GET_NAME ($27)

This function returns the filename of the currently running application.

To get the compete pathname of the current application, use GET_PREFIX for prefix number 1/, and affix that prefix to the file name returned by this call.

❖ *Note:* If your program uses SET_PREFIX to reset prefix 1/ to anything other than its initial value, be sure it first uses GET_PREFIX on 1/ and saves the results. Otherwise there may be no way to recover the full pathname of the current application.

GET_NAME ($27)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	data_buffer	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the current application's file name.

Possible ProDOS 16 errors

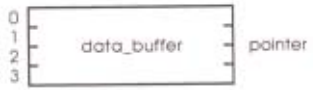
$07 ProDOS is busy

GET_BOOT_VOL ($28)

This function returns the name of the volume from which the file named PRODOS was last executed. PRODOS is the operating system loader; it loads both ProDOS 16 and ProDOS 8 into memory. Execution of PRODOS may occurr

- at system startup
- from a reboot
- by execution from an Applesoft BASIC dash (—) command
- by loading PRODOS into memory at $002000 and executing a JMP to that address

The volume name returned by this call is identical to the prefix specified by */. See Chapter 5.



GET_BOOT_VOL ($28)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	data_buffer	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the boot volume's name.

Possible ProDOS 16 errors

$07 ProDOS is busy

QUIT ($29)

Calling this function terminates the present application. It also closes all open files, sets the current system file level to zero, and deallocates any installed interrupt handlers. ProDOS 16 can then do one of three things:

- launch a file specified by the quitting program
- launch a file specified by the user
- automatically launch a program specified in the quit return stack

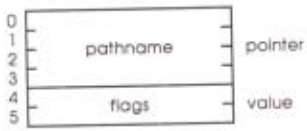
The **quit return stack** is a table maintained in memory by ProDOS 16. It provides a convenient means for a shell program to pass execution to subsidiary programs (even other shells), while ensuring that control eventually returns to the shell.

For example, a program selector may push its User ID onto the quit return stack whenever it launches an application (by making a `QUIT` call). That program may or may not specify yet another program when it quits, and it may or may not push its own User ID onto the quit return stack. Eventually, however, when no more programs have been specified and no others are waiting for control to return to them, the program selector's User ID will be pulled from the stack and it will be executed once again.

Two `QUIT` call parameters control these options, as follows:

1. Pathname pointer:
   a. If the pathname pointer in the parameter block points to a pathname of nonzero length, the indicated program is loaded and executed.
   b. If pathname is null (zero) or if it points to a null pathname (one with a zero length byte), ProDOS 16 pulls a User ID from the quit return stack and executes the program with that ID.
   c. If pathname is null *and* the quit return stack is empty, ProDOS 16 executes a built-in interactive dispatcher that allows the user to
      - reboot the computer
      - execute the file `SYSTEM/START` on the boot disk
      - enter the name of the next application to launch

2. Flag word:
   The flag word contains two boolean values: a return flag and a restart-from-memory flag.
   a. If the return flag value is TRUE (bit 15=1), the User ID of the program making the `QUIT` call is pushed onto the quit return stack. If the return flag is FALSE, no ID is pushed onto the stack.
   b. If the value of the restart-from-memory flag is TRUE (bit 14=1), the program is capable of being restarted from a dormant state in the computer's memory. If the restart-from-memory flag is FALSE, the program must always be reloaded from disk when it is run. Every time a program's User ID is pushed onto the quit return stack, the information from this flag is saved along with it. The System Loader uses this information when it reloads or **restarts** the program later (see Chapter 17).



QUIT ($29)
Parameter block

❖ *Note:* The pathname designated in this call may be a partial pathname with an implied or explicit prefix number. However, the total length of the expanded prefix (the full pathname except for the file name) must not exceed 64 characters. Other ProDOS 16 calls do not restrict pathname length as severely.

Further details of the operation of the `QUIT` function are explained in Chapter 5.

Parameter description

Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the next file to execute.
$04–$05	flags	**parameter name:** flag word **size and type:** word value **range of values:** $0000–$C000 Two boolean flags in a 16-bit field. The bits are defined as follows: **bit** — **significance** 15 — if = 1, place calling program's User ID on return stack 14 — if = 1, calling program may be restarted from memory 13–0 — (reserved)

Possible ProDOS 16 errors

`QUIT` never returns to the caller. Therefore, it cannot return an error. However, other parts of ProDOS 16 may. For example, if an interrupting program (such as a desk accessory) ignores established conventions and uses a `QUIT` call, error $07 (ProDOS is busy) may occur. For programming rules covering such specialized applications, see *Programmer's Introduction to the Apple IIGS*.

If a nonfatal error occurs, execution passes to an interactive routine that allows the user to select another program to launch. Errors that may cause this include:

$07	ProDOS is busy
$40	Invalid syntax
$46	File not found
$5C	Not an executable file
$5D	Operating system not available
$5E	Cannot deallocate /RAM
$5F	Return stack overflow

Fatal errors cause execution to halt. For example, If the `QUIT` call results in the loading of a ProDOS 8-based application, and if the system disk has been altered with a different version of ProDOS 8 (file P8), it is a fatal error ($11). Execution halts and the following message is displayed on the screen:

```
Wrong OS version $0011
```

If the `QUIT` call results in the loading of a ProDOS 16-based application that is too large to fit in the available memory or that for some other reason cannot be loaded, execution halts and the following message is displayed on the screen:

```
Can't run next application. Error=$XXXX
```

where *$XXXX* is an error code—typically a Tool Locator, Memory Manager, or System Loader error code.

GET_VERSION ($2A)

This function returns the version number of the currently running ProDOS 16 operating system.

The returned version number is placed in the version parameter field. Both byte and bit values are significant. It has this format:

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	B	Major Release No.							Minor Release No.							

- Byte 0 is the minor release number (= 0 for ProDOS 16 version 1.0)
- Byte 1 is the major release number (= 1 for ProDOS 16 version 1.0)
- B (the most significant bit of byte 1) = 0 for final releases
  = 1 for all prototype releases

GET_VERSION ($2A)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	version	**parameter name:** version **size and type:** word result (high-order byte zero) **range of values:** $0000–$FFFF The version number of ProDOS 16.

Possible ProDOS 16 errors

$07	ProDOS is busy

Chapter 13

Interrupt Control Calls

These calls allocate and deallocate interrupt handling routines.

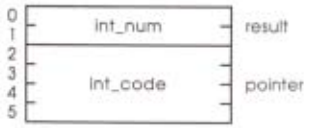
The ProDOS 16 interrupt control calls are described in the following order:

Number	Function	Purpose
$31	ALLOC_INTERRUPT	installs an interrupt handler
$32	DEALLOC_INTERRUPT	removes an interrupt handler

ALLOC_INTERRUPT ($31)

This function places the address of an interrupt-handling routine into the interrupt vector table. You should make this call before enabling the hardware that can cause the interrupt. It is your responsibility to make sure that the routine is installed at the proper location and that it follows interrupt conventions (see Chapter 7).

The returned `int_num` is a reference number for the handler. Its only use is to identify the handler when deallocating it; you must refer to a routine by its interrupt handler number to remove it from the system (with `DEALLOC_INTERRUPT`).



ALLOC_INTERRUPT ($31)
Parameter block

When ProDOS 16 receives an interrupt, it polls the installed handlers in sequence, according to their order in the interrupt vector table. The first handler installed has the highest priority. Each new handler installed is added to the end of the table; each one deallocated is removed from the list and the table is compacted.

❖ *Note:* Under ProDOS 8, the interrupt handler number is equal to the handler's position in the polling sequence. By contrast, the value of `int_num` under ProDOS 16 is unrelated to the order in which handlers are polled.

Parameter description

Offset	Label	Description
$00–$01	int_num	**parameter name:** interrupt handler number **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the interrupt handler by ProDOS 16.
$02–$05	int_code	**parameter name:** interrupt code **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of the interrupt handler routine.

Possible ProDOS 16 errors

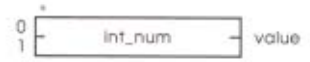
$07	ProDOS is busy
$25	Interrupt vector table full
$53	Invalid parameter

DEALLOC_INTERRUPT ($32)

This function clears the entry (specified by *int_num*) for an interrupt handler from the interrupt vector table.

Important You must disable the associated interrupt hardware before making this call. A fatal error will result if a hardware interrupt occurs after its entry has been cleared from the vector table.

DEALLOC_INTERRUPT has no effect on the order of the polling sequence for the remaining handlers. Any subsequently allocated handlers will be added to the end of the polling sequence.



DEALLOC_INTERRUPT ($32)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	int_num	**parameter name:** interrupt handler number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the interrupt handler by ProDOS 16.

Possible ProDOS 16 errors

$07	ProDOS is busy
$53	Invalid parameter

Part III

The System Loader

The System Loader is an Apple IIGS tool set that works closely with ProDOS 16. It is responsible for loading all program code and data into the Apple IIGS memory. It is capable of static and dynamic loading and relocating of code and data segments, subroutines, and libraries.

Chapter 14 explains in general terms how the System Loader works. Chapter 15 details some of its functions and data structures. Chapter 16 gives programming suggestions for using the System Loader. Chapter 17 shows how to make loader calls and describes each call in detail. See Appendix E for a complete list of System Loader error codes.

Chapter 14

Introduction to the System Loader

This chapter gives a basic picture of the System Loader, defines some of the important terms needed to explain what the loader does, describes its interactions with the Memory Manager, and presents an outline of the procedures it follows when loading a program into memory. Additional related terms are defined in the Glossary.

What is the System Loader?

The System Loader is a set of software routines that manages the loading of program segments into the Apple IIGS. It is an Apple IIGS tool set; as such, it is independent of ProDOS 16. However, it works very closely with ProDOS 16 and with the Memory Manager, another tool set. The System Loader has several improvements over the loading method under ProDOS 8 on other Apple II computers:

- It makes loading easier and more convenient. Under ProDOS 8, the only automatic loading is performed by the boot code, which searches the boot disk for the first .SYSTEM file (type $FF) and loads it into location $2000. If a system program needs to call another application it must do all the work itself, either by making ProDOS 8 calls or by providing its own loader. On the Apple IIGS, calls to the System Loader perform the task more simply.
- It is a *relocating loader*: it loads relocatable programs at any available location in memory. Under ProDOS 8, a program must be loaded at a fixed memory address, or at an address specified by the system program that does the loading. The relocating loader relieves the programmer of the burden (and restriction) of deciding where to load programs.
- It is a *segment loader*: it can load different segments of a program independently, to use memory efficiently.
- It is a *dynamic loader*: it can load certain program segments as they are needed during execution, rather than at boot time only.

The System Loader handles files generated by the **APW Linker**; the linker handles files produced by an Apple IIGS **assembler** or **compiler.** The linker, assembler, and compilers are part of the **Apple IIGS Programmer's Workshop** (APW), a powerful and flexible set of development programs designed to help programmers produce Apple IIGS applications efficiently and conveniently. See Chapter 6 of this manual for more information and references on Apple IIGS Programmer's Workshop.

Loader terminology

The System Loader is a program that processes **load files.** Load files are ProDOS 16 applications or other types of program files. They contain machine-language code or data and must follow object module format (OMF) specifications, as defined in the *Apple IIGS Programmer's Workshop Reference*. Each load file consists of **load segments** that can be loaded into memory independently.

Load segments can be either **static** or **dynamic**. A program's static segments are loaded into memory at initial load time (when the program is first started up); they must stay in memory until the program is complete. Dynamic load segments, on the other hand, are not placed in memory at initial load time; they are loaded as needed during program execution. Dynamic loading can be automatic (through the **Jump Table**) or manual (at the specific request of the application through System Loader function calls). When a dynamic segment is no longer needed by the program that called it, it can be **purged**, or deleted, by the Memory Manager.

Segments can be **absolute**, **relocatable**, or **position-independent**. An absolute segment must be loaded into a specific location in memory, or it will not function properly. A relocatable segment can execute correctly wherever the System Loader places it. Least restricted of all is a position-independent segment; its functioning is totally unaffected by its location in memory. It can even be moved from one location to another between executions. Most Apple IIGS code is relocatable, but not position-independent.

Load files can contain segments of various kinds. Some segments consist of program code or data; others provide location information to the loader. The **Jump Table segment**, when loaded into memory, provides a mechanism by which segments in memory can trigger the loading of other needed segments. Each load file can have only one Jump Table segment. A load file can also have one segment called the **Pathname segment**, which provides a cross-reference between file numbers (in the Jump Table segment) and pathnames (on disk) of dynamic segments. A third special type of segment is the **initialization segment**. It contains any code that has to be executed first, before the rest of the segments are loaded.

When the System Loader is called to load a program, it loads all static load segments including the Jump Table segment and the Pathname segment. The Jump Table and the **Pathname Table** are constructed from these two segments, respectively. During this process, a **Memory Segment Table** is also constructed in memory. These three tables are discussed in more detail in the next chapter.

A controlling program is a program that requests the System Loader to perform an **initial load** on another major program, usually an application. The **User ID Manager** assigns a unique identification number (User ID) to that application, so the loader may quickly locate all of the application's segments if necessary. A switcher is an example of a controlling program; ProDOS 16 and the APW Shell are also controlling programs. A word processor is an example of an application.

Interface with the Memory Manager

The System Loader and the Memory Manager work closely together. The Memory Manager is an Apple IIGS tool set (firmware program) that is responsible for allocating memory in the Apple IIGS. It provides space for load segments, tells the System Loader where to place them, and moves segments around within memory when additional space is needed.

When the System Loader loads a program segment, it calls the Memory Manager to allocate a corresponding **memory block**. Memory blocks have attributes that are closely related to the load segments in them. If the program segment is *static*, its memory block is marked as **unpurgeable** (meaning that its contents cannot be erased) and **fixed** (meaning that its position cannot be changed), as long as the program is running. If the program segment is *dynamic*, its memory block is initially marked as **purgeable** but locked (temporarily unpurgeable and fixed; subject to change during execution of the program). If the dynamic segment is *position-independent*, its memory block is marked as **movable**; otherwise, it is fixed.

To **unload** a segment, the System Loader calls the Memory Manager to make the coresponding memory block purgeable. If the controlling program wishes to unload *all* segments associated with a particular application (for example, at shutdown), it calls the System Loader's User Shutdown function, which in turn calls the Memory Manager to purge the application's memory blocks.

To speed up execution of a finder or switcher that may need to rapidly reload shut-down applications, the User Shutdown function can optionally put an application into a dormant state. The loader calls the Memory Manager to purge the application's *dynamic* segments, and make all *static* segments purgeable. This process frees space but keeps the unloaded application's essential segments in memory. However, if for any reason memory runs out and the Memory Manager is forced to purge one of those static segments, that application can no longer be used—the next time it is needed, it must be loaded from its disk file. See "User Shutdown" and "Restart" in Chapter 17.

❖ *Note:* Strictly speaking, load segments are never *purged* or *locked*; those are actions taken on the memory blocks that hold the segments. For simplicity, however, this manual may in certain cases apply terms such as purged or locked to segments.

A typical load segment will be placed in a memory block that is

Locked
Fixed
Purge Level = 0 (if the segment is static)
Purge Level = 3 (if the segment is dynamic)

Depending on other requirements the segment may have, such as alignment in memory, the load segment-memory block relationship may be more complex. Table 14-1 shows all possible relationships between the two that may hold at load time. The direct-page/stack segment has special characteristics described in Chapter 6.

Table 14-1
Load-segment/memory-block relationships (at load time)

Load Segment Attribute	Memory Block Attribute
static	unpurgeable, fixed (unmovable)
dynamic	purgeable, locked
absolute (ORG > 0)	fixed address
relocatable	(no specific relation)
position-independent	not fixed (movable)
not postion-independent	fixed (unmovable)
KIND = $11	fixed-bank
BANKSIZE = 0	may cross bank boundary
BANKSIZE = $10 000	may not cross bank boundary
ALIGN = 0	not bank- or page-aligned†
ALIGN = $100	page-aligned†
ALIGN = $10 000	bank-aligned†
direct-page/stack (KIND = $12)	purgeable, fixed-bank ($00), page-aligned

†Alignment may also be controlled by the value in the BANKSIZE field—see Appendix D.

❖ *Note:* ORG, KIND, BANKSIZE and ALIGN are segment header fields, described in Appendix D of this manual and under "Object Module Format" in *Apple IIGS Programmer's Workshop Reference.*

A memory block can be purged through a call to the System Loader, but other attributes can be changed only through Memory Manager calls. Memory block properties useful to an application may include

- Start location
- Size of block
- User ID (identifies the application the block is part of)
- Purge level (0 to 3: 0 = unpurgeable, 3 = most purgeable)

These properties may be accessed either through the Memory Segment Table (see Chapter 15), or through the block's memory handle, which is part of the Memory Segment Table. If the memory handle is NIL (points to a null pointer), the memory block has been purged.

