	`spsPointer`	`Ptr`	On input: a pointer to an sResource. On output: a pointer to a data structure.
→	`spID`	`SignedByte`	The ID of the sResource entry.
✕	`spByteLanes`	`SignedByte`	

DESCRIPTION

You provide a pointer to the sResource in the `spsPointer` field, and the ID of the entry in the `spID` field. The `SFindStruct` function returns a pointer to the data structure in the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the value of the `spByteLanes` field of the parameter block. Your application should not depend on the value returned in this field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SFindStruct` function are

Trap macro	Selector
`_SlotManager`	$0006

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0006

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information about obtaining a copy of a data structure pointed to by the offset field of an sResource entry, rather than a pointer to the data structure, see the next section, which describes the `SReadStruct` function.

SReadStruct

You can use the `SReadStruct` function to obtain a copy of any data structure pointed to by an sResource entry. You might want to use this function, for example, when the data structure type is defined by the card designer.

```
FUNCTION SReadStruct (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spResult`	`Ptr`	A pointer to a memory block.
→	`spsPointer`	`Ptr`	A pointer to the structure.
→	`spSize`	`LongInt`	The length in bytes of the structure.
X	`spByteLanes`	`SignedByte`	

DESCRIPTION

The `SReadStruct` function copies any arbitrary data structure from the declaration ROM of an expansion card into memory.

You provide a pointer to the structure in the `spsPointer` field and specify the size of the structure in the `spSize` field. You must also allocate a memory block for the result and send a pointer to it in the `spResult` field.

The `SReadStruct` function copies the data structure into the memory block pointed to by the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the value of the `spByteLanes` field of the parameter block. Your application should not depend on the value returned in this field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadStruct` function are

Trap macro	Selector
`_SlotManager`	$0007

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$0007

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information about obtaining a pointer to a data structure pointed to by the offset field of an sResource entry, rather than a copy of the data structure, see the description of the `SFindStruct` function on page 2-48.

Enabling, Disabling, Deleting, and Restoring sResources

The functions in this section are primarily for use by device drivers. The `SetSRsrcState` function enables and disables sResources. The next two functions, `SDeleteSRTRec` and `InsertSRTRec`, delete sResources from and restore them to the slot resource table. The `SUpdateSRT` function updates the slot resource table record for an existing sResource.

SetSRsrcState

You can use the `SetSRsrcState` function to select which sResources are enabled.

```
FUNCTION SetSRsrcState (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spParamData`	`LongInt`	Either a value of 0 to enable the sResource or a value of 1 to disable it.
→	`spSlot`	`SignedByte`	The slot number.
→	`spId`	`SignedByte`	The sResource ID.
→	`spExtDev`	`SignedByte`	The external device identifier.

DESCRIPTION

The `SetSRsrcState` function enables or disables an sResource. All of the Slot Manager functions recognize enabled sResources, while only the `SGetSRsrc` and `SGetTypeSRsrc` functions (described on page 2-33 and page 2-35, respectively) can recognize disabled ones.

You specify the sResource to enable or disable with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter, and you specify whether to enable or disable it in the `spParamData` field. The Slot Manager enables the sResource when the `spParamData` field has a value of 0 and disables it when the field has a value of 1.

SPECIAL CONSIDERATIONS

The `SetSRsrcState` function is available only with version 1 or later of the Slot Manager. You can use the `SVersion` function, described on page 2-30, to determine whether the Slot Manager is version 1 or later.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetSRsrcState` function are

Trap macro	Selector
`_SlotManager`	$0009

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0009

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more information on enabling and disabling sResources, see "Enabling and Disabling sResources" on page 2-18.

For information on finding disabled sResources, see the description of the `SGetSRsrc` function on page 2-33 and the description of the `SGetTypeSRsrc` function on page 2-35.

SDeleteSRTRec

You can use the `SDeleteSRTRec` function to remove an sResource from the slot resource table.

```
FUNCTION SDeleteSRTRec (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spSlot`	`SignedByte`	The slot number.
→	`spId`	`SignedByte`	The sResource ID.
→	`spExtDev`	`SignedByte`	The external device identifier.

DESCRIPTION

The `SDeleteSRTRec` function deletes an sResource from the slot resource table. This routine is typically called by a card's `PrimaryInit` code to delete any sResources that are not appropriate for the system as configured.

SPECIAL CONSIDERATIONS

The `SDeleteSRTRec` function is available only with Manager. You can use the `SVersion` function, described on page 2-30, to determine whether the Slot Manager is version 1 or later.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SDeleteSRTRec` function are

Trap macro	Selector
`_SlotManager`	$0031

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0031

Registers on exit

D0 Result code

SEE ALSO

For more information about the slot resource table, see "About the Slot Manager" on page 2-15. For information about restoring an sResource to the slot resource table, see the `InsertSRTRec` function, described next. For more information on deleting and restoring sResources, see "Deleting and Restoring sResources" on page 2-17.

InsertSRTRec

You can use the `InsertSRTRec` function to add an sResource to the slot resource table.

```
FUNCTION InsertSRTRec (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spsPointer`	`Ptr`	A `NIL` pointer.
→	`spParamData`	`LongInt`	Either a value of 0 to enable the sResource or a value of 1 to disable it.
→	`spRefNum`	`Integer`	The device driver reference number.
→	`spSlot`	`SignedByte`	The slot number.
→	`spId`	`SignedByte`	The sResource ID.
→	`spExtDev`	`SignedByte`	The external device identifier.

DESCRIPTION

The `InsertSRTRec` function installs an sResource from the firmware of a NuBus card into the slot resource table. For example, if the user makes a selection in the Monitors control panel that requires your video card to switch to a new sResource that was deleted by `PrimaryInit` code, you can use the `InsertSRTRec` function to restore that sResource.

You specify an sResource with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter. You must set the `spsPointer` field to `NIL`. Set the `spParamData` field to 1 to disable the restored sResource or to 0 to enable it.

If you place a valid device driver reference number in the `spRefNum` field, the Slot Manager updates the `dCtlDevBase` field in that device driver's device control entry (that is, in the device control entry that has that driver reference number in the `dCtlRefNum` field). The `dCtlDevBase` field contains the base address of the slot device. For a video card this is the base address for the pixel map in the card's `GDevice` record (which is described in *Inside Macintosh: Imaging With QuickDraw*). For other types of cards the base address is optional and defined by the card designer.

The base address consists of the card's slot address plus an optional offset that the card designer can specify using the `MinorBaseOS` or `MajorBaseOS` entries of the sResource. The Slot Manager calculates the base address by using bit 2 (the `f32BitMode` flag) of the `sRsrcFlags` entry of the sResource. As shown in Table 2-4, the Slot Manager first checks the value of bit 2 of the `sRsrcFlags` field, and then it checks for a `MinorBaseOS` entry. If it finds one, it uses this value to create a 32-bit value to store in the `dCtlDevBase` field. If it does not find a `MinorBaseOS` entry, it uses the value in the `MajorBaseOS` entry, if any.

Table 2-4 How the Slot Manager determines the base address of a slot device

sRsrcFlags	MinorBaseOS	MajorBaseOS	Address format
Field missing	$x xxxx	Any or none	$Fs0x xxxx
Field missing	None	$xx xxxx	$sxxx xxxx
Bit 2 is 0	$x xxxx	Any or none	$Fs0x xxxx
Bit 2 is 0	None	$xx xxxx	$sxxx xxxx
Bit 2 is 1	$x xxxx	Any or none	$Fsxx xxxx
Bit 2 is 1	None	$xx xxxx	$sxxx xxxx

Note
In this table, *x* represents any hexadecimal digit and *s* represents a slot number. ◆

SPECIAL CONSIDERATIONS

The `InsertSRTRec` function is available only with version 1 or later of the Slot Manager. You can use the `SVersion` function, described on page 2-30, to determine whether the Slot Manager is version 1 or later.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `InsertSRTRec` function are

Trap macro	Selector
`_SlotManager`	$000A

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $000A

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap
`smUnExBusErr`	–308	Bus error
`smBadRefId`	–330	Reference ID not found in list
`smBadsList`	–331	Bad sResource: Id1 < Id2 < Id3 ... format is not followed
`smReservedErr`	–332	Reserved field not zero
`smSlotOOBErr`	–337	Slot number out of bounds
`smNoMoresRsrcs`	–344	Specified sResource not found
`smBadsPtrErr`	–346	Bad pointer was passed to `SCalcSPointer`
`smByteLanesErr`	–347	`ByteLanes` field in card's format block was determined to be zero

SEE ALSO

For more information about the slot resource table, see "About the Slot Manager" on page 2-15.

For information about deleting an sResource from the slot resource table, see the `SDeleteSRTRec` function, described on page 2-52. For more information on deleting and restoring sResources, see "Deleting and Restoring sResources" on page 2-17.

For more information about the device control entry and device driver reference numbers, see the chapter "Device Manager" in this book.

SUpdateSRT

For system software versions earlier than System 7, you can use the `SUpdateSRT` function to update the slot resource table record for an existing sResource. A new record will be added if the sResource does not already exist in the slot resource table.

```
FUNCTION SUpdateSRT (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spIOReserved`	`Integer`	The value to be stored in the `IOReserved` field of the slot resource table.
→	`spRefNum`	`Integer`	The device driver reference number.
→	`spSlot`	`SignedByte`	The slot number.
→	`spId`	`SignedByte`	The sResource ID.
→	`spExtDev`	`SignedByte`	The external device identifier.

DESCRIPTION

The `SUpdateSRT` function adds or updates an record in the slot resource table. You specify an sResource with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter. If a matching record is found

in the slot resource table, the `RefNum` and `IOReserved` fields of the table are updated. If the record is not found, the sResource is added to the table by reading the appropriate declaration ROM. Updates may be made to enabled sResources only.

SPECIAL CONSIDERATIONS

In System 7, this function was replaced by the `InsertSRTRec` function (described on page 2-54). You should use the `SUpdateSRT` function only if version 1 or later of the Slot Manager is not available. You can use the `SVersion` function, described on page 2-30, to determine whether the Slot Manager is version 1 or later.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SUpdateSRT` function are

Trap macro	Selector
`_SlotManager`	$002B

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$002B

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap
`smEmptySlot`	–300	No card in this slot
`smUnExBusErr`	–308	Bus error
`smBadRefId`	–330	Reference ID not found in list
`smSlotOOBErr`	–337	Slot number out of bounds
`smNoMoresRsrcs`	–344	Specified sResource not found

SEE ALSO

For more information about the slot resource table, see "About the Slot Manager" on page 2-15.

For information about the preferred routine for adding an sResource to the slot resource table, see the `InsertSRTRec` function, described on page 2-54. For information about deleting an sResource from the slot resource table, see the `SDeleteSRTRec` function, described on page 2-52.

Loading Drivers and Executing Code From sResources

The functions in this section allow you to load the device driver associated with an sResource or execute code from an `sExecBlock` data structure. Both of the functions in this section require you to provide extra information in a structure of type `SEBlock`. See "Slot Execution Parameter Block" on page 2-27 for information about the fields of this structure.

SGetDriver

You can use the `SGetDriver` function to load an sResource's device driver.

```
FUNCTION SGetDriver (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`Handle`	A handle to the device driver.
→	`spsExecPBlk`	`Ptr`	A pointer to the `SEBlock`.
→	`spSlot`	`SignedByte`	The slot number.
→	`spID`	`SignedByte`	The sResource ID.
→	`spExtDev`	`SignedByte`	The external device ID.
X	`spSize`	`SignedByte`	
X	`spFlags`	`SignedByte`	

DESCRIPTION

The `SGetDriver` function loads a device driver from an sResource into a relocatable block in the system heap.

You specify an sResource with the `spSlot`, `spID`, and `spExtDev` fields of the Slot Manager parameter block you point to in the `spBlkPtr` parameter, and provide a pointer to a slot execution parameter block in the `spsExecPBlk` field.

The `SGetDriver` function searches the sResource for an `sRsrcLoadRec` entry. If it finds one, it loads the `sLoadDriver` record and executes it. If no `sRsrcLoadRec` entry exists, the `SGetDriver` function looks for an `sRsrcDrvrDir` entry. If it finds one, it loads the driver into memory.

The `SGetDriver` function returns a handle to the driver in the `spResult` field of the parameter block.

SPECIAL CONSIDERATIONS

The `SGetDriver` function allocates memory; your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SGetDriver` function are

Trap macro	Selector
`_SlotManager`	$002D

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register A0 contains a handle to the loaded driver, and register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $002D

Registers on exit

A0 Handle to loaded driver

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more information about sResources, including the `sRsrcDrvrDir` and `sRsrcLoadRec` entry types, see *Designing Cards and Drivers for the Macintosh Family,* third edition.

SExec

You can use the `SExec` function to execute code stored in an `sExecBlock` data structure.

```
FUNCTION SExec (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	`spsExecPBlk`	`Ptr`	A pointer to the `SEBlock`.
→	`spID`	`SignedByte`	The ID of the `sExecBlock` entry in the sResource.
✕	`spResult`	`LongInt`	

DESCRIPTION

The `SExec` function loads `sExecBlock` code from an sResource into the current heap zone, checks its revision level, and executes the code.

You specify the `sExecBlock` by providing a pointer to the sResource in the `spsPointer` field and the ID of the `sExecBlock` entry in the `spID` field. You must also provide in the `spsExecPBlk` field a pointer to a slot execution parameter block. The `SEBlock` structure allows you to provide information about the execution of the `sExecBlock` code.

The `SExec` function passes the `sExecBlock` code a pointer to the `SEBlock` structure in register A0.

SPECIAL CONSIDERATIONS

The `SExec` function allocates memory; your application should not call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SExec` function are

Trap macro	Selector
`_SlotManager`	$0023

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$0023

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smCodeRevErr`	–333	The revision of the code to be executed by `sExec` was wrong
`smCPUErr`	–334	The CPU field of the code to be executed by `sExec` was wrong
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more information about the `sExecBlock` data structure, see page 2-9.

Getting Information About Expansion Cards and Declaration ROMs

The functions in this section return information about slot status or about entire declaration ROMs, instead of single sResources. The `SReadInfo` function returns information from the slot information record maintained by the Slot Manager for a particular slot. See "Slot Information Record," beginning on page 2-24 for a description of the slot information record.

The `SReadFHeader` functions returns a copy of the information in the format block of a card's declaration ROM. The `SCkCardStat` function returns a card's initialization status. The `SCardChanged` function reports whether the card in a particular slot has changed.

The `SFindDevBase` function returns the base address of a slot device.

SReadInfo

You can use the `SReadInfo` function to obtain a copy of the slot information record for a particular slot.

```
FUNCTION SReadInfo (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spResult`	`Pointer`	A pointer to a slot information record.
→	`spSlot`	`SignedByte`	The slot number.
X	`spSize`	`LongInt`	

DESCRIPTION

The Slot Manager maintains a slot information record for each slot. The `SReadInfo` function copies the information from this data structure for the requested slot.

You specify the slot with the `spSlot` parameter. You must also allocate a slot information record, and provide a pointer to it in the `spResult` field. The `SReadInfo` function copies the information in the slot information record maintained by the Slot Manager into the data structure pointed to by the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the contents of the `spSize` field. Your application should not depend on the value returned in this field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadInfo` function are

Trap macro	Selector
`_SlotManager`	$0010

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0010

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smEmptySlot`	–300	No card in this slot

SEE ALSO

For general information about the slot information record, see "About the Slot Manager" on page 2-15. To obtain a pointer to the `SInfoRecord` data structure, instead of a copy of it, see the next section, which describes the `SReadFHeader` function.