Loading a relocatable segment

The following brief description of parts of the operation of the System Loader shows how the linker, loader, and Memory Manager work together to produce and load a relocatable program segment. Figure 14-1 diagrams the process in a simplified form.

Load-file structure

Load files conform to a subset of object module format (OMF). In OMF, each module (file) consists of one or more segments; each segment is further made up of one or more **records.** In a load file specifically, each segment (apart from specialized segments such as the load file tables described in Chapter 15) consists of a header followed by program code or data, in turn followed (if the segment is relocatable) by a **relocation dictionary.** The relocation dictionary is created by the linker as it converts an object segment into a load segment. The program code or data consists of two types of records: LCONST records, which hold all code and data, and DS records, used for filling space with zeros. The relocation dictionary consists of two general types of records: RELOC records, which give the loader the information it needs to resolve local (intrasegment) references, and INTERSEG records, which give the loader the information it needs to resolve external (intersegment) references. cRELOC, cINTERSEG, and SUPER records are also found in relocation dictionaries—they are compressed versions of RELOC and INTERSEG records. The detailed formats of all OMF records are presented in *Apple IIGS Programmer's Workshop Reference.*

When a relocatable segment is loaded into memory, it is placed at a location determined by the Memory Manager. Furthermore, only the first part of the segment (the program code itself) is loaded into the part of memory reserved by the Memory Manager; the relocation dictionary, if present, is loaded into a buffer or work area used by the loader. After loading the segment, the loader *relocates* it, using the information in the relocation dictionary.

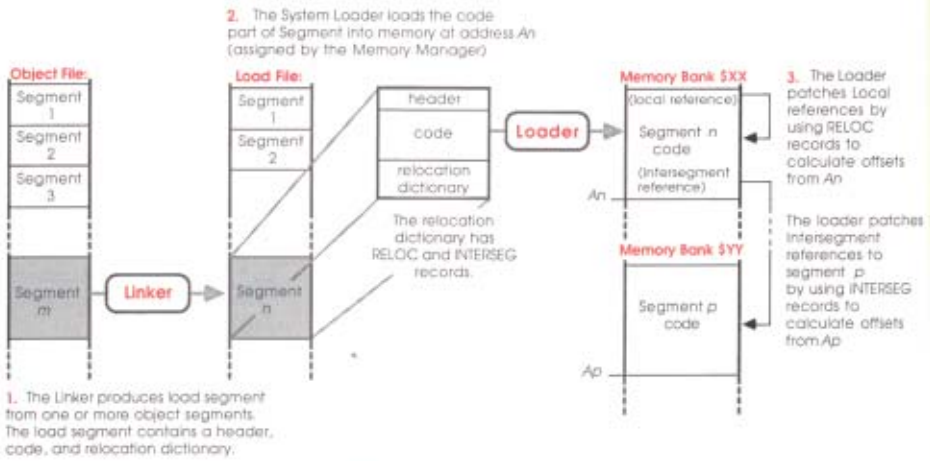


Figure 14-1
Loading a relocatable segment

Relocation

After the System Loader has placed a load segment in memory, it must (unless the segment consists of absolute code) relocate its address references. **Relocation** describes the processing of a load segment so that it will execute properly at the memory location at which it has been loaded. It consists of **patching** (substituting the proper values for) address operands that refer to locations both within and external to the segment. The relocation dictionary part of the segment contains all the information needed by the loader to do this patching. Relocation is performed as follows:

1. Local references in the load segment (coded in the original object file as offsets from the beginning of the segment) are patched from `RELOC` records in the relocation dictionary. Using the starting address of the segment (available from the Memory Manager through the Memory Segment Table), the loader adds that address to each offset, so that the correct memory address is referenced.

2. External references (references to other segments) are coded in the original object module as global variables (subroutine names or entry points). The linker and loader handle them as follows:

   a. If the reference is to a static segment, the *linker* will have calculated the proper file number, segment number, and offset of the referenced (external) segment, and placed that information in an `INTERSEG` record in the relocation dictionary. When the load segment is loaded, the *loader* uses the `INTERSEG` record and the memory location of the external segment (available from the Memory Manager through the Memory Segment Table), and then patches the external reference with the proper memory address of the external segment.

   b. If the reference is to a dynamic segment, the *linker* will have created a slightly different `INTERSEG` record: instead of referencing the file number, segment, and offset of the referenced external segment itself, the `INTERSEG` record references the file number, segment number, and offset of an entry in the *Jump Table*. Therefore, when the load segment is loaded, the *loader* patches the reference to point to the Jump Table entry. That entry, in turn, is what transfers control to the external segment at its proper memory address (if and when the referenced segment is loaded).

   The Jump Table and the reasons for this indirect referencing are described further in Chapter 15. The main point of interest here is that, when it performs relocation, the loader doesn't care whether an intersegment reference is to a static or to a dynamic segment—it treats both in exactly the same way.

The System Loader performs several other functions when it loads dynamic segments, including searching for the name of the segment in the Pathname Table before loading, and patching the appropriate Jump Table entry afterward. These and other functions are described in more detail in the next two chapters.

Chapter 15

System Loader Data Tables

This chapter describes the data tables set up in memory during a load, to provide cross-reference information to the loader. The Memory Segment Table allows the loader to keep track of which segments have been loaded and where they are in memory. The **Jump Table** allows programs to reference routines in dynamic segments that may not currently be in memory. The Pathname Table provides a cross-reference between file numbers and file pathnames of dynamic segments. The **Mark List** speeds relocation by keeping track of relocation dictionaries.

Memory Segment Table

The Memory Segment Table is a linked list, each entry of which describes a memory block known to the System Loader. Memory blocks are allocated by the Memory Manager during loading of segments from a load file, and each block corresponds to a single load segment. Figure 15-1 shows the format of each entry in the Memory Segment Table.

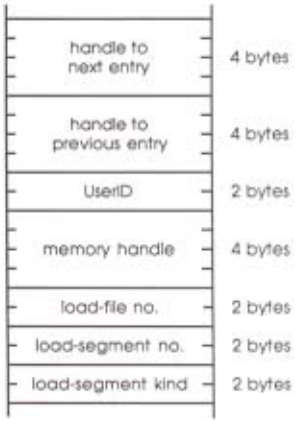


Figure 15-1
Memory Segment Table entry

The fields have the following meanings:

Handle to next entry: The memory handle of the next entry in the Memory Segment Table. This number is 0 for the last entry.

Handle to previous entry: The memory handle of the previous entry in the Memory Segment Table. This number is 0 for the first entry.

User ID: The identification number assigned to the memory block this segment inhabits. Normally, the User ID is available directly from the Memory Manager through the memory handle. However, if the block has been purged its handle is NIL and the User ID must be read from this field.

Memory handle: The identifying number of the memory block, obtained from the Memory Manager. Additional memory block information is available through this handle. This handle is NIL if the block has been purged.

Load-file number: The number of the load file from which the segment was obtained. If the segment is in the initial load file, the number is 1.

Load-segment number: The segment number of the segment in the load file.

Load-segment kind: The value of the `KIND` field in the load segment's header. Segment kinds are described in Appendix D.

Jump Table

When a program (load file) is initially loaded, only the static load segments are placed in memory; at that point the System Loader has all the information it needs to resolve all symbolic references among them. Until a dynamic segment is loaded, however, the loader cannot resolve references to it because it does not know where in memory it will be. Thus static segments may be directly referenced (by each other and by dynamic segments), but dynamic segments can be referenced only through JSL (Jump to Subroutine Long) calls to the Jump Table. This section describes how that mechanism works.

The Jump Table is a structure that allows a program to reference dynamic segments. It consists of the Jump Table Directory and one or more Jump Table segments.

On disk, Jump Table segments are load segments (of kind $02), created by the linker to resolve references to dynamic segments. Any load file or run-time library file may contain a Jump Table segment.

In memory, the Jump Table Directory is created by the loader as it loads Jump Table segments. The Jump Table Directory is a linked list, each entry of which points to a single Jump Table segment encountered by the loader. Figure 15-2 shows the format of an entry in the Jump Table Directory.

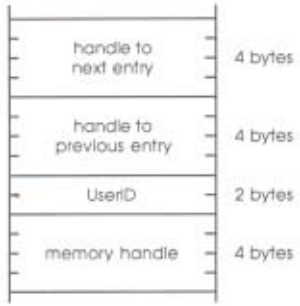


Figure 15-2
Jump Table Directory entry

The fields have the following meanings:

Handle to next entry: The memory handle of the next entry in the Jump Table Directory. This number is 0 for the last entry.

Handle to previous entry: The memory handle of the previous entry in the Jump Table Directory. This number is 0 for the first entry.

User ID: The identification number assigned to the Jump Table segment that this Directory entry refers to.

Memory handle: The handle of the memory block containing the Jump Table segment that this Directory entry refers to.

Like the Directory, the individual Jump Table segments consist of a series of entries. The next three subsections describe the creation, loading, and use of a single Jump Table *segment* entry. The entry is used to resolve a single JSL instruction in a program segment.

❖ *Note:* Throughout this manual, the term *Jump Table entry* refers to a Jump Table *segment* entry, not a Jump Table *directory* entry.

Creation of a Jump Table entry

The Jump Table load segment is created by the linker, as the linker processes an object file. Each time the linker encounters a `JSL` to a routine in an external dynamic segment, it creates an `INTERSEG` record in the relocation dictionary of the load segment, and (if it has not done so already) an entry for that routine in the Jump Table segment. The `INTERSEG` record links the `JSL` to the Jump Table entry that was just created. Figure 15-3 shows the format of the Jump Table entry that the linker creates. See also section a of Figure 15-5.

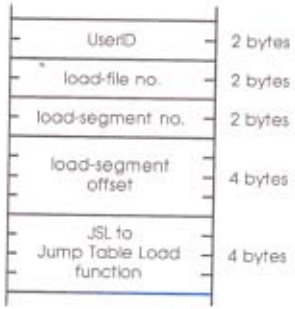


Figure 15-3
Jump Table entry (unloaded state)

The fields have the following meanings:

User ID: The User ID of the referenced dynamic segment.

Load-file number: The load-file number of the referenced dynamic segment.

Load-segment number: The load-segment number of the referenced dynamic segment.

Load-segment offset: The location of the referenced address within the referenced dynamic segment.

JSL to Jump Table Load function: A long subroutine jump to the Jump Table Load function. The Jump Table Load function is described in Chapter 17.

The final entry in a Jump Table segment has a load-file number of zero, to indicate that there are no more entries in the segment.

Modification at load time

At load time, the loader places the program segment and the Jump Table segment into memory (it does not yet load the referenced dynamic segment). To link the Jump Table segment with any other Jump Table segments it may have loaded, it creates the Jump Table Directory. The Jump Table is now complete.

Using the information in the `INTERSEG` record, the loader patches the `JSL` instruction in the program segment so that it references the proper part of the Jump Table in memory. It also patches the actual address of the Jump Table Load function into the Jump Table entry. The Jump Table entry is now in its *unloaded state*. See section A of Figure 15-5.

Use during execution

During program execution, when the `JSL` instruction in the original load segment is encountered, the following sequence of events takes place:

1. Control transfers to the proper Jump Table entry.
2. The `JSL` in the entry transfers control to the System Loader's Jump Table Load function.
3. The Jump Table Load function gets the load-file number, load-segment number, and load-segment offset of the dynamic segment from the Jump Table entry. Then it gets the file pathname of the dynamic segment from the Pathname Table.
4. The System Loader loads the dynamic segment into memory.
5. The loader changes the dynamic segment's entry in the Jump Table to its *loaded state*. The loaded state is identical to the unloaded state, except that the `JSL` to the Jump Table Load function is replaced by a `JML` (unconditional Jump Long) to the external reference itself. Figure 15-4 shows the format for the loaded state.

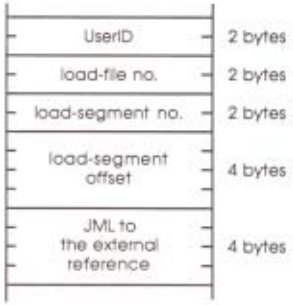


Figure 15-4
Jump Table entry (loaded state)

6. The loader transfers control to the dynamic segment. When the new segment has finished its task (typically it is a subroutine and exits with an `RTL`, Return from Subroutine Long), control returns to the statement following the original `JSL` instruction. See section B of Figure 15-5.

Jump Table diagram

Figure 15-5 is a simplified diagram of how the Jump Table works. It follows the creation, loading, and use of a single Jump Table entry, needed to resolve a single instruction in load segment *n*. The instruction is a JSL to a subroutine named *routine* in dynamic segment *a*.

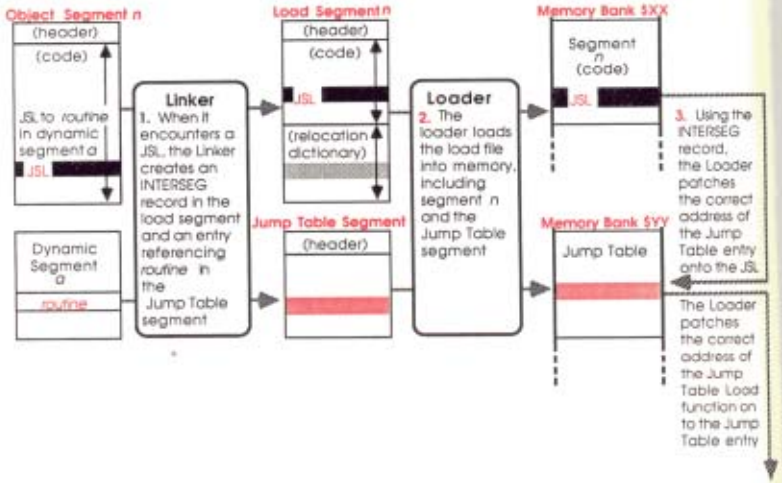


Figure 15-5A
How the Jump Table works

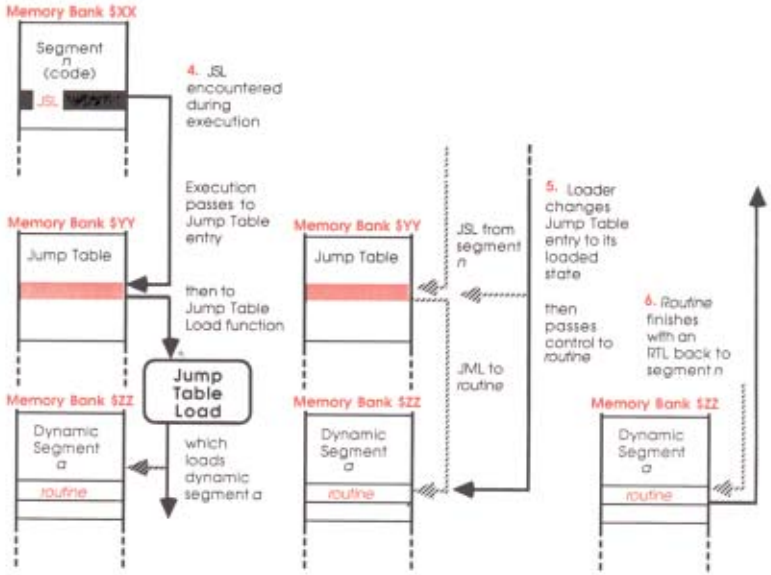


Figure 15-5B
How the Jump Table works (continued)

Pathname Table

The Pathname Table provides a cross-reference between file numbers and file pathnames, to help the System Loader find the load segments that must be loaded dynamically. The Pathname Table is a linked list of individual pathname entries; it starts with an entry for the pathname of the initial load file, and includes any entries from segments of kind $04 (Pathname segments) that the loader encounters during the load. Also, if run-time library files are referenced during program execution, their own pathname segments are linked to the original one.

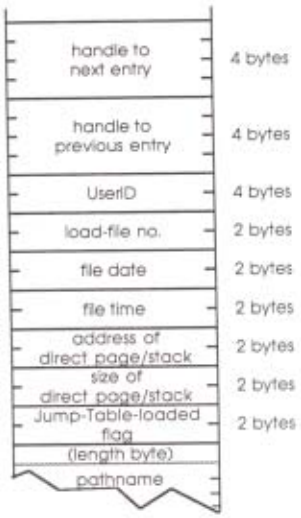


Figure 15-6
Pathname Table entry

A load file's Pathname *segment* (`KIND` = $04) is constructed by the linker and contains one entry for each run-time library file referenced by the file. Each entry consists of a load-file number, file date and time, and a pathname. The exact format for Pathname-segment entries is given in *Apple IIGS Programmer's Workshop Reference*.

The Pathname *Table* is constructed in memory by the loader; its entries are identical to Pathname segment entries, except that each also contains two link handles, a User ID field, and direct-page/stack information. Figure 15-6 shows the format of a Pathname Table entry.

The fields have the following meanings:

Handle to next entry: the memory handle of the next entry in the Pathname Table. For the last entry, the value of the handle is 0.

Handle to previous entry: the memory handle of the previous entry in the Pathname Table. For the first entry, the value of the handle is 0.