SReadFHeader

You can use the `SReadFHeader` function to obtain a copy of the information in the format block of a declaration ROM.

```
FUNCTION SReadFHeader (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spResult`	`Pointer`	A pointer to an `FHeaderRec` data structure (described on page 2-26).
→	`spSlot`	`SignedByte`	The slot number.
✕	`spsPointer`	`Ptr`	
✕	`spSize`	`LongInt`	
✕	`spOffsetData`	`LongInt`	
✕	`spByteLanes`	`SignedByte`	

DESCRIPTION

The `SReadFHeader` function copies the information from the format block of the expansion card in the requested slot to an `FHeaderRec` data structure you provide.

You specify the slot with the `spSlot` parameter. You must also allocate an `FHeaderRec` data structure and provide a pointer to it in the `spResult` field.

The `SReadInfo` function copies the information in the format block into the data structure pointed to by the `spResult` field.

SPECIAL CONSIDERATIONS

This function may alter the contents of the `spsPointer`, `spSize`, `spOffsetData`, and `spByteLanes` fields. Your application should not depend on the values returned in these fields.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadFHeader` function are

Trap macro	Selector
`_SlotManager`	$0013

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$0013

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smEmptySlot`	–300	No card in this slot

SEE ALSO

For general information about the format block, see "The Format Block," beginning on page 2-13. For information about the fields of the format block, see *Designing Cards and Drivers for the Macintosh Family,* third edition.

SCkCardStat

You can use the `SCkCardStat` function to check the initialization status of an expansion card.

```
FUNCTION SCkCardStat (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spSlot`	`SignedByte`	The slot number.
X	`spResult`	`LongInt`	

DESCRIPTION

The `SCkCardStat` function checks the `InitStatusA` field of the slot information record for the expansion card in the designated slot. You specify the slot in the `spSlot` field of the Slot Manager parameter block you point to in the `spBlkPtr` parameter. The `SCkCardStat` function returns the `noErr` result code if the `InitStatusA` field contains a nonzero value.

SPECIAL CONSIDERATIONS

This function may alter the contents of the `spResult` field. Your application should not depend on the values returned in this field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SCkCardStat` function are

Trap macro	Selector
`_SlotManager`	$0018

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0018

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smEmptySlot`	–300	No card in this slot

SEE ALSO

For more information about card initialization, see "About the Slot Manager," beginning on page 2-15.

SCardChanged

You can use the `SCardChanged` function to determine if the card in a particular slot has been changed.

```
FUNCTION SCardChanged (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spSlot`	`SignedByte`	The slot number.
←	`spResult`	`LongInt`	A Boolean signifying whether the card was changed.

DESCRIPTION

The `SCardChanged` function checks if the expansion card in a slot has been changed (that is, if the card's `sPRAMInit` record has been initialized). You specify the slot in the `spSlot` field of the Slot Manager parameter block you point to in the `spBlkPtr` parameter.

The `SCardChanged` function returns a value of `TRUE` in the `spResult` field of the parameter block if the card has been changed.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SCardChanged` function are

Trap macro	Selector
`_SlotManager`	$0022

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0022

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
smEmptySlot	–300	No card in this slot

SFindDevBase

You can use the SFindDevBase function to determine the base address of a slot device.

```
FUNCTION SFindDevBase (spBlkPtr: SpBlockPtr): OSErr;
```

spBlkPtr A pointer to a Slot Manager parameter block.

Parameter block

→	spSlot	SignedByte	The slot number.
→	spId	SignedByte	The sResource ID.
←	spResult	LongInt	The device base address.

DESCRIPTION

The SFindDevBase function returns the base address of a device, using information contained in the sResource. Use of the base address is optional (except for video cards) and device-specific. For a video card this must be the base address for the pixel map in the card's GDevice record (which is described in *Inside Macintosh: Imaging With QuickDraw*.) For other types of cards, the base address is defined by the card designer. The Slot Manager makes no use of this information.

The base address consists of the card's slot address plus an optional offset that the card designer can specify using the MinorBaseOS or MajorBaseOS entries of the sResource. See Table 2-4 on page 2-55 for a description of how the Slot Manager calculates the base address.

You specify the slot in the spSlot field of the Slot Manager parameter block you point to in the spBlkPtr parameter, and the sResource ID with the spId field. The SFindDevBase function returns the base address in the spResult field of the parameter block.

Note
The base address of a slot device is also stored in the dCtlDevBase field of the device control entry. The InsertSRTRec function automatically updates the dCtlDevBase field when a new record is added to the slot resource table. You need to call SFindDevBase only if you used the SUpdateSRTRec function to update the slot resource table. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SFindDevBase` function are

Trap macro	Selector
`_SlotManager`	$001B

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $001B

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smEmptySlot`	–300	No card in this slot

SEE ALSO

For more information about how the device base address is calculated, see the description of the `InsertSRTRec` function on page 2-54.

Accessing Expansion Card Parameter RAM

The Macintosh Operating System reserves six bytes of parameter RAM per slot for any card-specific information that the card designer chooses to store. The functions in this section allow you to read or change the value of these bytes. Both of the functions in this section use the slot parameter RAM record to return the parameter RAM values.

SReadPRAMRec

You can use the `SReadPRAMRec` function to read the parameter RAM information for a particular slot.

```
FUNCTION SReadPRAMRec (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	spSlot	SignedByte	The slot number.
→	spResult	Pointer	A pointer to an SPRAMRecord data structure (described on page 2-27).
✕	spSize	LongInt	

DESCRIPTION

The Macintosh Operating System allocates one SPRAMRecord data structure for each slot in the system parameter RAM. The Slot Manager initializes this structure with the data from the sPRAMInit record on the firmware of the expansion card. The SReadPRAMRec function provides a copy of this information to your application.

You specify the slot number in the spSlot field of the Slot Manager parameter block you point to in the spBlkPtr parameter. You must also allocate a SPRAMRecord data structure and store a pointer to it in the spResult field. The SReadPRAMRec function copies the appropriate parameter RAM information into this data structure.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SReadPRAMRec function are

Trap macro	Selector
_SlotManager	$0011

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When _SlotManager returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0011

Registers on exit

D0 Result code

RESULT CODES

noErr	0	No error
smEmptySlot	–300	No card in this slot

SEE ALSO

For more information about the sPRAMInit record, see *Designing Cards and Drivers for the Macintosh Family,* third edition.

SPutPRAMRec

You can use the `SPutPRAMRec` function to change the values stored in a slot's parameter RAM.

```
FUNCTION SPutPRAMRec (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spsPointer`	`Ptr`	A pointer to an `SPRAMRecord` data structure (described on page 2-27).
→	`spSlot`	`SignedByte`	The slot number.

DESCRIPTION

The `SPutPRAMRec` function allows you to change the values stored in the parameter RAM of a slot.

In the parameter block you point to in the `spBlkPtr` parameter, you specify the slot number with the `spSlot` field and provide the new parameter RAM values in a `SPRAMRecord` data structure pointed to by the `spsPointer` field.

The `SPutPRAMRec` function copies the information from the six vendor-use fields into the parameter RAM for the slot. This function does not copy the `boardID` field, which is Apple-defined.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPutPRAMRec` function are

Trap macro	Selector
`_SlotManager`	$0012

You must set up register D0 with the routine selector and register A0 with the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0012

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smEmptySlot`	–300	No card in this slot

Managing the Slot Interrupt Queue

The Slot Manager maintains an interrupt queue for each slot. If your card generates interrupts, you can install a slot interrupt handler to process the interrupts. You use the `SIntInstall` function to install an interrupt handler in the slot interrupt queue, and the `SIntRemove` function to remove an interrupt handler from the queue.

SIntInstall

You use the `SIntInstall` function to install an interrupt handler in the slot interrupt queue for a designated slot.