User ID: the ID associated with this entry. Generally, each load file has a unique User ID, and a single entry in the Pathname Table. Each new run-time library encountered during execution is assigned the application's User ID.

File number: the number assigned to a specific load file by the linker. File number 1 is reserved for the initial load file.

File date: the date on which the file was last modified.

File time: the time at which the file was last modified.

The *file date* and *file time* are ProDOS 16 directory items retrieved by the linker during linking. They are included in the Pathname Table as an identity check on run-time library files (they are ignored for other file types). To ensure that the run-time library file used at program execution is the same one originally linked by the linker, the System Loader compares these values to the directory entries of the run-time library file to be loaded. If they do not match, the System Loader will not load the file.

Direct-page/stack address: the starting address of the buffer allocated (at initial load) for the file's direct page (zero page) and stack.

Direct-page/stack size: the size (in bytes) of the buffer allocated for the file's direct page and stack.

The direct-page/stack address and size fields are in the Pathname Table to allow the Restart function to more quickly resurrect a dormant application (see "Restart" and "User Shutdown" in Chapter 17). These two fields are ignored for run-time library files.

Jump-Table-loaded flag: a flag that indicates whether the load file's Jump Table segment has been loaded. It's value is always TRUE (1) for initial load files; its initial value is FALSE (0) for runtime libraries.

File pathname: the full or partial pathname of this entry. Partial pathnames with the following two prefix numbers are stored in the table unchanged (unexpanded):

1/ = the current application's subdirectory

2/ = system library subdirectory (initially `/V/SYSTEM/LIBS`, where `/V/` is the boot volume name)

The System Loader expands all other partial pathnames before storing them in the Pathname Table.

The pathname is a *Pascal string*, meaning that it consists of a length byte (of value *n*) followed by an ASCII string (*n* bytes long) that is the pathname itself.

Mark List

The Mark List is a table constructed by the System Loader to keep track of where, within a load file, each segment's relocation dictionary is located. The Mark List speeds relocation because, once a code segment is loaded, the loader needn't search through it again to find the relocation dictionary—the Mark List allows it to go directly to the location of the segment's relocation dictionary.

Figure 15-7 shows the format of the Mark List.

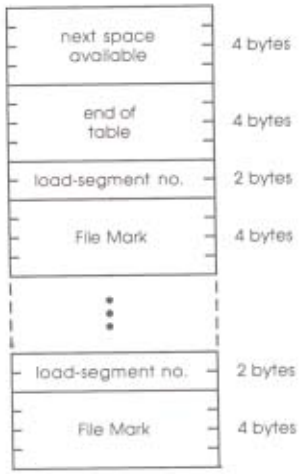


Figure 15-7
Mark List format

The fields have the following meanings:

Next available space: The relative offset (in bytes from the beginning of the Mark List) to the next wempty space in the Mark List.

end of table: The relative offset to the end of the Mark List—in other words, its size in bytes.

load-segment number: The number of the load segment whose relocation dictionary is specified in the following field.

File Mark: The relative offset (in bytes from the beginning of the load file) to the relocation dictionary of the segment specified in the preceding field. *File Mark* in this table has the same meaning as *Mark*, or *current file position*, in ProDOS 16 (see Chapter 2).

Chapter 16

Programming With the System Loader

This chapter discusses how you can use the capabilities of the System Loader at several different levels, depending on the complexity of the programs you wish to write. It also gives requirements for designing controlling programs (shells)—programs that control the loading and execution of other programs.

Programming suggestions for ProDOS 16 are in Chapter 6 of this manual. More general information on how to program for the Apple IIGS is available in *Programmer's Introduction to the Apple IIGS*. For language-specific programming instructions, consult the appropriate language manual in the Apple IIGS Programmer's Workshop (see "Apple IIGS Programmer's Workshop" in Chapter 6).

Static programs

The functioning of the System Loader is completely transparent to simple applications. Any program that is loaded into memory in its entirety at the beginning of execution, and which does not call any other programs or routines that must be loaded during run time, need not know anything about the System Loader. If such a static program is in proper object module format, it will be automatically loaded, relocated, and executed whenever it is called.

Programming with dynamic segments

You may write Apple IIGS programs that use memory more efficiently than the simple application described above. If your program is divided into static and dynamic segments, only the static segments are loaded when the program is started up. Dynamic segments are loaded only as needed during execution, and the memory they occupy is available again when they are no longer needed.

Dynamic loading also is transparent to the typical application; no System Loader commands are necessary to invoke it. If you segment your program as you write the source code, and if you define the proper segments as dynamic and static when the object code is linked, the loading and execution of dynamic segments will be completely automatic.

Because segments are specified as static or dynamic at link time, you may experiment with several configurations of a single program after it has been assembled. For example, you might first run the program as a single static segment, then run several different static-dynamic combinations to see which gives the best performance for the amount of memory required. In this way the same program could be tailored to different machines with different memory configurations.

In general, the least-used parts of a program are the best candidates for dynamic segments, since loading and executing a dynamic segment takes longer than executing a static segment. Furthermore, making a large, seldom-used segment dynamic might make the *initial* load of a program faster, since the static part of the load file will be smaller.

Dynamic segments can be used as *overlays* (segments with the same fixed starting address that successively occupy the same memory area), but this structure is not recommended for the Apple IIGS. If all segments are instead relocatable, the Memory Manager has more flexibility in finding the best place for each allocated segment, whether or not it happens to be a space formerly occupied by another segment of the same program.

Programming with run-time libraries

❖ *Note:* Although the System Loader supports run-time libraries, initial releases of other Apple IIGS system software may not. This section discusses how to program for run-time libraries when full support for them becomes available.

A run-time library is a load file. Like other libraries or subroutine files, it contains general routines that may be referenced by a program. As with other libraries, references to it are resolved by the linker.

Unlike other libraries, however, its segments are not physically *appended* to the program that references it; instead, the linker creates a reference to it in the program's load file. The run-time library remains on disk (or in memory) as an independent load file; when one of its segments is referenced during program execution, the segment is then loaded and executed dynamically.

As with dynamic segments, loading of run-time library segments is transparent to the typical application. No System Loader commands are necessary to invoke it; as far as the loader is concerned, the run-time library is just another load file with dynamic segments.

The most useful difference between run-time library segments and other dynamic segments is that they may be *shared* among programs. Routines for drawing or calculating, dialog boxes or graphic images, or any other segments that might be of use to more than one program can be put into run-time libraries. And, being dynamic, they help keep the initial load file small.

Important In using both run-time libraries and other dynamic segments, make sure that the volumes containing all needed segments and libraries are on line at run time. A fatal error occurs if the System Loader cannot find a dynamic segment it needs to load.

User control of segment loading

To make the greatest use of the System Loader, programs may make loader calls directly. For most applications this is not necessary, but for programs with specialized needs the System Loader offers this capability.

Your application can manually load other segments using the Load Segment By Number and Load Segment By Name calls. Load Segment By Number requires the application to know the load file number and segment number of the segment to load; Load Segment By Name uses the load file pathname and segment name of the desired segment. Both require User ID as an input; the User ID for each segment and each pathname are available from the Memory Segment Table and Pathname Table, respectively. Other segment information available through the Get Load Segment Info call.

One advantage of manually loading a dynamic segment is that it can be referenced in a more direct manner. Automatically-loaded dynamic segments can be referenced only through a `JSL` to the Jump Table; however, if the segment is data such as a table of values, you may wish to simply access those values rather than pass execution to the segment. By manually loading the segment, locking it, and dereferencing its memory handle (obtaining a pointer to the start of the segment), you may then directly reference any location in the table. Of course, since the loader does not resolve any symbolic references in the manually loaded segment, the application must know its exact structure.

A program is responsible for managing the segments it loads. That is, it must unload them (using Unload Segment By Number) or make them purgeable and unlocked (through Memory Manager calls) when they are no longer needed.

Designing a controlling program

A program may cause the loading of another program in one of two ways:

- The program can make a ProDOS 16 `QUIT` call. ProDOS 16 and the System Loader remove the quitting program from memory, then load and execute the specified new program.
- The program can call the System Loader directly. The loader loads the specified new program without unloading the original program, then hands control back to the original program.

A controlling program is an application that loads and executes other programs using the second method. It uses powerful System Loader calls that are normally reserved for use by ProDOS 16. Certain types of finders, switchers and shells may be controlling programs; if you are writing such a program you should follow the conventions given here.

An application needs to be a controlling program only if it must *remain* in memory after it calls another program. If it is necessary only that control *return* to the original program after the called program quits, the ProDOS 16 `QUIT` call is sufficient for that. For example, a finder, which always returns after an application that it calls quits, does not have to be a controlling program; it is not in memory while the application is running. On the other hand, the Apple IIGS Programmer's Workshop Shell, which has functions needed by the subprograms that it calls, *is* a controlling program; it remains active in memory while its subprograms execute.

❖ *Note:* Subprograms are file type $B5, called **shell applications.** They too must follow certain conventions. See "Object Module Format" in *Apple IIGS Programmer's Workshop Reference*, and *Programmer's Introduction to the Apple IIGS.*

If you write a controlling program, please follow these guidelines:

1. The controlling program should request a User ID for the subprogram, either directly from the User ID Manager or indirectly, by calling the System Loader's Initial Load function with an input User ID (`MainID`) of zero. The controlling program should then pass the returned User ID to the subprogram in the accumulator.
2. Use the System Loader's Initial Load function to first load any subprogram. The function returns the subprogram's starting address and User ID to your controlling program; the controlling program can then decide when and where to pass control to the subprogram.
3. When your controlling program passes execution to the subprogram, it may also pass parameters and an identifier string. The pointer to the buffer containing that information should be placed in the X (high-order word) and Y (low-order word) registers. The buffer should contain an 8-character shell identifier string, followed by a null-terminated string consisting of the complete input line or command line through which the subprogram was called.

❖ *Note:* ProDOS 16 does not pass an identifier string or command line when it launches a shell application. It places zeros in the X and Y registers.

4. Your controlling program is responsible for establishing the appropriate input and output vectors for its subprograms. For example, when ProDOS 16 launches a $B5 file, it sets the global I/O hooks to point to the firmware Pascal drivers for 80-column screen and keyboard. The identifier string your controlling program passes to the subprogram allows it to check to make sure it is running in the proper I/O environment (that is, under your controlling program and not another).
5. The controlling program should observe the ProDOS 16 conventions for register initialization and direct-page/stack allocation. See Chapter 6.
6. If you want your controlling program to support shell applications that terminate with a ProDOS 16 `QUIT` call, the controlling program must intercept all ProDOS 16 calls. That way when a subprogram quits, the controlling program, rather than ProDOS 16, regains control.
7. When the shell application exits back to the controlling program, it leaves an error code in the accumulator. Two values are reserved: $0000 means no error, and $FFFF means a non-specific shell-application error. Your controlling program and subprograms may define any other errors as needed.
8. Your controlling program is totally responsible for the subprogram's disposition. When the subprogram is finished, the controlling program must remove it from memory and release all resources associated with its User ID. The best way to do this is to call the System Loader's User Shutdown function.
9. If the subprogram *itself* manually loads other programs, then it is also a controlling program and must observe all the conventions listed here. in particular, it must be certain to dispose of all memory resources associated with the subprogram that *it* loaded, before itself quitting and passing control back to the original controlling program.

   The practice of using shell applications as controlling programs is discouraged.

Shutting down and restarting applications

Through alternate use of the User Shutdown and Restart functions, a controlling program can rapidly switch execution among several applications. If none of an application's static segments have been removed from memory since shutdown, Restart brings the application back rapidly because disk access is not required.

However, only software that is **restartable** can be restarted in this way. Restartable software reinitializes its variables every time it gains control; it makes no assumptions about the state of the machine when it starts up. If a subprogram exits with a `QUIT` call, it specifies whether it is restartable or not; otherwise, the controlling program is responsible for deciding whether a program qualifies as restartable.

Summary: loader calls categorized

The following table categorizes System Loader calls by the types of programs that make them. Most applications, whether their segments are static or dynamic, and whether or not they use run-time libraries, need make none of these calls. Applications that load dynamic segments manually may call any of the *user-callable* functions. Controlling programs and ProDOS 16 call the *system-wide* functions. Only the System Loader itself may call the *internal* functions. Functions not listed in Table 16-1 either do nothing or are executed only at system startup.

Table 16-1
System Loader functions categorized by caller

User-Callable	System-Wide	Internal
Loader Version	Initial Load	Jump Table Load
Loader Status	Restart	Cleanup
Load Segment By Number	Get User ID	
Unload Segment By Number	Get Pathname	
Load Segment By Name	User Shutdown	
Unload Segment		
Get Load Segment Info		

Chapter 17

System Loader Calls

Introduction

This chapter explains how System Loader functions are called, and describes the following calls:

Number	Function	Purpose
$01	Loader Initialization	(executed at system startup)
$02	Loader Startup	(no function)
$03	Loader Shutdown	(no function)
$04	Loader Version	returns System Loader version
$05	Loader Reset	(no function)
$06	Loader Status	returns initialization status
$09	Initial Load	loads an application
$0A	Restart	restarts a dormant application
$0B	Load Segment By Number	loads a single segment
$0C	Unload Segment By Number	unloads a single segment
$0D	Load Segment By Name	loads a single segment
$0E	Unload Segment	unloads a single segment
$0F	Get Load Segment Info	returns a segment's handle
$10	Get User ID	returns User ID for a pathname
$11	Get Pathname	returns pathname for a User ID
$12	User Shutdown	makes an application dormant
--	Jump Table Load	loads a dynamic segment
--	Cleanup	frees memory space

How calls are made

The System Loader is an Apple IIGS tool set (tool number 17, or hexadecimal $11). You call its functions using either macro calls (not described here) or the standard Apple IIGS tool calling sequence, as follows:

1. Push any required space for returned results onto the stack.
2. Push each input value onto the stack, in the proper order.
3. Execute the following call block:

```
LDX    #$11+FuncNum|8
JSL    Dispatcher
```

where

#$11 is the System Loader tool set number
FuncNum is the number of the function being called (**|8** means "shift left by 8 bits".)
Dispatcher is the address of the Tool Dispatcher ($E1 0000).

It is the responsibility of the caller (usually a controlling program) to prepare the stack for each function it calls, and to pull any results off the stack. Error status is returned in the accumulator (A register); furthermore, the carry bit is set (1) if the call is unsuccessful, and cleared (0) if the call is sucessful.

The Jump Table Load function does not use the above calling sequence, and cannot be called directly by an application. It is called indirectly, through a call to a Jump Table entry. The absolute address of the function is patched into the Jump Table by the System Loader at load time.

Parameter types

There are four types of parameters passed in the stack: values, results, pointers, and handles. Each is either an *input to* or an *output from* the loader function being called.

- A **value** is a numerical quantity, either 2 bytes (word; see Table 3-1) or 4 bytes (long word) in length, that the caller passes to the System Loader. It is an input parameter.
- A **result** is a numerical quantity, either 2 bytes (word) or 4 bytes (long word) in length, that the System Loader passes back to the caller. It is an output parameter.

- A **pointer** is the address of a location containing data, code, or buffer space in which the System Loader can receive or place data. A pointer may be 2 bytes (word) or 4 bytes (long word) in length. The pointer itself, and the data it points to, may be either input or output.
- A **handle** is a special type of pointer: it is a pointer to a pointer. It is the 4-byte address of a location that *itself* contains the address of a location containing data, code, or buffer space. In System Loader calls, a handle is always an output.

Format for System Loader call descriptions

The following sections describe the System Loader calls in detail. Each description contains these elements:

- the full name of the call
- a brief description of what function it performs
- the call's function number
- the call's assembly-language macro name (use it if you make macro calls)
- the call's parameter list (input and output)
- the stack configuration both before and after making the call
- a list of possible error codes
- the sequence of events the call invokes (if the brief description is not complete enough)

Parameter list note: In the parameter lists, *input* parameters are listed in the order in which they are pushed *onto* the stack; *output* parameters are listed in the order in which they are pulled *from* the stack. Check the stack diagrams if you are uncertain of the proper order in which to push any of the parameters.

Stack diagram note: Unlike other memory tables in this manual, the stack diagrams are organized in units of *words*—that is, each tick mark represents *two bytes* of stack space.

Loader Initialization ($01)

This routine initializes the System Loader; it is called by the system software at boot time. Loader Initialization clears all loader tables and sets the initial state of the system, making no assumptions about the current or previous state of the machine. The System Loader's global variables (see Appendix D) are defined at this time.

The Initialization routine is required for all Apple IIGS tool sets.

Function Number: $01

Macro Name: `LoaderInit`

Parameters

(none)

Possible errors

(none)

Loader Startup ($02)

The Startup routine is required for all Apple IIGS tool sets. For the System Loader, this function does nothing and need never be called.

Function Number: $02

Macro Name: `LoaderStartup`

Parameters

(none)

Possible errors

(none)

Loader Shutdown ($03)

The Shutdown routine is required for all Apple IIGS tool sets. For the System Loader, this function does nothing and need never be called.

Function Number: $03

Macro Name: `LoaderShutdown`

Parameters

(none)

Possible errors

(none)

Loader Version ($04)

The Loader Version function returns the version number of the System Loader currently in use. The version number has this format:

where

- Byte 0 is the minor release number (= 0 for System Loader version 1.0)
- Byte 1 is the major release number (= 1 for System Loader version 1.0)
- B (the most significant bit of byte 1) = 0 for final releases
  = 1 for all prototype releases

The Version routine is required for all Apple IIGS tool sets.

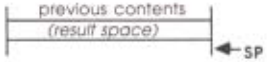
Function Number: $04

Macro Name: `LoaderVersion`

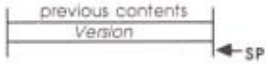
Parameters

	Name	Size and Type
Input	(none)	
Output	Loader version	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

(none)

Loader Reset ($05)

The Reset routine is required for all Apple IIGS tool sets. For the System Loader, this function does nothing and need never be called.

Function Number: $05

Macro Name: `LoaderReset`

Parameters

(none)

Possible errors

(none)

Loader Status ($06)

This routine returns the current status (initialized or uninitialized) of the System Loader. A nonzero result means TRUE (initialized); a zero result means FALSE (uninitialized). A result of TRUE is always returned by this call because the System Loader is always in the initialized state.

The Status routine is required for all Apple IIGS tool sets.

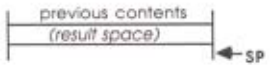
Function Number: $06

Macro Name: `LoaderStatus`

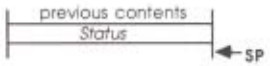
Parameters

	Name	Size and Type
Input	(none)	
Output	status	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

(none)

Initial Load ($09)

This function is called by a controlling program (such as a shell or a switcher) to ask the System Loader to perform an initial load of a program.

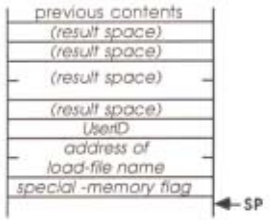
Function Number: $09

Macro Name: `InitialLoad`

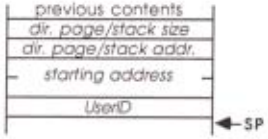
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
	address of load-file pathname	long word pointer (4 bytes)
	special-memory flag	word value (2 bytes)
Output	User ID	word result (2 bytes)
	starting address	long word pointer (4 bytes)
	address of direct-page/ stack buffer	word pointer (2 bytes)
	size of direct-page/ stack buffer	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1104	File is not a load file
$1105	System Loader is busy
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of events

When the Initial Load function is called, the following sequence of events occurs.

1. The function checks the `TypeID` and `MainID` fields of the specified User ID.
   a. If both fields are nonzero, the System Loader uses it to allocate space for the segments to be loaded.
   b. If the `TypeID` field is zero, the System Loader obtains a new User ID from the User ID Manager, to assign to all segments of that file. The new `TypeID` is given the value 1, meaning that the new file is classified as an application.
   c. If only the `MainID` field is zero, the System Loader obtains a new User ID from the User ID Manager, using the supplied `TypeID` and `AuxID`.

   The User ID Manager (described in *Apple IIGS Toolbox Reference*) guarantees that User ID's are unique to each application, tool, desk accessory, and so forth. See Appendix D of this manual for a brief description of the `User ID` format and the `TypeID` field.

2. The function checks the value of the special-memory flag. If it is TRUE (nonzero), the System Loader will not load any *static* segments into special memory (banks $00 and $01—see Chapter 3). The special-memory flag does not affect the load addresses of dynamic segments.
3. The function calls ProDOS 16 to open the specified (by pathname) load file. If any ProDOS 16 error occurs, or if the file is not a load file (type $B3-$BE), the System Loader returns the appropriate error code.

❖ *Note:* If the load file is a ProDOS 8 system file (type $FF) or a ProDOS 8 binary file (type $06), the loader will not load it.

4. Once the load file is opened, the System Loader adds the load-file information to the Pathname Table, and calls the Load Segment By Number function for each static segment in the load file.
   □ If any static segment loaded is an Initialization Segment (segment kind=$10), the System Loader immediately transfers control to it. When the System Loader regains control, it loads the rest of the static segments without passing control to them.
   □ If a direct-page/stack segment (`KIND`=$12) is loaded, the System Loader returns the segment's starting address and size.

❖ *Note:* The System Loader treats a direct-page/stack segment as a locked, unpurgeable, static segment. The segment cannot be moved or purged as long is the application is active, but it becomes purgeable at shutdown.

   □ If any of the static segments cannot be loaded, the System Loader aborts the load and returns the error from the Load Segment By Number function.

5. Once it has loaded all the static segments, the System Loader returns the starting address of the first segment (other than an initialization segment) of load file 1 to the controlling program. It then transfers execution to the controlling program. The controlling program itself is responsible for setting the stack and direct registers and for transferring control to the just-loaded program.

Restart ($0A)

This function is called by a controlling program (such as a shell or a switcher) to ask the System Loader to resurrect a dormant application—one that has been shut down (by the User Shutdown function), but is still in memory.

Only programs that are restartable can be successfully resurrected through this call. A restartable program always reinitializes its variables and makes no assumptions about machine state each time it executes.

To make it restartable, a program may include a **Reload segment** containing all necessary initialization information. A Reload segment is always loaded from the file at startup, even when a program is resarted.

❖ *Note:* The controlling program that makes the Restart call is responsible for making sure that the program it specifies is indeed restartable. The System Loader makes no such checks.

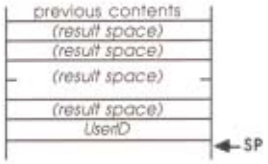
Function Number: $0A

Macro Name: Restart

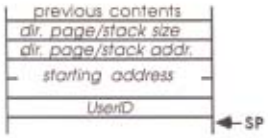
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
Output	User ID	word result (2 bytes)
	starting address	long word pointer (4 bytes)
	address of direct-page/ stack buffer	word pointer (2 bytes)
	size of direct-page/ stack buffer	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1101	Application not found
$1105	System Loader is busy
$1108	User ID error
$00xx	ProDOS 16 error
$02xx	Memory Manager error

Sequence of events

When the Restart function is called, the following sequence of events occurs.

1. An existing, nonzero User ID must be specified (the Aux ID part is ignored). If the User ID is zero, error $1108 is returned. If the User ID is unknown to the System Loader, error $1101 is returned.

2. The Restart function can work only if all of the specified program's static segments are still in memory. What that means is that no segments in the Memory Segment Table with the specified User ID can have been purged.
   a. The System Loader checks the memory handle of each Memory Segment Table entry with that User ID. If none are set to NIL the segments are all in memory.
   b. The System Loader then resurrects the application by calling the Memory Manager to make each of the application's segments unpurgeable and locked.
   c. The loader reloads any Reload segments it finds, and executes any initialization segments it finds.
   d. The loader returns the application's complete User ID, the first segment's starting address, and the direct page and stack information (from the Pathname Table) to the caller.
3. If any of the application's static segments are no longer in memory, the function does the following:
   a. It calls the Cleanup routine to purge all references to that User ID from the System Loader's tables and delete the User ID itself.
   b. It calls the Initial Load function to load the application. The application receives a new User ID, which is returned to the caller.

Load Segment By Number ($0B)

The Load Segment By Number routine is the workhorse function of the System Loader. Other System Loader functions that load segments do so by calling this function. It loads a specific load segment into memory; the segment is specified by its load-file number, load-segment number, and User ID.

❖ *Note:* Applications use this function to manually load dynamic segments. An application may also use Load Segment By Number to manually load a *static* segment. However, in that case the System Loader does not patch the correct address of the newly loaded segment onto any existing references to it. Therefore the segment can be accessed only through its starting address.

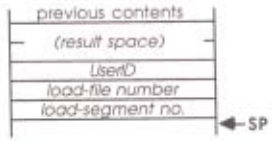
Function Number: $0B

Macro Name: `LoadSegNum`

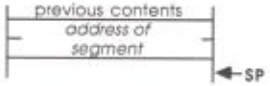
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
Output	address of segment	long word pointer (4 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1101	Segment not found
$1102	Incompatible OMF version
$1104	File is not a load file
$1105	System Loader is busy
$1107	File version error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of events

When the Load Segment By Number function is called, the following sequence of events occurs.

1. First the loader checks to find out if the requested load segment is already in memory: it searches the Memory Segment Table to determine if there is an entry for the segment. If the entry exists, the loader checks the value of the memory handle to find out whether the corresponding memory block is still in memory. If so, the function terminates without returning an error. If an entry exists but the memory block has been purged, the entry is deleted.
2. If the segment is not already in memory, the System Loader looks in the Pathname Table to get the load-file pathname from the load-file number.
3. The System Loader checks the file type of the referenced file. If it is not a load file (type $B3-$BE), then error $1104 is returned.

4. If the file is type $B4 (run-time library file), the System Loader compares the file's modification date and time values to the file date and file time in the Pathname Table. If they do not match, error $1107 is returned and the load is not performed.
5. ProDOS 16 is called to open the file. If ProDOS 16 cannot open the file, it returns an appropriate error code.
6. After ProDOS 16 successfully opens the load file, the System Loader searches the file for a load segment corresponding to the specified load-segment number. If none is found, error $1101 is returned.

   If the load segment is found, its header is checked (segment headers are described under "Object Module Format" in *Apple IIGS Programmer's Workshop Reference*). If the segment's OMF version number is incompatible with the current System Loader version, error $1102 is returned. If the value in the header's `SEGNUM` field does not match the specified load-segment number, error $1109 is returned. If the values in the `NUMSEX` and `NUMLEN` fields are not 0 and 4, respectively, error $110B is returned.
7. If the load segment is found and the header is correct, a memory block of the size specified in the `LENGTH` field of the segment header is requested from the Memory Manager. If the `ORG` field in the segment header is not zero, then a memory block starting at the address specified by `ORG` is requested (`ORG` is normally zero for Apple IIGS programming; that is, most segments are relocatable). Other segment attributes are set according to values in other segment header fields—see Chapter 14.
8. If a nonzero User ID is specified, the memory block is given that User ID. If the specified User ID is zero, the memory block is given the current User ID (value of `USERID` global variable).

9. If the requested memory is not available, the Memory Manager and System Loader use these techniques to free space:
   a. The Memory Manager unloads unneeded segments by purging their corresponding memory blocks. Blocks are purged according to their *purge levels*. For example, all level-3 blocks are purged before the first level-2 block is purged. Any dynamic segment whose memory block's purge level is zero cannot be unloaded.
   b. If all purgeable segments have been unloaded and the Memory Manager still cannot allocate enough memory, it moves any *movable* blocks to enlarge contiguous memory areas.
   c. If all eligible memory blocks have been purged or moved, and the Memory Manager still cannot allocate enough memory, the System Loader Cleanup routine is called to free any unused parts of the System Loader's memory. The Memory Manager then tries once more to allocate the requested memory.
   d. If the Memory Manager is still unsuccessful, the System Loader returns the last Memory Manager error that occurred.
10. Once the Memory Manager has allocated the requested memory, the System Loader puts the load segment into memory, and processes the relocation dictionary (if any).

❖ *Note:* If any records within the segment are not of a proper type ($E2, $E3, $F1, $F2, or $00), error $110A is returned. See Appendix D for an explanation of record types.

11. An entry for the segment is added to the Memory Segment Table.
12. The System Loader returns the starting address of the segment to the controlling program.

Unload Segment By Number ($0C)

This function unloads a specific load segment from memory. The segment is specified by its load-file number and load-segment number, and its User ID.

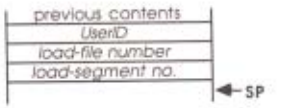
Function Number: $0C

Macro Name: `UnLoadSegNum`

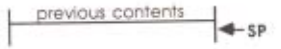
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
Output	(none)	

Stack Before Call:



Stack After Call:



Possible errors

$1101	Segment not found
$1105	System Loader is busy
$00xx	ProDOS 16 error
$02xx	Memory Manager error

Sequence of events

When the Unload Segment By Number function is called, the following sequence of events occurs.

1. The System Loader searches the Memory Segment Table for the specified load-file number and load-segment number. If there is no such entry, error $1101 is returned.
2. If the Memory Segment Table entry is found, the loader calls the Memory Manager to make *purgeable* (purge level = 3) the memory block in which the segment resides.
3. The loader changes all entries in the Jump Table that reference the unloaded segment to their unloaded states.

Special conditions:

- If the specified User ID is zero, the current User ID (value of USERID) is assumed.
- If both the load-file number and load-segment number are nonzero, the specified segment is unloaded regardless of whether it is static or dynamic. If either input is zero, only *dynamic* segments are unloaded, as noted next.
- If the specified load-file number is zero, all dynamic segments for that User ID are unloaded.
- If the specified load-segment number is zero, all dynamic segments for the specified load file are unloaded.

❖ *Note:* If a *static* segment is unloaded, the application that it is part of cannot be restarted from a dormant state. See "Restart" and "User Shutdown," in this chapter.

Load Segment By Name ($0D)

This function loads a *named* segment into memory. The segment is named by its load file's pathname, and its segment name (from the SEGNAME field in the segment header). A nonzero User ID may be specified if the loaded segment is to have a User ID different from the current User ID.

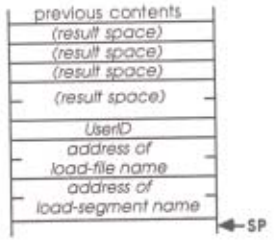
Function Number: $0D

Macro Name: LoadSegName

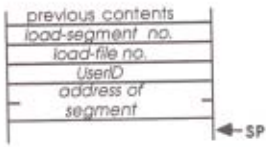
Parameters

	Name	Size and Type
Input:	User ID	word value (2 bytes)
	address of load-file name	long word pointer (4 bytes)
	address of load-segment name	long word pointer (4 bytes)
Output:	address of segment	long word pointer (4 bytes)
	User Id	word result (2 bytes)
	load-file number	word result (2 bytes)
	load-segment number	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1101	Segment not found
$1104	File is not a load file
$1105	System Loader is busy
$1107	File version error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign
$00xx	ProDOS 16 error
$02xx	Memory Manager error

Sequence of events

When the Load Segment By Name function is called, the following sequence of events occurs.

1. The System Loader gets the load-file pathname from the pointer given in the function call.
2. The System Loader checks the file type of the referenced file, from the file's disk directory entry. If it is not a load file (type $B3–$BE), error $1104 is returned.
3. If it is a load file, the loader calls ProDOS 16 to open the file. If ProDOS 16 cannot open the file, it returns the appropriate error code.
4. After the load file has been successfully opened by ProDOS 16, the System Loader searches the file for a segment with the specified name. If it finds none, error $1101 is returned.
5. If the load segment is found, the System Loader notes the segment number. It also checks the Pathname Table to see if the load file is listed. If the file is listed, the loader gets the load file number from the table; if not, it adds a new entry to the Pathname Table, assigning an unused file number to the load file. If the Jump-Table-loaded flag in the Pathname Table is FALSE, the loader loads the Jump Table segment (if any) from the load file and sets the Jump-Table-loaded flag to TRUE.
6. Now that it has both the load-file number and the segment number of the requested segment, the System Loader calls the Load Segment By Number function to load the segment. If the Load Segment By Number function returns an error, the Load Segment By Name function returns the same error. If the Load Segment By Number function is successful, the Load Segment By Name function returns the load file number, the load segment number, the User ID, and the starting address of the memory block in which the load segment was placed.

Unload Segment ($0E)

This function unloads the load segment containing the specified address. By using Unload Segment, an application can unload a segment without having to know its load-segment number, load-file number, name or User ID.

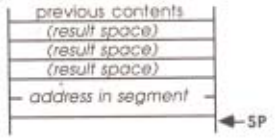
Function Number: $0E

Macro Name: UnloadSeg

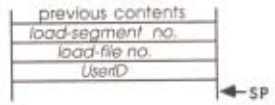
Parameters

	Name	Size and Type
Input	address in segment	long word pointer (4 bytes)
Output	User ID	word result (2 bytes)
	load-file number	word result (2 bytes)
	load-segment number	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1101	Segment not found
$1105	System Loader is busy
$00xx	ProDOS 16 error
$02xx	Memory Manager error

Sequence of events

When the Unload Segment function is called, the following sequence of events occurs.

1. The function calls the Memory Manager to identify the memory block containing the specified address. If the address is not within an allocated memory block, error $1101 is returned.
2. If the memory block is found, the function uses the memory handle returned by the Memory Manager to find the block's User ID. It then scans the Memory Segment Table for an entry with that User ID and handle. If no such entry is found, error $1101 is returned.
3. If the Memory Segment Table entry is found, the function does one of two things:
   a. If the Memory Segment Table entry refers to any segment other than a Jump Table segment, the function extracts the load-file number and load-segment number from the entry.
   b. If the Memory Segment Table entry refers to a Jump Table segment, the function extracts the load-file number and load-segment number in the *Jump Table entry* at the address specified in the function call.
4. The function then calls the Unload Segment By Number function to unload the segment.