```
FUNCTION SIntInstall (sIntQElemPtr: SQElemPtr;
                      theSlot: Integer) : OsErr;
```

`sIntQElemPtr`
A pointer to a slot interrupt queue element record, described on page 2-28.

`theSlot` The slot number.

DESCRIPTION

The `SIntInstall` function adds a new element to the interrupt queue for a slot. You provide a pointer to a slot interrupt queue element in the `sIntQElemPtr` parameter and specify the slot number in `theSlot`.

The Slot Manager calls your interrupt handler using a `JSR` instruction. Your routine must preserve the contents of all registers except A1 and D0, and return to the Slot Manager with an `RTS` instruction. Register D0 should be set to 0 if your routine did not service the interrupt, or any other value if the interrupt was serviced. Your routine should not set the processor priority below 2, and must return with the processor priority equal to 2.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `SIntInstall` function is `_SIntInstall` ($A075).

You must set up register D0 with the slot number and register A0 with the address of the slot queue element. When `_SIntInstall` returns, register D0 contains the result code.

Registers on entry

A0 address of the slot queue element

D0 slot number

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

SIntRemove

You use the `SIntRemove` function to remove an interrupt handler from a slot's interrupt queue.

```
FUNCTION SIntRemove (sIntQElemPtr: SQElemPtr;
                     theSlot: Integer) : OsErr;
```

`sIntQElemPtr`
A pointer to a slot interrupt queue element record, described on page 2-28.

`theSlot` The slot number.

DESCRIPTION

The `SIntRemove` function removes an element from the interrupt queue for a slot. You provide a pointer to a slot interrupt queue element in the `sIntQElemPtr` parameter and specify the slot number in `theSlot`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `SIntRemove` function is `_SIntRemove` ($A076).

You must set up register D0 with the slot number and register A0 with the address of the slot queue element. When `_SIntRemove` returns, register D0 contains the result code.

Registers on entry

A0 address of the slot queue element

D0 slot number

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error

SEE ALSO

For a description of the slot interrupt queue element record, see "Slot Interrupt Queue Element" on page 2-28.

Low-Level Routines

The routines in this section are used internally by the Macintosh Operating System during startup, and as needed by the Slot Manager. They are included here for reference only, and as an aid to debugging. These routines are not required or supported for application-level programming. Applications and device drivers should rely only on the high-level routines described in the previous section, "Slot Manager Routines."

▲ **WARNING**
The routines in this section are internal Macintosh Operating System functions that may be changed without notice by Apple Computer, Inc. These routines may not be supported by future versions of the Operating System. ▲

InitSDeclMgr

This function is used only by the Macintosh Operating System.

```
FUNCTION InitSDeclMgr (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

DESCRIPTION

The `InitSDeclMgr` function initializes the Slot Manager. The contents of the parameter block are undefined. This function allocates the slot information record and checks each slot for a card. If a card is present, the Slot Manager validates the card's firmware and the resulting information is placed in the slot's `sInfoRecord`. For empty slots, or cards that fail to initialize, the Slot Manager stores the appropriate error code in the `initStatusA` field of the `sInfoRecord` for the slot.

SPECIAL CONSIDERATIONS

The `InitSDeclMgr` function allocates memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `InitSDeclMgr` function are

Trap macro	Selector
`_SlotManager`	$0020

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0020

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smUnExBusErr`	–308	A bus error occurred
`smDisposePErr`	–312	An error occurred during execution of `DisposePtr`
`smBadsPtrErr`	–346	Bad `spsPointer` value
`smByteLanesErr`	–347	Bad `spByteLanes` value

SEE ALSO

For more information about Slot Manager initialization, see "About the Slot Manager," beginning on page 2-15.

SCalcSPointer

This function is used only by the Macintosh Operating System.

```
FUNCTION SCalcSPointer (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

↔	`spsPointer`	`Ptr`	A pointer to a byte in declaration ROM.
→	`spOffsetData`	`LongInt`	The offset in bytes to desired pointer.
→	`spByteLanes`	`SignedByte`	The byte lanes used.

DESCRIPTION

The `SCalcSPointer` function returns a pointer to a given byte in the declaration ROM of an expansion card.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SCalcSPointer` function are

Trap macro	Selector
`_SlotManager`	$002C

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $002C

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SCalcStep

This function is used only by the Macintosh Operating System.

```
FUNCTION SCalcStep (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`LongInt`	The function result.
→	`spsPointer`	`Ptr`	A pointer to a byte in declaration ROM.
→	`spByteLanes`	`SignedByte`	The byte lanes used.
→	`spFlags`	`SignedByte`	Flags.

DESCRIPTION

The `SCalcStep` function calculates the field sizes in the block pointed to by `spBlkPtr`. It is used for stepping through the card firmware one field at a time. If the `fConsecBytes` flag is set the function calculates the step value for consecutive bytes; otherwise it calculates it for consecutive IDs.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SCalcStep` function are

Trap macro	Selector
`_SlotManager`	$0028

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0028

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SFindBigDevBase

This function is obsolete.

```
FUNCTION SFindBigDevBase (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spSlot`	`SignedByte`	The slot number.
→	`spId`	`SignedByte`	The sResource ID.
←	`spResult`	`LongInt`	The device base address.

DESCRIPTION

The `SFindBigDevBase` function has been superseded by the `SFindDevBase` function. Currently, both functions execute the same code and return the same result. However, for future compatibility you should use only the `SFindDevBase` function described on page 2-66.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SFindBigDevBase` function are

Trap macro	Selector
`_SlotManager`	$001C

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $001C

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smEmptySlot`	–300	No card in this slot

SEE ALSO

For information about the supported function for finding a device base address, see the description of the `SFindDevBase` function on page 2-66.

SFindSInfoRecPtr

This function is used only by the Macintosh Operating System.

```
FUNCTION SFindSInfoRecPtr (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spResult`	`LongInt`	A pointer to the slot information record.
→	`spSlot`	`SignedByte`	The slot number.

DESCRIPTION

The `SFindSInfoRecPtr` function returns a pointer to the slot information record for a particular slot.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SFindSInfoRecPtr` function are

Trap macro	Selector
`_SlotManager`	$002F

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $002F

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information about the high-level routine for reading the slot information record, see the description of the `SReadInfo` function on page 2-61.

SFindSRsrcPtr

This function is used only by the Macintosh Operating System.

```
FUNCTION SFindSRsrcPtr (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	`spSlot`	`SignedByte`	The slot number of the requested sResource.
→	`spId`	`SignedByte`	The sResource ID of the requested sResource.
X	`spResult`	`LongInt`	

DESCRIPTION

The `SFindSRsrcPtr` function finds an sResource given its slot number and sResource ID. This function ignores disabled sResources.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SFindSRsrcPtr` function are

Trap macro	Selector
`_SlotManager`	$0030

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0030

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information about the high-level routines for locating sResources, see "Finding sResources," beginning on page 2-31.

SGetSRsrcPtr

This function is used only by the Macintosh Operating System.

```
FUNCTION SGetSRsrcPtr (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
→	`spParamData`	`LongInt`	The parameter flags.
→	`spSlot`	`SignedByte`	The slot number of the requested sResource.
→	`spID`	`SignedByte`	The sResource ID of the requested sResource.
→	`spExtDev`	`SignedByte`	The external device identifier.

DESCRIPTION

The `SGetSRsrcPtr` function finds an sResource given its slot number and sResource ID. This function can search disabled sResources.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SGetSRsrcPtr` function are

Trap macro	Selector
`_Slot Manager`	$001D

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $001D

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information about the high-level routines for locating sResources, see "Finding sResources," beginning on page 2-31.