The outputs of this function (load-file number, load-segment number, and User ID) can be used as inputs to other System Loader functions such as Load Segment By Number.

Get Load Segment Info ($0F)

This function returns the Memory Segment Table entry corresponding to the specified (by number) load segment.

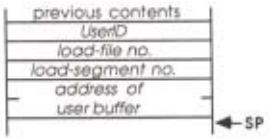
Function Number: $0F

Macro Name: `GetLoadSegInfo`

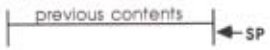
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
	address of user buffer	long word pointer (4 bytes)
Output	(filled user buffer)	

Stack Before Call:



Stack After Call:



Possible errors

$1101	Entry not found
$1105	System Loader is busy
$00xx	ProDOS 16 error
$02xx	Memory Manager error

Sequence of events

When the Get Load Segment Info function is called, the following sequence of events occurs.

1. The Memory Segment Table is searched for the specified entry. If the entry is not found, error $1101 is returned.
2. If the entry is found, the contents of the entry (except for the link pointers) are copied into the user buffer.

Get User ID ($10)

This function returns the User ID associated with the specified pathname. A controlling program can use this function to determine whether it can restart an application or must perform an initial load.

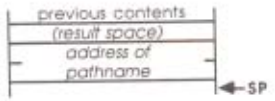
Function Number: $10

Macro Name: `GetUserID`

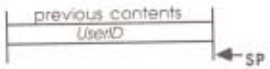
Parameters

	Name	Size and Type
Input	address of pathname	long word pointer (4 bytes)
Output	User ID	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1101	Entry not found
$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of events

When the Get User ID function is called, the following sequence of events occurs.

1. The System Loader searches the Pathname Table for the specified pathname. If the input pathname is a partial pathname and starts with a prefix number other than `1/` or `2/`, it is expanded to a full pathname before the search.
2. If it finds a match, the loader returns the User ID from that entry in the Pathname Table.

Get Pathname ($11)

This function returns the pathname associated with the specified User ID. ProDOS 16 uses this call to set the application prefix (1/) for a program that is restarted from memory.

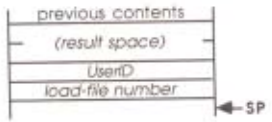
Function Number: $11

Macro Name: GetPathname

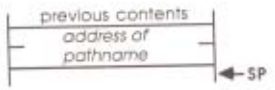
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
	File number	word value (2 bytes)
Output	Address of pathname	long word result (4 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1101	Entry not found
$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of events

When the Get Pathname function is called, the following sequence of events occurs.

1. The System Loader searches the Pathname Table for the specified User ID and file number.
2. If it finds a match, the loader returns the address of the pathname from that entry in the Pathname Table.

User Shutdown ($12)

This function is called by the controlling program to close down an application that has just terminated.

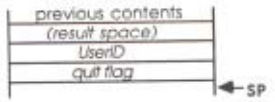
Function Number: $12

Macro Name: UserShutdown

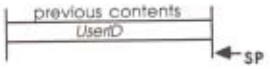
Parameters

	Name	Size and Type
Input:	User ID	word value (2 bytes)
	quit flag	word value (2 bytes)
Output:	User ID	word result (2 bytes)

Stack Before Call:



Stack After Call:



Possible errors

$1105	System Loader is busy
$00*xx*	ProDOS 16 error
$02*xx*	Memory Manager error

Sequence of events

❖ *Note:* This function is designed to support the options provided in the ProDOS 16 `QUIT` function. The quit flag in this call corresponds to the *flag word* parameter in the ProDOS 16 `QUIT` call. Only bits 14 and 15 of the flag are significant: If bit 15 is set, the quitting program wishes control to return to it eventually; if bit 14 is set, the program is restartable. See the description of the Restart function in this chapter.

When the User Shutdown function is called, the following sequence of events occurs.

1. The System Loader checks the specified User ID. If it is zero, the loader assumes it is the current User ID (= value of `USERID` global variable). In any case, loader ignores (by setting to zero) all values in the AuxID field of the User ID.
2. The loader checks the value of the quit flag.
   a. If the quit flag is zero, the Memory Manager *disposes* (permanently deallocates) all memory blocks with the specified User ID. The System Loader then calls its Cleanup routine to purge the loader's internal tables of all references to that User ID. The User ID itself is deleted so that the system no longer recognizes it.

      In this case the application is completely gone. It cannot be restarted from memory or quickly reloaded.

   b. If the quit flag is $8000 (bit 15 set to 1), the Memory Manager *purges* (temporarily deallocates) all memory blocks with the specified User ID. The System Loader's internal tables for that User ID, including the Pathname Table entry, remain intact.

      In this case the application can be reloaded quickly but it cannot be restarted from memory.

   c. If the quit flag has any other value, the Memory Manager

   - *disposes* all blocks corresponding to *dynamic* segments with the specified User ID
   - makes *purgeable* all blocks corresponding to *static* segments with that User ID
   - *purges* all other blocks with that User ID

      In addition, the System Loader removes all entries for that User ID from the Jump Table Directory.

The application is now in a dormant state—disconnected but not gone. It may be resurrected very quickly by the System Loader because all its static segments are still in memory. Once any of its static segments is purged by the Memory Manager, however, the program is truly lost and must be reloaded from disk if it is needed again.

Jump Table Load

This function is called by an *unloaded* Jump Table entry in order to load a dynamic load segment. Besides the function call, the unloaded Jump Table entry includes the load-file number and load-segment number of the dynamic segment to be loaded. The Jump Table is described in Chapter 15.

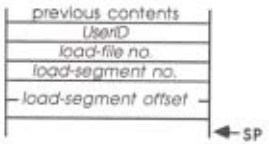
Function Number: none

Macro Name: none

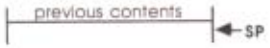
Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
	load-file number	word value (2 bytes)
	load-segment number	word value (2 bytes)
	load-segment offset	long word value (4 bytes)
Output	(none)	

Stack Before Call:



Stack After Call:



❖ *Note:* Because this function is never called directly by a controlling program, the program need not know what parameters it requires.

Possible errors

$1101	Segment not found
$1104	File is not a load file
$1105	System Loader is busy
$00xx	ProDOS 16 error
$02xx	Memory Manager error

Sequence of events

When the Jump Table Load function is called, the following sequence of events occurs.

1. The function calls the Load Segment By Number function, using the load-file number and load-segment number in the Jump Table entry. If the Load Segment By Number function returns any error, the System Loader considers it a fatal error and calls the System Failure Manager.
2. If the Load Segment By Number function successfully loads the segment, the Jump Table Load function changes the Jump Table entry to its *loaded* state: it replaces the JSL to the Jump Table Load function with a JML to the absolute address of the reference in the just-loaded segment.
3. The function transfers control to the address of the reference.

Cleanup

This routine is used to free additional memory when needed. It scans the System Loader's internal table and removes all entries that reference purged or disposed segments.

❖ *Note:* Because this function is never called directly by a controlling program, the program need not know what parameters it requires.

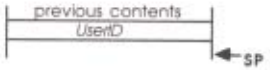
Function Number: none

Macro Name: none

Parameters

	Name	Size and Type
Input	User ID	word value (2 bytes)
Output	(none)	

Stack Before Call:



Stack After Call:

Possible errors

(none)

Sequence of events

When the Cleanup routine is called, the following sequence of events occurs.

1. If the specified User ID is 0:
   a. The System Loader scans all entries in the Memory Segment Table.
   b. All dynamic segments for all User ID's are purged.
2. If the specified User ID is nonzero:
   a. The System Loader scans all entries in the Memory Segment Table with that User ID.
   b. *All* load segments (both dynamic *and* static) for that User ID are purged.
   c. All entries in the Memory Segment Table, Jump Table directory, and Pathname Table for that User ID are deleted.

Appendixes

Appendix A

ProDOS 16 File Organization

This appendix describes in detail how ProDOS 16 stores files on disks. For most applications, the operating system insulates you from this level of detail. However, you must use this information if, for example, you want to

- list the files in a directory
- copy a sparse file without increasing the file's size
- compare two sparse files

Keep in mind that ProDOS 8 and ProDOS 16 have identical file structures. The information presented here applies equally to both systems.

This appendix first explains the organization of information on volumes. Next, it shows the format and organization of volume directories, subdirectories, and the various stages of standard files. Finally it presents a set of diagrams showing the formats of individual header and entry fields.

❖ *Note:* In this appendix, *format* refers to the arrangement of information (such as headers, pointers and data) within a file. *Organization* refers to the manner in which a single file is stored on disk, in terms of individual 512-byte blocks.

Organization of information on a volume

When a volume is formatted for use with ProDOS 16, its surface is partitioned into an array of tracks and sectors. In accessing a volume, ProDOS 16 requests not a track and sector, but a logical block from the device corresponding to that volume. That device's driver translates the requested block number into the proper track and sector number; the physical location of information on a volume is unimportant to ProDOS 16 and to an application that uses ProDOS 16. This appendix discusses the organization of information on a volume in terms of logical blocks, not tracks and sectors.

When the volume is formatted, information needed by ProDOS 16 is placed in specific logical blocks, starting with the first block (block 0). A loader program is placed in blocks 0 and 1 of the volume. This program enables ProDOS 16 (or ProDOS 8) to be booted from the volume. Block 2 of the volume is the **key block** (the first block) of the volume directory file; it contains descriptions of (and pointers to) all the files in the volume directory. The *volume directory* occupies a number of consecutive blocks, typically four, and is immediately followed by the **volume bit map**, which records whether each block on the volume is used or unused. The volume bit map occupies consecutive blocks, one for every 4,096 blocks, or fraction thereof, on the volume. The rest of the blocks on the disk contain *subdirectory file* information, *standard file* information, or are empty. The first blocks of a volume look something like Figure A-1.

Figure A-1
Block organization of a volume

The precise format of the volume directory, volume bit map, subdirectory files and standard files are explained in the following sections.

Format and organization of directory files

The format and organization of the information contained in volume directory and subdirectory files is quite similar. Each consists of a key block followed by zero or more blocks of additional directory information. The fields in a directory's key block are:

- a pointer to the next block in the directory
- a header that describes the directory
- a number of file entries describing, and pointing to, the files in that directory
- zero or more unused bytes

The fields in subsequent (nonkey) blocks in a directory are:

- pointers to the preceding and succeeding blocks in the directory
- a number of entries describing, and pointing to, the files in that directory
- zero or more unused bytes

The format of a directory file is represented in Figure A-2.

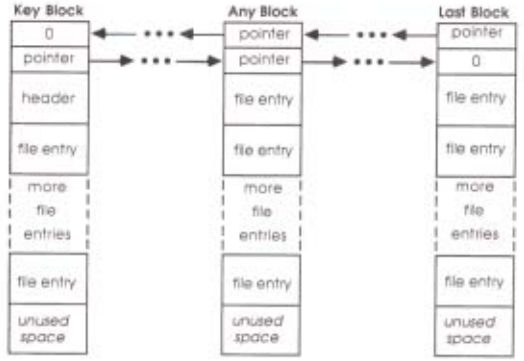


Figure A-2
Directory file format and organization

The header is the same length as all other entries in a directory file. The only difference between a volume directory file and a subdirectory file is in the header format.

Pointer fields

The first four bytes of each block used by a directory file contain pointers to the preceding and succeeding blocks in the directory file, respectively. Each pointer is a two-byte logical block number—low-order byte first, high-order byte second. The key block of a directory file has no preceding block; its first pointer is zero. Likewise, the last block in a directory file has no successor; its second pointer is zero.

❖ *Note:* The block pointers described in this appendix, which hold *disk* addresses, are two bytes long. All other ProDOS 16 pointers, which hold *memory* addresses, are four bytes long. In either case, ProDOS 16 pointers are always stored with the low-order byte first and the high-order byte last. See Chapter 3, "ProDOS 16 and Apple IIGS Memory."

Volume directory headers

Block 2 of a volume is the key block of that volume's directory file. The volume directory header is at byte position $0004 of the key block, immediately following the block's two pointers. Thirteen fields are currently defined to be in a volume directory header: they contain all the vital information about that volume. Figure A-3 illustrates the format of a volume directory header. Following Figure A-3 is a description of each of its fields.

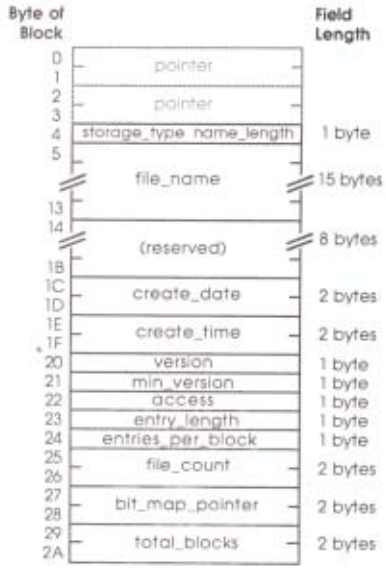


Figure A-3
The volume directory header

storage_type and **name_length** (1 byte): Two four-bit (nibble) fields are packed into this byte. A value of $F in the high-order nibble (`storage_type`) identifies the current block as the key block of a volume directory file. The low-order nibble contains the length of the volume's name (see the `file_name` field, below). The value of `name_length` can be changed by a `CHANGE_PATH` call.

file_name (15 bytes): The first *n* bytes of this field, where *n* is the value of `name_length`, contain the volume's name. This name must conform to the file name (volume name) syntax explained in Chapter 2. The name does not begin with the slash that usually precedes volume names. This field can be changed by the `CHANGE_PATH` call.

reserved (8 bytes): Reserved for future expansion of the file system.

create_date (2 bytes): The date on which this volume was initialized. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

create_time (2 bytes): The time at which this volume was initialized. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

version (1 byte): The file system version number of ProDOS 8 or ProDOS 16 under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS 16 to determine the format of the file, and adjust their interpretation processes accordingly. For ProDOS 16, `version` = 0.

❖ *Note:* Version in this sense refers to the *file system* version only. At present, all ProDOS operating systems use the same file system and therefore have the same file system version number (0). In particular, the file system version number is unrelated to the *program* version number returned by the `GET_VERSION` call.

min_version: Reserved for future use. For ProDOS 16, it is 0.

access (1 byte): Determines whether this volume directory can be read, written, destroyed, or renamed. The format of this field is described under "Header and Entry Fields," in this appendix.

entry_length (1 byte): The length in bytes of each entry in this directory. The volume directory header itself is of this length. For ProDOS 16, `entry_length` = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. For ProDOS 16, `entries_per_block` = $0D.

file_count (2 bytes): The number of active file entries in this directory file. An active file is one whose `storage_type` is not 0. Figure A-5 shows the format of file entries.

bit_map_pointer (2 bytes): The block address of the first block of the volume's bit map. The bit map occupies consecutive blocks, one for every 4,096 blocks (or fraction thereof) on the volume. You can calculate the number of blocks in the bit map using the `total_blocks` field, described below.

The bit map has one bit for each block on the volume: a value of 1 means the block is free; 0 means it is in use. If the number of blocks used by all files on the volume is not the same as the number recorded in the bit map, the directory structure of the volume has been damaged.

total_blocks (2 bytes): The total number of blocks on the volume.

Subdirectory headers

The key block of every subdirectory file is pointed to by an entry in a parent directory; for example, by an entry in a volume directory (Figure A-2). A subdirectory's header begins at byte position $0004 of the key block of that subdirectory file, immediately following the two pointers.

In format, a subdirectory header is quite similar to a volume directory header (only its last three fields are different). A subdirectory header has fourteen fields; those fields contain all the vital information about that subdirectory. Figure A-4 illustrates the format of a subdirectory header. A description of all the fields in the header follows the figure.

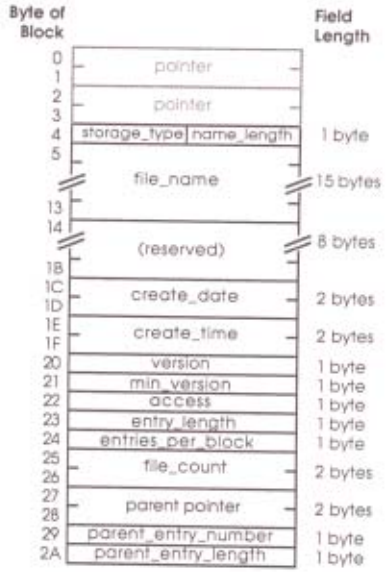


Figure A-4
The subdirectory header

storage_type and **name_length** (1 byte): Two four-bit (nibble) fields are packed into this byte. A value of $E in the high-order nibble (`storage_type`) identifies the current block as the key block of a subdirectory file. The low-order nibble contains the length of the subdirectory's name (see the `file_name` field, below). The value of name_length can be changed by a `CHANGE_PATH` call.

file_name (15 bytes): The first *name_length* bytes of this field contain the subdirectory's name. This name must conform to the file name syntax explained in Chapter 2. This field can be changed by the `CHANGE_PATH` call.

reserved (8 bytes): Reserved for future expansion of the file system.

create_date (2 bytes): The date on which this subdirectory was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

create_time (2 bytes): The time at which this subdirectory was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

version (1 byte): The file system version number of ProDOS 8 or ProDOS 16 under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS 16 to determine the format of the file, and adjust their interpretation processes accordingly. For ProDOS 16, `version` = 0.