SInitPRAMRecs

This function is used only by the Macintosh Operating System.

```
FUNCTION SInitPRAMRecs (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

DESCRIPTION

The `SInitPRAMRecs` function scans every slot and checks its `BoardId` value against the value stored in PRAM. If the values do not match, the `fCardIsChanged` flag is set and the board sResource is searched for a `PRAMInitData` entry. If one is found, the `sPRAMRecord` for the slot is initialized with the data from the card's `sPRAMInit` record; otherwise it is initialized to 0. The contents of the parameter block are undefined.

SPECIAL CONSIDERATIONS

The `SInitPRAMRecs` function may move memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SInitPRAMRecs` function are

Trap macro	Selector
`_SlotManager`	$0025

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0025

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smUnExBusErr`	–308	A bus error occurred
`smDisposePErr`	–312	An error occurred during execution of `DisposePtr`

SEE ALSO

For more information about Slot Manager initialization, see "About the Slot Manager," beginning on page 2-15.

SInitSRsrcTable

This function is used only by the Macintosh Operating System.

```
FUNCTION SInitSRsrcTable (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

DESCRIPTION

The `SInitSRsrcTable` function initializes the slot resource table. The contents of the parameter block are undefined.

SPECIAL CONSIDERATIONS

The `SInitSRsrcTable` function allocates memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SInitSRsrcTable` function are

Trap macro	Selector
`_SlotManager`	$0029

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0029

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smUnExBusErr`	–308	A bus error occurred
`smDisposePErr`	–312	An error occurred during execution of `DisposePtr`

SEE ALSO

For more information about Slot Manager initialization, see "About the Slot Manager," beginning on page 2-15.

SOffsetData

This function is used only by the Macintosh Operating System.

```
FUNCTION SOffsetData (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

↔	`spsPointer`	`Ptr`	On output: A pointer to the sResource entry.
←	`spOffsetData`	`LongInt`	The contents of the `offset` field.
→	`spID`	`SignedByte`	The ID of the sResource entry.
←	`spByteLanes`	`SignedByte`	The byte lanes from the card's format block.

DESCRIPTION

The `SOffsetData` function returns the value of the offset field of an sResource entry.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SOffsetData` function are

Trap macro	Selector
`_SlotManager`	$0024

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $0024

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For information about high-level routines for getting information from sResources, see the descriptions of the `SReadByte`, `SReadWord`, `SReadLong`, `SGetCString`, `SGetBlock`, `SReadStruct`, and `SFindStruct` functions in "Getting Information From sResources," beginning on page 2-40.

SPrimaryInit

This function is used only by the Macintosh Operating System.

```
FUNCTION SPrimaryInit (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spFlags`	`SignedByte`	Flags passed to the card's `PrimaryInit` code.

DESCRIPTION

Called by the Slot Manager during system startup, the `SPrimaryInit` function executes the code in the `PrimaryInit` entry of each card's board sResource. It passes the `spFlags` byte to the `PrimaryInit` code via the `seFlags` field of the `SEBlock`. The `fWarmStart` bit is set if a restart is being performed.

SPECIAL CONSIDERATIONS

The `SPrimaryInit` function may move memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPrimaryInit` function are

Trap macro	Selector
`_SlotManager`	$0021

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$0021

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smUnExBusErr`	–308	A bus error occurred
`smDisposePErr`	–312	An error occurred during execution of `DisposePtr`
`smBadsPtrErr`	–346	Bad `spsPointer` value
`smByteLanesErr`	–347	Bad `spByteLanes` value

SEE ALSO

For more information about Slot Manager initialization, see "About the Slot Manager," beginning on page 2-15.

SPtrToSlot

This function is used only by the Macintosh Operating System.

```
FUNCTION SPtrToSlot (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

→	`spsPointer`	`Ptr`	A pointer to a byte in declaration ROM.
←	`spSlot`	`SignedByte`	The slot number.

DESCRIPTION

The `SPtrToSlot` function returns the slot number of the card whose declaration ROM is pointed to by `spsPointer`. The value of `spsPointer` must have the form $Fsxx xxxx, where *s* is a slot number and *x* is a hexadecimal number.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPtrToSlot` function are

Trap macro	Selector
`_SlotManager`	$002E

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $002E

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smUnExBusErr`	–308	A bus error occurred
`smBadsPtrErr`	–346	Bad `spsPointer` value

SReadPBSize

This function is used only by the Macintosh Operating System.

```
FUNCTION SReadPBSize (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

↔	`spsPointer`	`Ptr`	A pointer to an sResource (described in "The sResource," beginning on page 2-7).
←	`spSize`	`LongInt`	The size of the `sBlock` data structure.
→	`spID`	`SignedByte`	The ID of the `sBlock` in the sResource.
←	`spByteLanes`	`SignedByte`	The byte lanes from the card's format block.
→	`spFlags`	`SignedByte`	Flags.

DESCRIPTION

The `SReadPBSize` function returns the size of an `sBlock` data structure.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SReadPBSize` function are

Trap macro	Selector
`_SlotManager`	$0026

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0 Address of the parameter block

D0 $00026

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found

SEE ALSO

For more information about the high-level routine for obtaining information from an `sBlock` data structure, see the description of the `SGetBlock` function on page 2-47.

SSearchSRT

This function is used only by the Macintosh Operating System.

```
FUNCTION SSearchSRT (spBlkPtr: SpBlockPtr): OSErr;
```

`spBlkPtr` A pointer to a Slot Manager parameter block.

Parameter block

←	`spsPointer`	`Ptr`	A pointer to a record in the slot resource table.
→	`spID`	`SignedByte`	The ID of the sResource entry.
→	`spExtDev`	`SignedByte`	The external device identifier.
→	`spSlot`	`SignedByte`	The slot.
→	`spFlags`	`SignedByte`	Flags.

DESCRIPTION

The `SSearchSRT` function searches the slot resource table for the record corresponding to the sResource in slot `spSlot` with list `spId` and external device identifier `spExtDev`, and returns a pointer to it in `spsPointer`. If the `fCkForNext` bit of `spFlags` is 0, the function searches for the specified record; if the flag is 1, it searches for the next record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SSearchSRT` function are

Trap macro	Selector
`_SlotManager`	$002A

On entry, register D0 contains the routine selector and register A0 contains the address of the Slot Manager parameter block. When `_SlotManager` returns, register D0 contains the result code.

Registers on entry

A0	Address of the parameter block
D0	$002A

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`smNoMoresRsrcs`	–344	Requested sResource not found
`smRecNotFnd`	–351	Record not found in the slot resource table

Summary of the Slot Manager

Pascal Summary

Constants

```
CONST
   {siStatusFlags field of SInfoRecord}
   fCardIsChanged       = 1;           {card has changed}

   {flags for SSearchSRT}
   fCkForSame           = 0;           {check for same sResource in table}
   fCkForNext           = 1;           {check for next sResource in table}

   {flag passed to card by SPrimaryInit during startup or restart}
   fWarmStart           = 2;           {warm start if set; else cold start}

   {constants for siState field of sInfoRecord}
   stateNil             = 0;           {state}
   stateSDMInit         = 1;           {slot declaration manager init}
   statePRAMInit        = 2;           {sPRAM record init}
   statePInit           = 3;           {primary init}
   stateSInit           = 4;           {secondary init}

   {bit flags for spParamData field of SpBlock}
   fAll                 = 0;           {if set, search all sResources}
   fOneSlot             = 1;           {if set, search in given slot only}
   fNext                = 2;           {if set, search for next sResource}
```