❖ *Note:* Version in this sense refers to the *file system* version only. At present, all ProDOS operating systems use the same file system and therefore have the same file system version number (0). In particular, the file system version number is unrelated to the *program* version number returned by the `GET_VERSION` call.

min_version (1 byte): The minimum version number of ProDOS 8 or ProDOS 16 that can access the information in this file. This byte allows older versions of ProDOS 8 and ProDOS 16 to determine whether they can access newer files. For ProDOS 16, `min_version` = 0.

access (1 byte): Determines whether this subdirectory can be read, written, destroyed, or renamed, and whether the file needs to be backed up. The format of this field is described under "Header and Entry Fields," in this appendix. A subdirectory's access byte can be changed by the `SET_FILE_INFO` and `CLEAR_BACKUP_BIT` calls.

entry_length (1 byte): The length in bytes of each entry in this subdirectory. The subdirectory header itself is of this length. For ProDOS 16, `entry_length` = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. For ProDOS 16, `entries_per_block` = $0D.

file_count (2 bytes): The number of active file entries in this subdirectory file. An active file is one whose `storage_type` is not 0. See "File Entries" for more information about file entries.

parent_pointer (2 bytes): The block address of the directory file block that contains the entry for this subdirectory. This and all other two-byte pointers are stored low-order byte first, high-order byte, second.

parent_entry_number (1 byte): The entry number for this subdirectory within the block indicated by `parent_pointer`.

parent_entry_length (1 byte): The entry_length for the directory that owns this subdirectory file. Note that with these last three fields you can calculate the precise position on a volume of this subdirectory's file entry. For ProDOS 16, `parent_entry_length` = $27.

File entries

Immediately following the pointers in any block of a directory file are a number of entries. The first entry in the key block of a directory file is a header, all other entries are file entries. Each entry has the length specified by that directory's `entry_length` field, and each file entry contains information that describes, and points to, a single subdirectory file or standard file.

An entry in a directory file may be active or inactive, that is, it may or may not describe a file currently in the directory. If it is inactive, the first byte of the entry (`storage_type` and `name_length`) has the value zero.

The maximum number of entries, including the header, in a block of a directory is recorded in the `entries_per_block` field of that directory's header. The total number of active file entries, not including the header, is recorded in the `file_count` field of that directory's header.

Figure A-5 describes the format of a file entry.

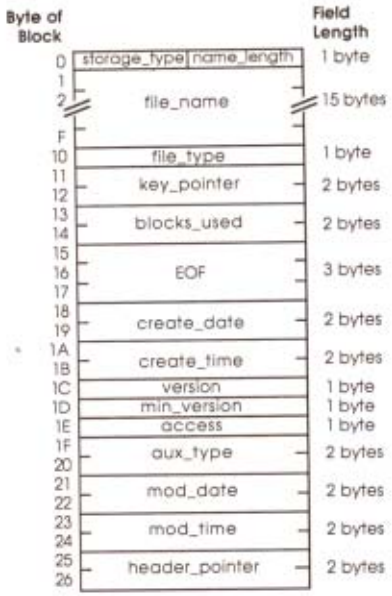


Figure A-5
The file entry

storage_type and **name_length** (1 byte): Two four-bit (nibble) fields are packed into this byte. The value in the high-order nibble (`storage_type`) specifies the type of file pointed to by this file entry:

$1 = Seedling file
$2 = Sapling file
$3 = Tree file
$4 = Pascal area
$D = Subdirectory

Seedling, sapling, and tree files are described under "Format and Organization of Standard Files," in this appendix. The low-order nibble contains the length of the file's name (see the `file_name` field, below). The value of `name_length` can be changed by a `CHANGE_PATH` call.

file_name (15 bytes): The first *name_length* bytes of this field contain the file's name. This name must conform to the file name syntax explained in Chapter 2. This field can be changed by the `CHANGE_PATH` call.

file_type (1 byte): A descriptor of the internal format of the file. Table A-1 (at the end of this appendix) is a list of the currently defined values of this byte.

key_pointer (2 bytes): The block address of:

- the master index block (if the file is a tree file)
- the index block (if the file is a sapling file)
- the data block (if the file is a seedling file)

blocks_used (2 bytes): The total number of blocks actually used by the file. For a subdirectory file, this includes the blocks containing subdirectory information, but not the blocks in the files pointed to. For a standard file, this includes both informational blocks (index blocks) and data blocks. See "Format and Organization of Standard Files" in this appendix.

EOF (3 bytes): A three-byte integer, lowest byte first, that represents the total number of bytes readable from the file. Note that in the case of sparse files, EOF may be greater than the number of bytes actually allocated on the disk.

create_date (2 bytes): The date on which the file pointed to by this entry was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

create_time (2 bytes): The time at which the file pointed to by this entry was created. The format of these bytes is described under "Header and Entry Fields," later in this appendix.

version (1 byte): The file system version number of ProDOS 8 or ProDOS 16 under which the file pointed to by this entry was created. This byte allows newer versions of ProDOS 16 to determine the format of the file, and adjust their interpretation processes accordingly. For ProDOS 16, `version` = 0.

❖ *Note:* Version in this sense refers to the *file system* version only. At present, all ProDOS operating systems use the same file system and therefore have the same file system version number. The file system version number is unrelated to the *program* version number returned by the `GET_VERSION` call.

min_version (1 byte): The minimum version number of ProDOS 8 or ProDOS 16 that can access the information in this file. This byte allows older versions of ProDOS 8 and ProDOS 16 to determine whether they can access newer files. For ProDOS 16, `min_version` = 0.

access (1 byte): Determines whether this file can be read, written, destroyed, or renamed, and whether the file needs to be backed up. The format of this field is described under "Header and Entry Fields," later in this appendix. The value of this field can be changed by the `SET_FILE_INFO` and `CLEAR_BACKUP_BIT` calls. You cannot delete (destroy) a subdirectory that contains any files.

aux_type (2 bytes): A general-purpose field in which an application can store additional information about the internal format of a file. For example, the ProDOS 8 BASIC system program uses this field to record the load address of a BASIC program or binary file, or the record length of a text file.

mod_date (2 bytes): The date on which the last `CLOSE` operation after a `WRITE` was performed on this file. The format of these bytes is described under "Header and Entry Fields," later in this appendix. This field can be changed by the `SET_FILE_INFO` call.

mod_time (2 bytes): The time at which the last `CLOSE` operation after a `WRITE` was performed on this file. The format of these bytes is described under "Header and Entry Fields," later in this appendix. This field can be changed by the `SET_FILE_INFO` call.

header_pointer (2 bytes): This field is the block address of the key block of the directory that owns this file entry. This and all two-byte pointers are stored low-order byte first, high-order byte second.

Reading a directory file

This section deals with the general techniques of reading from directory files, not with the specifics. The ProDOS 16 calls with which these techniques can be implemented are explained in Chapters 9 and 10.

Before you can read from a directory, you must know the directory's pathname. With the directory's pathname, you can open the directory file, and obtain a reference number (*ref_num*) for that open file. Before you can process the entries in the directory, you must read three values from the directory header:

- length of each entry in the directory (*entry_length*)
- number of entries in each block of the directory (*entries_per_block*)
- total number of files in the directory (*file_count*).

Using the reference number to identify the file, read the first 512 bytes from the file, and into a buffer (`ThisBlock` in the following example). The buffer contains two two-byte pointers, followed by the entries; the first entry is the directory header. The three values are at positions $1F through $22 in the header (positions $23 through $26 in the buffer). In this example, these values are assigned to the variables `EntryLength`, `EntriesPerBlock`, and `FileCount`.

```
Open(DirPathname, RefNum);                                  {Get reference number}
ThisBlock                := Read512Bytes(RefNum);          {Read a block into buffer}
EntryLength              := ThisBlock[$23];                  {Get directory info}
EntriesPerBlock          := ThisBlock[$24];
FileCount                := ThisBlock[$25] + (256 * ThisBlock[$26]);
```

Once these values are known, an application can scan through the entries in the buffer, using a pointer (`EntryPointer`) to the beginning of the current entry, a counter (`BlockEntries`) that indicates the number of entries that have been examined in the current block, and a second counter (`ActiveEntries`) that indicates the number of active entries that have been processed.

An entry is active and is processed only if its first byte, the `storage_type and name_length`, is nonzero. All entries have been processed when *ActiveEntries* is equal to *FileCount*. If all the entries in the buffer have been processed, and *ActiveEntries* doesn't equal *FileCount*, then the next block of the directory is read into the buffer.

```
EntryPointer                    := EntryLength + $04;          {Skip header entry}
BlockEntries                    := $02;                {Prepare to process entry two}
ActiveEntries                   := $00;                {No active entries found yet }

while ActiveEntries < FileCount do begin
    if ThisBlock[EntryPointer] <> $00 then begin                        {Active entry}
            ProcessEntry(ThisBlock[EntryPointer]);
            ActiveEntries := ActiveEntries + $01
    end;
    if ActiveEntries < FileCount then                      {More entries to process}
            if BlockEntries = EntriesPerBlock
                    then begin                          {ThisBlock done. Do next one}
                    ThisBlock       := Read512Bytes(RefNum);
                    BlockEntries    := $01;
                    EntryPointer    := $04
            end
            else begin                                    {Do next entry in ThisBlock}
                    EntryPointer    := EntryPointer + EntryLength;
                    BlockEntries    := BlockEntries + $01
            end
end;
Close(RefNum);
```

This algorithm processes entries until all expected active entries have been found. If the directory structure is damaged, and the end of the directory file is reached before the proper number of active entries has been found, the algorithm fails.

Format and organization of standard files

Each active entry in a directory file points to the key block (the first block) of another file, which itself is either a subdirectory file or a *standard file*. As shown below, the key block of a standard file may have several types of information in it. The `storage_type` field in that file's entry must be used to determine the contents of the key block. This section explains the organization of the three stages of standard file: seedling, sapling, and tree. These are the files in which all programs and data are stored.

Every block in a standard file is either a data block or an index block. Data blocks have no predefined format—they contain whatever information the file was created to hold. Index blocks, on the other hand, have a very specific format—they consist of nothing but 2-byte pointers, giving the (disk) adresses of other blocks that make up the file. Furthermore, the low-order byte of each pointer is in the first half of the block, whereas the high-order byte of the pointer is in the second half of the block. An index block can have up to 256 pointers, so if a pointer's low-order byte is at address *n* in the block, its high-order byte is at address *n*+256.

❖ *Note:* Deleting a file or changing its logical size (EOF) can alter the contents of its index blocks. See "DESTROY" in Chapter 9 and "SET_EOF" in Chapter 10.

Growing a tree file

The following scenario demonstrates the *growth* of a tree file on a volume. This scenario is based on the block allocation scheme used by ProDOS 16 on a 280-block flexible disk that contains four blocks of volume directory, and one block of volume bit map. Larger capacity volumes might have more blocks in the volume bit map, but the process would be identical.

A formatted, but otherwise empty, ProDOS 16 volume is used like this:

Blocks 0-1	Loader
Blocks 2-5	Volume directory
Block 6	Volume bit map
Blocks 7-279	Unused

If you open a new file of a nondirectory type, one data block is immediately allocated to that file. An entry is placed in the volume directory, and it points to block 7, the new data block, as the key block for the file. The key block is indicated below by an arrow.

The volume now looks like this:

	Blocks 0-1	Loader
	Blocks 2-5	Volume directory
	Block 6	Volume bit map
—>	**Block 7**	**Data block 0**
	Blocks 8-279	Unused

This is a **seedling file**: its key block contains up to 512 bytes of data. If you write more than 512 bytes of data to the file, the file *grows* into a **sapling file**. As soon as a second block of data becomes necessary, an index block is allocated, and it becomes the file's key block: this index block can point to up to 256 data blocks (it uses two-byte pointers). A second data block (for the data that won't fit in the first data block) is also allocated.

The volume now looks like this:

	Blocks 0-1	Loader
	Blocks 2-5	Volume directory
	Block 6	Volume bit map
	Block 7	**Data block 0**
—>	**Block 8**	**Index block 0**
	Block 9	**Data block 1**
	Blocks 10-279	Unused

This sapling file can hold up to 256 data blocks: 128K of data. If the file becomes any bigger than this, the file *grows* again, this time into a **tree file**. A master index block is allocated, and it becomes the file's key block: the master index block can point to up to 128 index blocks, and each of these can point to up to 256 data blocks. Index block 0 becomes the first index block pointed to by the master index block. In addition, a new index block is allocated, and a new data block to which it points.

Here's a new picture of the volume:

	Blocks 0-1	Loader
	Blocks 2-5	Volume directory
	Block 6	Volume bit map
	Block 7	**Data block 0**
	Block 8	**Index block 0**
	Blocks 9-263	**Data blocks 1-255**
—>	**Block 264**	**Master index block**
	Block 265	**Index block 1**
	Block 266	**Data block 256**
	Blocks 267-279	Unused

As data is written to this file, additional data blocks and index blocks are allocated as needed, up to a maximum of 129 index blocks (one a master index block), and 32,768 data blocks, for a maximum capacity of 16,777,215 bytes of data in a file. If you did the multiplication, you probably noticed that a byte was lost somewhere. The last byte of the last block of the largest possible file cannot be used because EOF cannot exceed 16,777,216. If you are wondering how such a large file might fit on a small volume such as a flexible disk, refer to the description of sparse files in this appendix.

This scenario shows the growth of a single file on an otherwise empty volume. The process is a bit more confusing when several files are growing—or being deleted—simultaneously. However, the block allocation scheme is always the same: when a new block is needed, ProDOS 16 always allocates the first unused block in the volume bit map.

Seedling files

A seedling file is a standard file that contains no more than 512 data bytes ($0 <= EOF <= $200). This file is stored as one block on the volume, and this data block is the file's key block.

The organization of such a seedling file appears in Figure A-6.

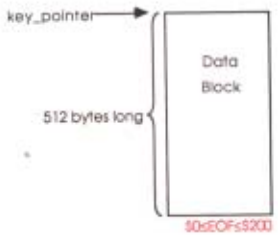


Figure A-6
Format and organization of a seedling file

The file is called a seedling file because it is the smallest possible ProDOS 16 standard file; if more than 512 data bytes are written to it, it grows into a sapling file, and thence into a tree file.

The `storage_type` field of a directory entry that points to a seedling file has the value $1.

Sapling files

A sapling file is a standard file that contains more than 512 and no more than 128K bytes ($200 < EOF <= $20000). A sapling file comprises an index block and 1 to 256 data blocks. The index block contains the block addresses of the data blocks. Figure A-7 shows the organization.

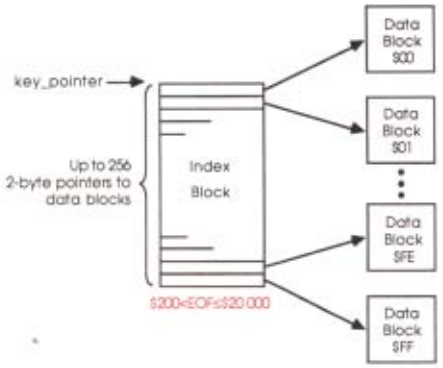


Figure A-7
Format and organization of a sapling file

The key block of a sapling file is its index block. ProDOS 16 retrieves data blocks in the file by first retrieving their addresses in the index block.

The `storage_type` field of a directory entry that points to a sapling file has the value $2.

Tree files

A tree file contains more than 128K bytes, and less than 16Mb ($20000 < EOF < $1000000). A tree file consists of a master index block, 1 to 128 index blocks, and 1 to 32,768 data blocks. The master index block contains the addresses of the index blocks, and each index block contains the addresses of up to 256 data blocks. The organization of a tree file is shown in Figure A-8.

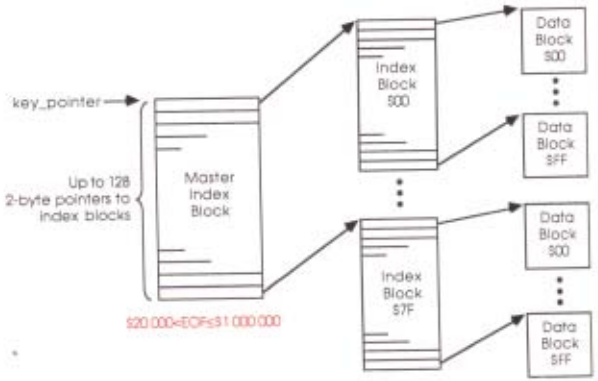


Figure A-8
Format and organization of a tree file

The key block of a tree file is the master index block. By looking at the master index block, ProDOS 16 can find the addresses of all the index blocks; by looking at those blocks, it can find the addresses of all the data blocks.