Data Types

```
TYPE SpBlock =                         {Slot Manager parameter block}
   PACKED RECORD
      spResult:         LongInt;       {function result}
      spsPointer:       Ptr;           {structure pointer}
      spSize:           LongInt;       {size of structure}
      spOffsetData:     LongInt;       {offset or data}
      spIOFileName:     Ptr;           {reserved for Slot Manager}
      spsExecPBlk:      Ptr;           {pointer to SEBlock data structure}
      spParamData:      LongInt;       {flags}
```

```
   spMisc:           LongInt;       {reserved for Slot Manager}
   spReserved:       LongInt;       {reserved for Slot Manager}
   spIOReserved:     Integer;       {ioReserved field from SRT}
   spRefNum:         Integer;       {driver reference number}
   spCategory:       Integer;       {Category field of sRsrcType entry}
   spCType:          Integer;       {cType field of sRsrcType entry}
   spDrvrSW:         Integer;       {DrSW field of sRsrcType entry}
   spDrvrHW:         Integer;       {DrHW field of sRsrcType entry}
   spTBMask:         SignedByte;    {sRsrcType entry bit mask}
   spSlot:           SignedByte;    {slot number}
   spID:             SignedByte;    {sResource ID}
   spExtDev:         SignedByte;    {external device ID}
   spHwDev:          SignedByte;    {hardware device ID}
   spByteLanes:      SignedByte;    {valid byte lanes}
   spFlags:          SignedByte;    {flags used by Slot Manager}
   spKey:            SignedByte;    {reserved for Slot Manager}
END;
SpBlockPtr = ^SpBlock;

SInfoRecord =                       {slot information record}
PACKED RECORD
   siDirPtr:         Ptr;           {pointer to sResource directory}
   siInitStatusA:    Integer;       {initialization error}
   siInitStatusV:    Integer;       {status returned by vendor }
                                    { initialization routine}
   siState:          SignedByte;    {initialization state}
   siCPUByteLanes:   SignedByte;    {byte lanes used}
   siTopOfROM:       SignedByte;    {highest valid address in ROM}
   siStatusFlags:    SignedByte;    {status flags}
   siTOConstant:     Integer;       {timeout constant for bus error}
   siReserved:       PACKED ARRAY [0..1] OF SignedByte;  {reserved}
   siROMAddr:        Ptr;           {address of top of ROM}
   siSlot:           Char;          {slot number}
   siPadding:        PACKED ARRAY [0..2] OF SignedByte;  {reserved}
END;
SInfoRecPtr = ^SInfoRecord;

FHeaderRec =                        {format header record}
PACKED RECORD
   fhDirOffset:      LongInt;       {offset to sResource directory}
   fhLength:         LongInt;       {length in bytes of declaration ROM}
   fhCRC:            LongInt;       {cyclic redundancy check}
   fhROMRev:         SignedByte;    {declaration ROM revision}
   fhFormat:         SignedByte;    {declaration ROM format}
```

```
   fhTstPat:         LongInt;       {test pattern}
   fhReserved:       SignedByte;    {reserved; must be 0}
   fhByteLanes:      SignedByte;    {byte lanes used by declaration ROM}
END;
FHeaderRecPtr = ^FHeaderRec;

SPRAMRecord =                       {slot parameter RAM record}
PACKED RECORD
   boardID:          Integer;       {Apple-defined card ID}
   vendorUse1:       SignedByte;    {reserved for vendor use}
   vendorUse2:       SignedByte;    {reserved for vendor use}
   vendorUse3:       SignedByte;    {reserved for vendor use}
   vendorUse4:       SignedByte;    {reserved for vendor use}
   vendorUse5:       SignedByte;    {reserved for vendor use}
   vendorUse6:       SignedByte;    {reserved for vendor use}
END;
SPRAMRecPtr = ^SPRAMRecord;

SEBlock =                           {slot execution parameter block}
PACKED RECORD
   seSlot:           SignedByte;    {slot number}
   sesRsrcId:        SignedByte;    {sResource ID}
   seStatus:         Integer;       {status of sExecBlock code}
   seFlags:          SignedByte;    {flags}
   seFiller0:        SignedByte;    {filler for word alignment}
   seFiller1:        SignedByte;    {filler}
   seFiller2:        SignedByte;    {filler}
   seResult:         LongInt;       {result of SLoadDriver}
   seIOFileName:     LongInt;       {pointer to driver name}
   seDevice:         SignedByte;    {device to read from}
   sePartition:      SignedByte;    {partition}
   seOSType:         SignedByte;    {type of OS}
   seReserved:       SignedByte;    {reserved}
   seRefNum:         SignedByte;    {driver reference number}
   seNumDevices:     SignedByte;    {number of devices to load}
   seBootState:      SignedByte;    {state of StartBoot code}
END;
```

```
SlotIntQElement =                   {slot interrupt queue element}
RECORD
   sqLink:           Ptr;           {pointer to next queue element}
   sqType:           Integer;       {queue type ID; must be sIQType}
   sqPrio:           Integer;       {priority value in low byte}
   sqAddr:           ProcPtr;       {interrupt handler}
   sqParm:           LongInt;       {optional A1 parameter}
END;
SQElemPtr = ^SlotIntQElement;
```

Slot Manager Routines

Determining the Version of the Slot Manager

```
FUNCTION SVersion           (spBlkPtr: SpBlockPtr): OSErr;
```

Finding sResources

```
FUNCTION SRsrcInfo          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SGetSRsrc          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SGetTypeSRsrc      (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SNextSRsrc         (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SNextTypeSRsrc     (spBlkPtr: SpBlockPtr): OSErr;
```

Getting Information From sResources

```
FUNCTION SReadDrvrName      (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SReadByte          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SReadWord          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SReadLong          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SGetCString        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SGetBlock          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SFindStruct        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SReadStruct        (spBlkPtr: SpBlockPtr): OSErr;
```

Enabling, Disabling, Deleting, and Restoring sResources

```
FUNCTION SetSRsrcState      (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SDeleteSRTRec      (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION InsertSRTRec       (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SUpdateSRT         (spBlkPtr: SpBlockPtr): OSErr;
```

Loading Drivers and Executing Code From sResources

```
FUNCTION SGetDriver          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SExec               (spBlkPtr: SpBlockPtr): OSErr;
```

Getting Information About Expansion Cards and Declaration ROMs

```
FUNCTION SReadInfo           (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SReadFHeader        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SCkCardStat         (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SCardChanged        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SFindDevBase        (spBlkPtr: SpBlockPtr): OSErr;
```

Accessing Expansion Card Parameter RAM

```
FUNCTION SReadPRAMRec        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SPutPRAMRec         (spBlkPtr: SpBlockPtr): OSErr;
```

Managing the Slot Interrupt Queue

```
FUNCTION SIntInstall         (sIntQElemPtr: SQElemPtr;
                              theSlot: Integer) : OsErr;
FUNCTION SIntRemove          (sIntQElemPtr: SQElemPtr;
                              theSlot: Integer) : OsErr;
```

Low-Level Routines

```
FUNCTION InitSDeclMgr        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SCalcSPointer       (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SCalcStep           (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SFindBigDevBase     (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SFindSInfoRecPtr    (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SFindSRsrcPtr       (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SGetSRsrcPtr        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SInitPRAMRecs       (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SInitSRsrcTable     (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SOffsetData         (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SPrimaryInit        (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SPtrToSlot          (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SReadPBSize         (spBlkPtr: SpBlockPtr): OSErr;
FUNCTION SSearchSRT          (spBlkPtr: SpBlockPtr): OSErr;
```

C Summary

Constants

```
enum {
   /* StatusFlags field of sInfoArray */
   fCardIsChanged = 1,              /* card has changed */

   /* flags for SearchSRT */
   fCkForSame     = 0,              /* check for same sResource in table */
   fCkForNext     = 1,              /* check for next sResource in table */

   /* flag passed to card by SPrimaryInit during startup or restart */
   fWarmStart     = 2,              /* warm start if set; else cold start */

   /* constants for siState field of sInfoRecord */
   stateNil       = 0,              /* state */
   stateSDMInit   = 1,              /* slot declaration manager init */
   statePRAMInit  = 2,              /* sPRAM record init */
   statePInit     = 3,              /* primary init */
   stateSInit     = 4,              /* secondary init */

   /* bit flags for spParamData field of SpBlock */
   fall           = 0,              /* if set, search all sResources */
   foneslot       = 1,              /* if set, search in given slot only */
   fnext          = 2               /* if set, search for next sResource */
};
```

Data Types

```
typedef struct SpBlock {            /* Slot Manager parameter block */
   long     spResult;               /* function result */
   Ptr      spsPointer;             /* structure pointer */
   long     spSize;                 /* size of structure */
   long     spOffsetData;           /* offset or data */
   Ptr      spIOFileName;           /* reserved for Slot Manager */
   Ptr      spsExecPBlk;            /* pointer to SEBlock structure */
   long     spParamData;            /* flags */
   long     spMisc;                 /* reserved for Slot Manager */
   long     spReserved;             /* reserved for Slot Manager */
   short    spIOReserved;           /* ioReserved field from SRT */
   short    spRefNum;               /* driver reference number */
```

```
   short    spCategory;             /* Category field of sRsrcType entry */
   short    spCType;                /* cType field of sRsrcType entry */
   short    spDrvrSW;               /* DrSW field of sRsrcType entry */
   short    spDrvrHW;               /* DrHW field of sRsrcType entry */
   char     spTBMask;               /* sRsrcType entry bit mask */
   char     spSlot;                 /* slot number */
   char     spID;                   /* sResource ID */
   char     spExtDev;               /* external device ID */
   char     spHwDev;                /* hardware device ID */
   char     spByteLanes;            /* valid byte lanes */
   char     spFlags;                /* flags used by Slot Manager */
   char     spKey;                  /* reserved for Slot Manager */
} SpBlock;
typedef SpBlock *SpBlockPtr;

typedef struct SInfoRecord {        /* slot information record */
   Ptr      siDirPtr;               /* pointer to sResource directory */
   short    siInitStatusA;          /* initialization error */
   short    siInitStatusV;          /* status returned by vendor
                                       initialization routine */
   char     siState;                /* initialization state */
   char     siCPUByteLanes;         /* byte lanes used */
   char     siTopOfROM;             /* highest valid address in ROM */
   char     siStatusFlags;          /* status flags */
   short    siTOConst;              /* timeout constant for bus error */
   char     siReserved[2];          /* reserved */
   Ptr      siROMAddr;              /* address of top of ROM */
   char     siSlot;                 /* slot number */
   char     siPadding[3];           /* reserved */
} SInfoRecord;
typedef SInfoRecord *SInfoRecPtr;

typedef struct FHeaderRec {         /* format header record */
   long     fhDirOffset;            /* offset to sResource directory */
   long     fhLength;               /* length in bytes of declaration ROM */
   long     fhCRC;                  /* cyclic redundancy check */
   char     fhROMRev;               /* declaration ROM revision */
   char     fhFormat;               /* declaration ROM format */
   long     fhTstPat;               /* test pattern */
   char     fhReserved;             /* reserved; must be 0 */
   char     fhByteLanes;            /* byte lanes used by declaration ROM */
} FHeaderRec;
typedef FHeaderRec *FHeaderRecPtr;
```

```
typedef struct SPRAMRecord {        /* slot parameter RAM record */
   short    boardID;                /* Apple-defined card ID */
   char     vendorUse1;             /* reserved for vendor use */
   char     vendorUse2;             /* reserved for vendor use */
   char     vendorUse3;             /* reserved for vendor use */
   char     vendorUse4;             /* reserved for vendor use */
   char     vendorUse5;             /* reserved for vendor use */
   char     vendorUse6;             /* reserved for vendor use */
} SPRAMRecord;
typedef SPRAMRecord *SPRAMRecPtr;

typedef struct SEBlock {            /* slot execution parameter block */
   unsigned char     seSlot;        /* slot number */
   unsigned char     sesRsrcId;     /* sResource ID */
   short             seStatus;      /* status of sExecBlock code */
   unsigned char     seFlags;       /* flags */
   unsigned char     seFiller0;     /* filler for word alignment */
   unsigned char     seFiller1;     /* filler */
   unsigned char     seFiller2;     /* filler */
   long              seResult;      /* result of SLoadDriver */
   long              seIOFileName;  /* pointer to driver name */
   unsigned char     seDevice       /* device to read from */
   unsigned char     sePartition;   /* partition */
   unsigned char     seOSType;      /* type of OS */
   unsigned char     seReserved;    /* reserved */
   unsigned char     seRefNum;      /* driver reference number */
   unsigned char     seNumDevices;  /* number of devices to load */
   unsigned char     seBootState;   /* state of StartBoot code */
} SEBlock;

typedef struct SlotIntQElement {    /* slot interrupt queue element */
   Ptr      sqLink;                /* pointer to next queue element */
   short    sqType;                /* queue type ID; must be sIQType */
   short    sqPrio;                /* priority value in low byte */
   ProcPtr  sqAddr;                /* interrupt handler */
   long     sqParm;                /* optional A1 parameter */
} SlotIntQElement;
typedef SlotIntQElement *SQElemPtr;
```

Slot Manager Functions

Determining the Version of the Slot Manager

```
pascal OSErr SVersion        (SpBlockPtr spBlkPtr);
```

Finding sResources

```
pascal OSErr SRsrcInfo       (SpBlockPtr spBlkPtr);
pascal OSErr SGetSRsrc       (SpBlockPtr spBlkPtr);
pascal OSErr SGetTypeSRsrc   (SpBlockPtr spBlkPtr);
pascal OSErr SNextSRsrc      (SpBlockPtr spBlkPtr);
pascal OSErr SNextTypeSRsrc  (SpBlockPtr spBlkPtr);
```

Getting Information From sResources

```
pascal OSErr SReadDrvrName   (SpBlockPtr spBlkPtr);
pascal OSErr SReadByte       (SpBlockPtr spBlkPtr);
pascal OSErr SReadWord       (SpBlockPtr spBlkPtr);
pascal OSErr SReadLong       (SpBlockPtr spBlkPtr);
pascal OSErr SGetCString     (SpBlockPtr spBlkPtr);
pascal OSErr SGetBlock       (SpBlockPtr spBlkPtr);
pascal OSErr SFindStruct     (SpBlockPtr spBlkPtr);
pascal OSErr SReadStruct     (SpBlockPtr spBlkPtr);
```

Enabling, Disabling, Deleting, and Restoring sResources

```
pascal OSErr SetSRsrcState   (SpBlockPtr spBlkPtr);
pascal OSErr SDeleteSRTRec   (SpBlockPtr spBlkPtr);
pascal OSErr InsertSRTRec    (SpBlockPtr spBlkPtr);
pascal OSErr SUpdateSRT      (SpBlockPtr spBlkPtr);
```

Loading Drivers and Executing Code From sResources

```
pascal OSErr SGetDriver      (SpBlockPtr spBlkPtr);
pascal OSErr SExec           (SpBlockPtr spBlkPtr);
```

Getting Information About Expansion Cards and Declaration ROMs

```
pascal OSErr SReadInfo       (SpBlockPtr spBlkPtr);
pascal OSErr SReadFHeader    (SpBlockPtr spBlkPtr);
pascal OSErr SCkCardStat     (SpBlockPtr spBlkPtr);
pascal OSErr SCardChanged    (SpBlockPtr spBlkPtr);
pascal OSErr SFindDevBase    (SpBlockPtr spBlkPtr);
```

Accessing Expansion Card Parameter RAM

```
pascal OSErr SReadPRAMRec    (SpBlockPtr spBlkPtr);
pascal OSErr SPutPRAMRec     (SpBlockPtr spBlkPtr);
```

Managing the Slot Interrupt Queue

```
pascal OSErr SIntInstall     (SQElemPtr sIntQElemPtr, short theSlot);
pascal OSErr SIntRemove      (SQElemPtr sIntQElemPtr, short theSlot);
```