The `storage_type` field of a directory entry that points to a tree file has the value $3.

Using standard files

An application program operates the same on all three types of standard files, although the `storage_type` in the file's entry can be used to distinguish between the three. A program rarely reads index blocks or allocates blocks on a volume: ProDOS 16 does that. The program need only be concerned with the data stored in the file, not with how they are stored.

All types of standard files are read as a sequence of bytes, numbered from 0 to (EOF–1), as explained in Chapter 2.

Sparse files

A sparse file is a sapling or tree file in which the number of data bytes that can be read from the file exceeds the number of bytes physically stored in the data blocks allocated to the file. ProDOS 16 implements sparse files by allocating only those data blocks that have had data written to them, as well as the index blocks needed to point to them.

For example, you can define a file whose EOF is 16K, that uses only three blocks on the volume, and that has only four bytes of data written to it. Refer to figure A-9 during the following explanation.

1. If you create a file with an EOF of $0, ProDOS 16 allocates only the key block (a data block) for a seedling file, and fills it with null characters (ASCII $00).
2. If you then set the EOF and Mark to position $0565, and write four bytes, ProDOS 16 calculates that position $0565 is byte $0165 ($0564–($0200 * 2)) of the third block (block $2) of the file. It then allocates an index block, stores the address of the current data block in position 0 of the index block, allocates another data block, stores the address of that data block in position 2 of the index block, and stores the data in bytes $0165 through $0168 of that data block. The EOF is now $0569.
3. If you now set the EOF to $4000 and close the file, you have a 16K sapling file that takes up three blocks of space on the volume: two data blocks and an index block (shaded in Figure A-9). You can read 16384 bytes of data from the file, but all the bytes before $0565 and after $0568 are nulls.

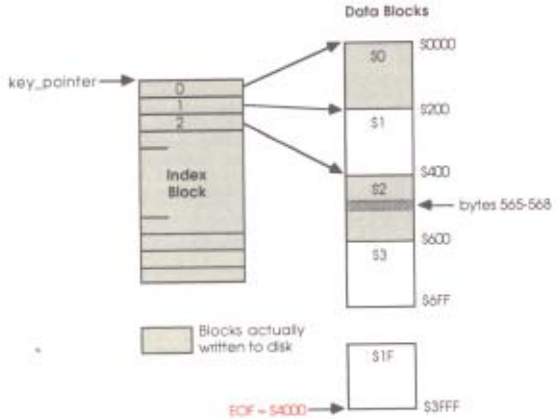


Figure A-9
An example of sparse file organization

Thus ProDOS 16 allocates volume space only for those blocks in a file that actually contain data. For tree files, the situation is similar: if none of the 256 data blocks assigned to an index block in a tree file have been allocated, the index block itself is not allocated.

❖ *Note:* The first data block of a standard file, be it a seedling, sapling, or tree file, is always allocated. Thus there is always a data block to be read in when the file is opened.

Locating a byte in a file

This is how to find a specific byte within a standard file:

The File Mark is a three-byte value that indicates an absolute byte position within a file. If the file is a tree file, then the high-order seven bits of the Mark determine the number (0 to 127) of the index block that points to the byte. That number is also the location of the low byte of the index block address within the master index block. The location of the high byte of the index block address is that number plus 256.

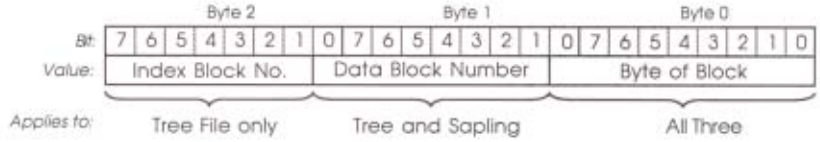


Figure A-10
File Mark format

If the file is a tree file or a sapling file, then the next eight bits of the Mark determine the number (0-255) of the data block pointed to by the indicated index block. That number is also the location of the low byte of the data block address within the index block. The high byte of the index block address is found at that value plus 256.

For tree, sapling, and seedling files, the value of the low nine bits of the Mark is the location of the byte within the selected data block.

Header and entry fields

The storage type attribute

The value in the `storage_type` field, the high-order four bits of the first byte of an entry, defines the type of header (if the entry is a header) or the type of file described by the entry. Table A-1 lists the currently defined storage type values.

Table A-1
Storage type values

	Storage type
$0	indicates an inactive file entry
$1	indicates a seedling file entry (EOF <= 256 bytes)
$2	indicates a sapling file entry (256 < EOF <= 128K bytes)
$3	indicates a tree file entry (128K < EOF < 16M bytes)
$4	indicates a Pascal operating system area on a partitioned disk
$D	indicates a subdirectory file entry
$E	indicates a subdirectory header
$F	indicates a volume directory header

ProDOS 16 automatically changes a seedling file to a sapling file and a sapling file to a tree file when the file's EOF grows into the range for a larger type. If a file's EOF shrinks into the range for a smaller type, ProDOS 16 changes a tree file to a sapling file and a sapling file to a seedling file.

The creation and last-modification fields

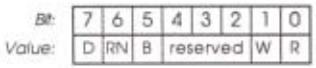
The date and time of the creation and last modification of each file and directory is stored as two four-byte values, as shown in Figure A-11.

Figure A-11
Date and time format

The values for the year, month, day, hour, and minute are stored as binary integers, and may be unpacked for conversion to normal integer values.

The access attribute

The access attribute field, or access byte (Figure A-12), determines whether the file can be read from, written to, deleted, or renamed. It also contains a bit that can be used to indicate whether a backup copy of the file has been made since the file's last modification.



where

D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

Figure A-12
Access byte format

A bit set to 1 indicates that the operation is enabled; a bit cleared to 0 indicates that the operation is disabled. The reserved bits are always 0. The most typical setting for the access byte is $C3 (11000011).

ProDOS 16 sets bit 5, the **backup bit,** to 1 whenever the file is changed (that is, after a `CREATE`, `RENAME`, `CLOSE` after `WRITE`, or `SET_FILE_INFO` operation). This bit should be reset to 0 whenever the file is duplicated by a backup program.

❖ *Note:* Only ProDOS 16 may change bits 2-4; only backup programs should clear bit 5 (using `CLEAR_BACKUP_BIT`).

The file type attribute

The `file_type` field in a directory entry identifies the type of file described by that entry. This field should be used by applications to guarantee file compatibility from one application to the next. The currently defined hexadecimal values of this byte are listed in Table A-2.

Table A-2 also lists the 3-character mnemonic file-type codes that should appear on catalog listings. For any file type without a specified mnemonic code, the catalog program should substitute the hexadecimal file type number.

❖ *Note:* SOS file types are included in Table A-2 because SOS and ProDOS have identical file systems.

Table A-2
ProDOS file types

File type	Mnemonic Code	Description
$00		Uncategorized file (SOS and ProDOS)
$01	BAD	Bad block file
$02 †	PCD	Pascal code file
$03 †	PTX	Pascal text file
$04	TXT	ASCII text file (SOS and ProDOS)
$05 †	PDA	Pascal data file
$06	BIN	General binary file (SOS and ProDOS 8)
$07 †	FNT	Font file
$08	FOT	Graphics screen file
$09 †	BA3	Business BASIC program file
$0A †	DA3	Business BASIC data file
$0B †	WPF	Word Processor file
$0C †	SOS	SOS system file
$0D–$0E †		(SOS reserved)
$0F	DIR	Directory file (SOS and ProDOS)
$10 †	RPD	RPS data file
$11 †	RPI	RPS index file
$12 †		AppleFile discard file
$13 †		AppleFile model file
$14 †		AppleFile report format file
$15 †		Screen Library file
$16–$18 †		(SOS reserved)

Table A-2 (continued)
ProDOS file types

File type	Mnemonic Code	Description
$19	ADB	AppleWorks Data Base file
$1A	AWP	AppleWorks Word Proc. file
$1B	ASP	AppleWorks Spreadsheet file
$1C–$AF		(reserved)
$B0	SRC	APW source file
$B1	OBJ	APW object file
$B2	LIB	APW library file
$B3	S16	ProDOS 16 application program file
$B4	RTL	APW run-time library file
$B5	EXE	ProDOS 16 shell application file
$B6		ProDOS 16 permanent initialization file
$B7		ProDOS 16 temporary initialization file
$B8		New desk accessory
$B9		Classic desk accessory
$BA		Tool set file
$BB–$BE		(reserved for ProDOS 16 load files)
$BF		ProDOS 16 document file
$C0–$EE		(reserved)
$EF	PAS	Pascal area on a partitioned disk
$F0	CMD	ProDOS 8 CI added command file
$F1–$F8		ProDOS 8 user defined files 1-8
$F9		(ProDOS 8 reserved)
$FA	INT	Integer BASIC program file
$FB	IVR	Integer BASIC variable file
$FC	BAS	Applesoft program file
$FD	VAR	Applesoft variables file
$FE	REL	Relocatable code file (EDASM)
$FF	SYS	ProDOS 8 system program file

†apply to SOS (Apple III) only

The auxiliary type attribute

Some applications use an another field in a file's directory entry, the auxiliary type field (`aux_type`), to store additional information not specified by the file type. Catalog listings may display the contents of this field under the heading "Subtype."

For example, APW source files (`file type $B0`) include a language-type designation in the `aux_type` field. The starting address for ProDOS 8 executable binary files (`file type $06`) may be in the aux_type field. The record size for random-access text files (`file type $04`) may be specified in the auxiliary type field.

ProDOS 16 and ProDOS 8 impose no restrictions (other than size) on the contents or format of the auxiliary type field. Individual applications may use those 2 bytes to store any useful information.

Appendix B

Apple II Operating Systems

This appendix explains the relationships between ProDOS 16 and three other operating systems developed for the Apple II family of computers (DOS, ProDOS 8, and Apple II Pascal), as well as two developed for the Apple III (SOS and Apple III Pascal).

If you have written programs for one of the other systems or are planning to write programs concurrently for ProDOS 16 and another system, this appendix may help you see what changes will be necessary to transfer your program from one system to another. If you are converting files from one system to another, this appendix may help you understand why some conversions may be more successful than others.

The first section gives a brief history. The next two sections give general comparisons of the other operating systems to ProDOS 16, in terms of file compatibility and operational similarity.

History

DOS

DOS stands for *Disk Operating System*. It is Apple's first operating system; before DOS, the firmware **Monitor program** controlled program execution and input/output.

DOS was developed for the Apple II computer. It provided the first capability for storage and retrieval of various types of files on disk (the Disk II); the System Monitor had allowed input/output (of binary data) to cassette tape only.

The latest version of DOS is DOS 3.3. It uses a 16-sector disk format, like ProDOS 8 and ProDOS 16. Earlier versions use a 13-sector format that cannot be read by ProDOS 8 or ProDOS 16.

SOS

SOS is the operating system developed for the Apple III computer. Its name is an acronym for *Sophisticated Operating System*, reflecting its increased capabilities over DOS. On the other hand, SOS requires far more memory space than either DOS or ProDOS 8 (below), which makes it impractical on computers with less than 256K of RAM.

ProDOS 8

ProDOS 8 (for *Professional Disk Operating System*) was developed for the newer members of the Apple II family of computers. It requires at least 64K of RAM memory, and can run on the Apple IIe, Apple IIc, and 64K Apple II Plus.

ProDOS 8 brings some of the advanced features of SOS to the Apple II family, without requiring as much memory as SOS does. Its commands are essentially a subset of the SOS commands.

The latest version of ProDOS 8 developed specifically for the Apple IIe and IIc is ProDOS 8 (1.1.1). An even more recent version, developed for the Apple IIGS but compatible with the IIe and IIc, is ProDOS 8 (1.2).

❖ *Note:* Prior to development of ProDOS 16, ProDOS 8 was called simply *ProDOS*.

ProDOS 16

ProDOS 16 is an extensive revision of ProDOS 8, developed specifically for the Apple IIGS (it will not run on other Apple II's). The *16* refers to the 16-bit internal registers in the Apple IIGS 65C816 microprocessor.

ProDOS 16 permits access to the entire 16 Mb addressable memory space of the Apple IIGS (ProDOS 8 is restricted to addressing 64K) and it has more "SOS-like" features than ProDOS 8 has. It also has some new features, not present in SOS, that ease program development.

There are two versions of ProDOS 16. Version 1.0 is a first-release system, consisting of a ProDOS 8 core surrounded by a "ProDOS 16-like" user interface. Version 2.0 is the complete implementation of the ProDOS 16 design.

Pascal

The Pascal operating system for the Apple II is modified and extended from UCSD Pascal, developed at the University of California at San Diego. The latest version, written for the Apple IIe/IIc and 64K Apple II Plus, is Pascal 1.3. It also runs on an Apple IIGS.

Pascal for the Apple III is a modified version of Apple II Pascal. It uses SOS for most of its operating system functions.

File compatibility

ProDOS 16, ProDOS 8, and SOS all use a hierarchical file system with the same format and organization. Every file on one system's disk can be read by either of the other systems. DOS and Pascal use significantly different formats.

The other systems compare to ProDOS 16 as follows:

ProDOS 8: ProDOS 16 and ProDOS 8 have identical file system organizations—therefore, ProDOS 16 can read all ProDOS 8 files. However, the System Loader under ProDOS 16 will not *execute* ProDOS 8 executable binary files (type $06). Likewise, ProDOS 8 can read but will not execute file types $B3–$BE; those file types are specific to ProDOS 16.

SOS: ProDOS 16 and SOS have identical file system organizations —therefore, ProDOS 16 can read (but not execute) all SOS files.

DOS: DOS does not have a hierarchical file system. ProDOS 16 cannot directly read DOS files (but see "Reading DOS 3.3 and Apple II Pascal Disks," in the following section).

Pascal: Apple II Pascal does not have a hierarchical file system. ProDOS 16 cannot directly read Apple II Pascal files (but see "Reading DOS 3.3 and Apple II Pascal Disks," below).

Apple III Pascal uses the SOS file system. Therefore ProDOS 16 can read (but not execute) all Apple III Pascal files.

Reading DOS 3.3 and Apple II Pascal disks

Both DOS 3.3 and ProDOS 8 140K flexible disks are formatted using the same 16-sector layout. As a consequence, the ProDOS 16 `READ_BLOCK` and `WRITE_BLOCK` calls are able to access DOS 3.3 disks too. These calls know nothing about the organization of files on either type of disk.

When using `READ_BLOCK` and `WRITE_BLOCK`, you specify a 512-byte block on the disk. When using RWTS (the DOS 3.3 counterpart to `READ_BLOCK` and `WRITE_BLOCK`), you specify the track and sector of a 256-byte chunk of data, as explained in the *DOS Programmer's Manual*. To use `READ_BLOCK` and `WRITE_BLOCK` to access DOS 3.3 disks, you must know what 512-byte block corresponds to the track and sector you want.

Table B-1 shows how to determine a block number from a given track and sector. First multiply the track number by 8, then add the sector offset that corresponds to the sector number. The half of the block in which the sector resides is determined by the half-of-block line (1 is the first half; 2 is the second).

Table B-1
Tracks and sectors to blocks (140K disks)

Block number = (8*track number) + sector offset

Sector:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Sector offset:	0	7	6	6	5	5	4	4	3	3	2	2	1	1	0	7
Half of block:	1	1	2	1	2	1	2	1	2	1	2	1	2	1	2	2

Refer to the *DOS Programmer's Manual* for a description of the file organization of DOS 3.3 disks.

Operating system similarity

This section compares the functional similarities among the operating systems. Functional similarity between two systems implies that they perform closely related operations, but it does not mean that thay have identical procedures or commands.

Input/Output

ProDOS 16 can perform I/O operations on files in disk drives (block devices) only. Under ProDOS 16, therefore, the current application is responsible for knowing the protocol necessary to communicate with character devices (such as the console, printers, and communication ports).

The other systems compare to ProDOS 16 as follows:

ProDOS 8: Like ProDOS 16, ProDOS 8 performs I/O on block devices only.

SOS: SOS communicates with all devices, both character devices and block devices, by making appropriate *file* access calls (such as open, read write, close). Under SOS, writing to one device is essentially the same as writing to another.

DOS: DOS allows communication with one type of device only—the Disk II drive. DOS 3.3 uses a 16-sector disk format; earlier versions of DOS use a 13-sector format. 13-sector Disk II disks cannot be read directly by DOS 3.3, SOS, ProDOS 8, or ProDOS 16.

Pascal: Apple II and Apple III Pascal provide access to both block devices and character devices, through *File I/O*, *Block I/O*, and *Device I/O* calls to the volumes on the devices.