Low-Level Functions

```
pascal OSErr InitSDeclMgr    (SpBlockPtr spBlkPtr);
pascal OSErr SCalcSPointer   (SpBlockPtr spBlkPtr);
pascal OSErr SCalcStep       (SpBlockPtr spBlkPtr);
pascal OSErr SFindBigDevBase (SpBlockPtr spBlkPtr);
pascal OSErr SFindSInfoRecPtr(SpBlockPtr spBlkPtr);
pascal OSErr SFindSRsrcPtr   (SpBlockPtr spBlkPtr);
pascal OSErr SGetSRsrcPtr    (SpBlockPtr spBlkPtr);
pascal OSErr SInitPRAMRecs   (SpBlockPtr spBlkPtr);
pascal OSErr SInitSRsrcTable (SpBlockPtr spBlkPtr);
pascal OSErr SOffsetData     (SpBlockPtr spBlkPtr);
pascal OSErr SPrimaryInit    (SpBlockPtr spBlkPtr);
pascal OSErr SPtrToSlot      (SpBlockPtr spBlkPtr);
pascal OSErr SReadPBSize     (SpBlockPtr spBlkPtr);
pascal OSErr SSearchSRT      (SpBlockPtr spBlkPtr);
```

Assembly-Language Summary

Data Structures

Slot Manager Parameter Block

0	`spResult`	long	function result
4	`spsPointer`	long	structure pointer
8	`spSize`	long	size of structure
12	`SpOffsetData`	long	offset or data
16	`spIOFileName`	long	reserved for Slot Manager
20	`spsExecPBlk`	long	pointer to SEBlock data structure
24	`spParamData`	long	flags
28	`spMisc`	long	reserved for Slot Manager
32	`spReserved`	long	reserved for Slot Manager
36	`spIOReserved`	word	`ioReserved` field from SRT
38	`spRefNum`	word	driver reference number
40	`spCategory`	word	`Category` field of `sRsrcType` entry
42	`spCType`	word	`cType` field of `sRsrcType` entry
44	`spDrvrSW`	word	`DrSW` field of `sRsrcType` entry
46	`spDrvrHW`	word	`DrHW` field of `sRsrcType` entry
48	`spTBMask`	byte	`sRsrcType` entry bit mask
49	`spSlot`	byte	slot number
50	`spID`	byte	sResource ID
51	`spExtDev`	byte	external device ID
52	`spHwDev`	byte	hardware device ID
53	`spByteLanes`	byte	valid byte lanes
54	`spFlags`	byte	flags used by Slot Manager
55	`spKey`	byte	reserved for Slot Manager

Slot Information Record

0	`siDirPtr`	long	pointer to sResource directory
4	`siInitStatusA`	word	initialization error
6	`siInitStatusV`	word	status returned by vendor initialization routine
8	`siState`	byte	initialization state
9	`siCPUByteLanes`	byte	byte lanes used
10	`siTopOfROM`	byte	highest valid address in ROM
11	`siStatusFlags`	byte	status flags
12	`siTOConst`	word	timeout constant for bus error
14	`siReserved`	word	reserved
16	`siROMAddr`	long	address of top of ROM
20	`siSlot`	byte	slot number
21	`siPadding`	3 bytes	reserved

Format Header Record

0	`fhDirOffset`	long	offset to sResource directory
4	`fhLength`	long	length in bytes of declaration ROM
8	`fhCRC`	long	cyclic redundancy check
12	`fhROMRev`	byte	declaration ROM revision
13	`fhFormat`	byte	declaration ROM format
14	`fhTstPat`	long	test pattern
18	`fhReserved`	byte	reserved; must be 0
19	`fhByteLanes`	byte	byte lanes used by declaration ROM

Slot Parameter RAM Record

0	`boardID`	word	Apple-defined card ID
2	`vendorUse1`	byte	reserved for vendor use
3	`vendorUse2`	byte	reserved for vendor use
4	`vendorUse3`	byte	reserved for vendor use
5	`vendorUse4`	byte	reserved for vendor use
6	`vendorUse5`	byte	reserved for vendor use
7	`vendorUse6`	byte	reserved for vendor use

Slot Execution Parameter Block

0	`seSlot`	byte	slot number
1	`sesRsrcId`	byte	sResource ID
2	`seStatus`	word	status of `sExecBlock` code
4	`seFlags`	byte	flags
5	`seFiller0`	byte	filler for word alignment
6	`seFiller1`	byte	filler
7	`seFiller2`	byte	filler
8	`seResult`	long	result of `SLoadDriver`
12	`seIOFileName`	long	pointer to driver name
16	`seDevice`	byte	device to read from
17	`sePartition`	byte	partition
18	`seOSType`	byte	type of operating system
19	`seReserved`	byte	reserved
20	`seRefNum`	byte	driver reference number
21	`seNumDevices`	byte	number of devices to load
22	`seBootState`	byte	state of `StartBoot` code

Slot Interrupt Queue Element

0	`sqLink`	long	pointer to next queue element
4	`sqType`	word	queue type ID; must be `sIQType`
6	`sqPrio`	word	priority value in low `byte`
8	`sqAddr`	long	pointer to interrupt handler
12	`sqParm`	long	optional A1 parameter

Trap Macros

Trap Macros Requiring Routine Selectors

`_SlotManager`

Selector	Routine
$0000	SReadByte
$0001	SReadWord
$0002	SReadLong
$0003	SGetCString
$0005	SGetBlock
$0006	SFindStruct
$0007	SReadStruct
$0008	SVersion
$0009	SetSRsrcState
$000A	InsertSRTRec
$000B	SGetSRsrc
$000C	SGetTypeSRsrc
$0010	SReadInfo
$0011	SReadPRAMRec
$0012	SPutPRAMRec
$0013	SReadFHeader
$0014	SNextSRsrc
$0015	SNextTypeSRsrc
$0016	SRsrcInfo
$0018	SCkCardStat
$0019	SReadDrvrName
$001B	SFindDevBase
$001C	SFindBigDevBase
$001D	SGetSRsrcPtr
$0020	InitSDeclMgr
$0021	SPrimaryInit
$0022	SCardChanged
$0023	SExec
$0024	SOffsetData
$0025	SInitPRAMRecs
$0026	SReadPBSize
$0028	SCalcStep

Selector	Routine
$0029	SInitSRsrcTable
$002A	SSearchSRT
$002B	SUpdateSRT
$002C	SCalcSPointer
$002D	SGetDriver
$002E	SPtrToSlot
$002F	SFindSInfoRecPtr
$0030	SFindSRsrcPtr
$0031	SDeleteSRTRec

Result Codes

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap
`smEmptySlot`	–300	No card in this slot
`smCRCFail`	–301	CRC check failed
`smFormatErr`	–302	The format of the declaration ROM is wrong
`smUnExBusErr`	–308	A bus error occurred
`smBLFieldBad`	–309	A valid `fhByteLanes` field was not found
`smDisposePErr`	–312	An error occurred during execution of `DisposePtr`
`smNoBoardsRsrc`	–313	There is no board sResource
`smNoBoardId`	–315	There is no board ID
`smInitStatVErr`	–316	The `InitStatusV` field was negative after `PrimaryInit`
`smBadRefId`	–330	Reference ID was not found in the given list
`smBadsList`	–331	The IDs are not in ascending order
`smReservedErr`	–332	A reserved field was not zero
`smCodeRevErr`	–333	The revision of the code to be executed by `sExec` was wrong
`smCPUErr`	–334	The CPU field of the code to be executed by `sExec` was wrong
`smsPointerNil`	–335	The `spsPointer` value is `NIL`: no list is specified
`smNilsBlockErr`	–336	The physical block size of an sBlock was zero
`smSlotOOBErr`	–337	The given slot was out of bounds or does not exist
`smSelOOBErr`	–338	Selector out of bounds or function not implemented
`smCkStatusErr`	–341	Status of slot is bad
`smGetDrvrNamErr`	–342	An error occurred during execution of `_sGetDrvrName`
`smNoMoresRsrcs`	–344	Requested sResource not found
`smBadsPtrErr`	–346	Bad `spsPointer` value
`smByteLanesErr`	–347	Bad `spByteLanes` value
`smRecNotFnd`	–351	Record not found in the slot resource table