Filing calls

SOS, ProDOS 8, and ProDOS 16 filing calls are all closely related. Most of the calls are shared by all three systems; furthermore, their numbers are identical in ProDOS 8 and SOS (ProDOS 16 calls have a completely different numbering system from either ProDOS 8 or SOS).

The other systems compare to ProDOS 16 as follows:

ProDOS 8: The ProDOS 8 `ON_LINE` call corresponds to the ProDOS 16 `VOLUME` call. When given a device name, `VOLUME` returns the volume name for that device. When given a unit number (derived from the slot and drive numbers), `ON_LINE` returns the volume name.

The ProDOS 8 `RENAME` call corresponds to the ProDOS 16 `CHANGE_PATH` call, except that `RENAME` can change only the *last* name in a pathname.

SOS: The SOS `GET_FILE_INFO` call returns the size of the file (the value of EOF). With ProDOS 16 you must first open the file and then use the `GET_EOF` call.

The SOS `VOLUME` call corresponds to the ProDOS 16 `VOLUME` call. When given a device name, `VOLUME` returns the volume name for that device.

The SOS calls `SET_MARK` and `SET_EOF` can use a displacement from the current position in the file. ProDOS 16 accepts only absolute positions in the file for these calls.

DOS: DOS calls distinguish between *sequential-access* and *random-access* text files. ProDOS 16 makes no such distinction, although the ProDOS 16 `READ` call in NEWLINE mode functions as a sequential-access read.

DOS uses `APPEND` and `POSITION` commands, roughly similar to ProDOS 16's `SET_MARK`, to set the current position in the file and to automatically extend the size of the file.

The `CLOSE` command in DOS can be given in immediate (from the keyboard) or deferred (in a program) mode. No ProDOS 16 commands can be given in immediate mode.

Pascal: Apple II Pascal distinguishes among *text files*, *data files*, and *code files*, each with different header formats; all ProDOS 16 files have identical header formats. The Pascal procedures `REWRITE` and `RESET` correspond to ProDOS 16's `CREATE` and `OPEN` calls. Pascal has more procedures for reading from and writing to files and devices than does ProDOS 16.

Because Apple III Pascal uses the SOS file system, its filing calls correspond directly to SOS calls.

Memory management

Under ProDOS 16, neither the operating system nor the application program perform memory management; allocation of memory is the responsibility of the Memory Manager, an Apple IIGS ROM-based tool set. When an application needs space for its own use, it makes a direct request to the Memory Manager. When it makes a ProDOS 16 call that requires the allocation of memory space, ProDOS 16 makes the appropriate request to the Memory Manager. The Apple IIGS Memory Manager is similar to the SOS memory manager, except that it is more sophisticated and is not considered part of the operating system.

The other systems compare to ProDOS 16 as follows:

ProDOS 8: A ProDOS 8 application is responsible for its own memory management. It must find free memory, and then allocate it by marking it off in the ProDOS 8 global page's memory bit map. ProDOS 8 protects allocated areas by refusing to write to any pages that are marked on the bit map. Thus it prevents the user from destroying protected memory areas (as long as all allocated memory is properly marked off, and all data is brought into memory using ProDOS 8 calls).

SOS: SOS has a fairly sophisticated Memory Manager that is part of the operating system itself. An application requests memory from SOS, either by location or by the amount needed. If the request can be satisfied, SOS grants it. That portion of memory is then the sole responsibility of the requestor until it is released.

DOS: DOS performs no memory management. Each application under DOS is completely responsible for its own memory allocation and use.

Pascal: Apple II Pascal uses a simple memory management system that controls the loading and unloading of code and data segments and tracks the size of the stack and heap.

Apple III Pascal uses SOS for memory management.

Interrupts

ProDOS 16 does not have any built-in interrupt-generating device drivers. Interrupt handling routines are therefore installed into ProDOS 16 separately, using the `ALLOC_INTERRUPT` call. When an interrupt occurs, ProDOS 16 polls the handling routines in succession until one of them claims the interrupt.

The other systems compare to ProDOS 16 as follows:

ProDOS 8: ProDOS 8 handles interrupts identically to ProDOS 16, except that it allows fewer installed handlers (4 vs. 16).

SOS: In SOS, any device capable of generating an interrupt must have a device driver capable of handling the interrupt; the device driver and its interrupt handler are inseparable and are considered to be part of SOS. In addition, SOS assigns a distinct interrupt priority to each device in the system.

DOS: DOS does not support interrupts.

Pascal: Apple II Pascal versions 1.2 and 1.3 support interrupts; earlier versions of Apple II Pascal do not.

Apple III Pascal uses the SOS interrupt system.

Appendix C

The ProDOS 16 Exerciser

The *ProDOS 16 Exerciser* is a program that lets you practice making operating system calls without writing an application. All ProDOS 16 functions execute just as they would when called from a program; therefore you can test how the calls work and, if necessary, correct any programming errors before coding your routines.

Starting the Exerciser

First, make a copy of the Exerciser disk and put the original away in a safe place. Consult your owner's manual if you need instructions on how to copy a disk.

The Exerciser may be the startup program on the diskette provided with this manual. If so, it should execute automatically when you turn on the machine and insert the diskette. Otherwise, select it from the desktop or program launcher that comes up when you start up the system. The program's filename is `EXERCISER`.

The first display is the menu screen. It shows all ProDOS 16 calls by number and name, as well as a few other commands you may enter. The menu screen always returns between execution of calls or commands

Making system calls

You make system calls from the exerciser by entering their call numbers. The number you enter is displayed at the bottom of the menu screen. You may clear the number at any time by pressing zero twice in succession.

After entering the number, press the Return key. The parameter block for the call you selected is displayed. Enter a value (or select the default provided by pressing the Return key) for each parameter; each time you press Return, the cursor moves downward one position in the parameter block. The cursor does not stop at any parameter that is a result *only* (that has no input value).

❖ *Note:* If, *while you are entering parameters*, you wish to correct a value, press the Escape key—it positions the cursor back at the top of the parameter block. At any other time, however, the Escape key returns you to the main menu.

Pathnames and other text strings are passed to and from ProDOS 16 in buffers referenced by pointers in the parameter blocks. Therefore, to enter or read a pathname you must provide a buffer for ProDOS 16 to read from or write to. In most cases, the Exerciser sets up a default buffer, pointed to by a default pointer parameter (see, for example, the `CREATE` call). The contents of the location referenced by that pointer are displayed on the screen, below the parameter block. For convenience, you can directly edit the displayed string on the screen; you needn't access the memory location itself.

After you have entered all the required parameters, press the Return key once more to execute the call. If everything has gone right, the parameter list now contains results returned by ProDOS 16, and the message "$00 call successful" appears at the bottom of the screen. If a ProDOS 16 error occurs, the proper error number and message are displayed instead. In addition, if an error occurs a small "c" should appear at the lower right corner of the screen, to indicate that the microprocessor's carry bit has been set.

Warning The Exerciser does not protect you from serious mistakes. With a `WRITE_BLOCK` call you can easily overwrite a critical block on one of your disks, destroying valuable file data or even the disk's directory. With a careless `FORMAT` call, you can destroy *all information* on your disk. *Be careful how you use this program!*

Other commands

In addition to practicing system calls, you may issue commands that allow you to list the contents of a directory, modify any part of the Apple IIGS RAM memory, enter the Monitor program, or quit the Exerciser.

List Directory (L)

Press L and you are prompted for the pathname of the volume or subdirectory whose contents you wish to list. For each file in the directory, the listing shows file name, file type (see table A-2), number of blocks used, date and time of last modification, date and time of creation, EOF (logical size in bytes), and subtype (value of the auxiliary type field). Press the Escape key to return to the main menu.

Modify Memory (M)

You use the Modify Memory command to place data in memory for ProDOS 16 to read, or to inspect the contents of a buffer that ProDOS 16 has written to.

Press M and you are prompted for a pointer to the part of memory you wish to access. Enter the proper address and press the Return key. A 256-byte (one-page) portion of memory is displayed, as 16 rows of 16 bytes each, beginning on a page boundary. Each row is preceded by the address of the first byte in that row; to the right of each row are the ASCII representations of the values of the bytes in the row.

Use the arrow keys to move the cursor around on the screen. To change the value of a byte, type the new value right over the old one. You can enter data in hexadecimal format only; the results of your entry are displayed on the screen in both hexadecimal and ASCII. For reference, Table C-1 lists ASCII characters and their decimal, hexadecimal, and binary equivalents.

You may undo up to the last 16 changes you made by typing U successively. To display the preceding or succeeding page in memory, press < or >.

Table C-1
ASCII character set

Char	Dec	Hex	Binary	Char	Dec	Hex	Binary
nul	0	0	00000000	(	40	28	00101000
soh	1	1	00000001	)	41	29	00101001
stx	2	2	00000010	*	42	2A	00101010
etx	3	3	00000011	+	43	2B	00101011
eot	4	4	00000100	,	44	2C	00101100
enq	5	5	00000101	-	45	2D	00101101
ack	6	6	00000110	.	46	2E	00101110
bel	7	7	00000111	/	47	2F	00101111
ht	9	9	00001001	0	48	30	00110000
lf	10	A	00001010	1	49	31	00110001
vt	11	B	00001011	2	50	32	00110010
ff	12	C	00001100	3	51	33	00110011
cr	13	D	00001101	4	52	34	00110100
so	14	E	00001110	5	53	35	00110101
si	15	F	00001111	6	54	36	00110110
dle	16	10	00010000	7	55	37	00110111
dc1	17	11	00010001	8	56	38	00111000
dc2	18	12	00010010	9	57	39	00111001
dc3	19	13	00010011	:	58	3A	00111010
dc4	20	14	00010100	;	59	3B	00111011
nak	21	15	00010101	<	60	3C	00111100
syn	22	16	00010110	=	61	3D	00111101
etb	23	17	00010111	>	62	3E	00111110
can	24	18	00011000	?	63	3F	00111111
em	25	19	00011001	@	64	40	01000000
sub	26	1A	00011010	A	65	41	01000001
esc	27	1B	00011011	B	66	42	01000010
fs	28	1C	00011100	C	67	43	01000011
gs	29	1D	00011101	D	68	44	01000100
rs	30	1E	00011110	E	69	45	01000101
us	31	1F	00011111	F	70	46	01000110
sp	32	20	00100000	G	71	47	01000111
!	33	21	00100001	H	72	48	01001000
"	34	22	00100010	I	73	49	01001001
#	35	23	00100011	J	74	4A	01001010
$	36	24	00100100	K	75	4B	01001011
%	37	25	00100101	L	76	4C	01001100
&	38	26	00100110	M	77	4D	01001101
'	39	27	00100111	N	78	4E	01001110

Table C-1 (continued)
ASCII character set

Char	Dec	Hex	Binary	Char	Dec	Hex	Binary
O	79	4F	01001111	h	104	68	01101000
P	80	50	01010000	i	105	69	01101001
Q	81	51	01010001	j	106	6A	01101010
R	82	52	01010010	k	107	6B	01101011
S	83	53	01010011	l	108	6C	01101100
T	84	54	01010100	m	109	6D	01101101
U	85	55	01010101	n	110	6E	01101110
V	86	56	01010110	o	111	6F	01101111
W	87	57	01010111	p	112	70	01110000
X	88	58	01011000	q	113	71	01110001
Y	89	59	01011001	r	114	72	01110010
Z	90	5A	01011010	s	115	73	01110011
[	91	5B	01011011	t	116	74	01110100
\	92	5C	01011100	u	117	75	01110101
]	93	5D	01011101	v	118	76	01110110
^	94	5E	01011110	w	119	77	01110111
_	95	5F	01011111	x	120	78	01111000
`	96	60	01100000	y	121	79	01111001
a	97	61	01100001	z	122	7A	01111010
b	98	62	01100010	{	123	7B	01111011
c	99	63	01100011	|	124	7C	01111100
d	100	64	01100100	}	125	7D	01111101
e	101	65	01100101	~	126	7E	01111110
f	102	66	01100110	del	127	7F	01111111
g	103	67	01100111				

Warning Modify Memory does not prevent you from changing values in parts of memory that are already in use. You can conceivably alter the Exerciser itself or other critical code, causing a system crash. Be careful what you modify!

Exit to Monitor (X)

The Monitor is a firmware program (see *Apple IIGS Firmware Reference*) that allows you to inspect and modify the contents of memory, assemble and disassemble code in a limited manner, and execute code in memory. You may enter the Monitor from the ProDOS 16 Exerciser.

To call the Monitor, press M. When the Monitor prompt (*) appears, you may issue any Monitor command. To leave the Monitor and return to the Exerciser, you must reboot the computer (press Control–Ⓞ–Reset) and, if necessary, re-execute the Exerciser from the desktop or program launcher.

Quit (Q)

To quit the ProDOS 16 exerciser, simply press Q. Of course, you may also quit by selecting the ProDOS 16 `QUIT` call ($27), filling out the parameter block, and executing the call.

Appendix D

System Loader Technical Data

This appendix assembles some specific technical details on the System Loader. For more information, see the referenced publications.

Object module format

The System Loader can load only code and data segments that conform to Apple IIGS **object module format**. Object module format is described in detail in *Apple IIGS Programmer's Workshop Reference*.

File types

File types for load files and other OMF-related files are listed below. For a complete list of ProDOS file types, see Table A-2 in Appendix A.

File type	Description
$B0	Source file (*aux_type* defines language)
$B1	Object file
$B2	Library file
$B3	Application file
$B4	Run-time library file
$B5	Shell application file
$B6 - $BE	*Reserved for system use. Currently defined types include:*
$B6	Permanent inititialization file
$B7	Temporary initialization file
$B8	New desk accessory
$B9	Classic desk accessory

Segment kinds

Whereas files are classified by type, segments are classified by **kind**. Each segment has a kind designation in the KIND field of its header. The five high-order bits in the KIND field describe specific attributes of the segment; the value in the low-order five-bit field describes the overall type of segment. Different combinations of attributes and type values yield different results for the segment kind.

The KIND field is two bytes long. Figure D-1 shows its format.

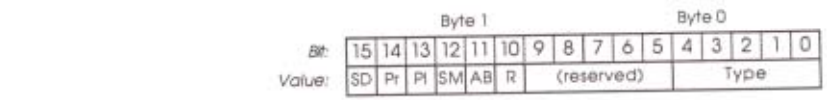


Figure D-1
Segment kind format

where the *attribute* bits (11-15) mean the following:

SD (bit 15)	=	static/dynamic	(0 = static; 1 = dynamic)
Pr (bit 14)	=	private	(0 = no; 1 = yes)
PI (bit 13)	=	position-independent	(0 = no; 1 = yes)
SM (bit 12)	=	may be in special memory	(0 = yes; 1 = no)

AB (bit 11)	=	absolute-bank	(0 = no; 1 = yes)
R (bit 10)	=	Reload	(0 = no, 1 = yes)

and the *type* field (bits 0-4) describes one of the following classifications of the segment:

Value of Type	Description
$00	code segment
$01	data segment
$02	Jump Table segment
$04	Pathname segment
$08	library dictionary segment
$10	initialization segment
$12	direct-page/stack segment

Segment attributes can be combined with particular types to yield different resultant values for KIND. For example, a *dynamic* Initialization Segment has KIND = $8010.

❖ *Note:* A Reload segment is always loaded from the file when a program starts up, even if the program is restarted from memory. It is used to initialize data for programs that would not otherwise be restartable.

Record codes

Load segments, like all OMF segments, are made up of records. Each type of record has a code number and a name. For a complete list of record types, see *Apple IIGS Programmer's Workshop Reference*. The only record types recognized by the System Loader are these:

Record Code	Name	Description
$E2	RELOC	intrasegment relocation record (in relocation dictionary)
$E3	INTERSEG	intersegment relocation record (in relocation dictionary)
$F1	DS	zero-fill record
$F2	LCONST	long-constant record (the actual code and data for each segment)
$F5	cRELOC	compressed intrasegment relocation record (in relocation dictionary)

Record Code	Name	Description
$F6	cINTERSEG	compressed intersegment relocation record (in relocation dictionary)
$F7	SUPER	super-compressed relocation record (the equivalent of many cRELOC or cINTERSEG records)
$00	END	the end of the segment

If the loader encounters any other type of record in a load segment, it returns error $110A.

Load-file numbers

Load files processed by the Apple IIGS Programmer's Workshop Linker at any one time are numbered consecutively from 1. Load file 1 is called the initial load file. All other files are considered to be run-time libraries.

A load-file number of 0 in a Jump Table segment or a Pathname segment indicates the end of the segment.

Load-segment numbers

In each load file created by the linker, segments are numbered consecutively by their position in the load file, starting at 1. The loader determines a segment's number by counting its position from the beginning of the load file. As a check, the loader also looks at the segment number in the segment's header.

The first static segment in a load file, which need not be segment number 1, is called the *main* segment—it is loaded first (except for any preceding initialization segments) and never leaves memory while the program is executing. Because a run-time library need have no static segments at all, it typically has no main segment.

Segment headers

The first part of every object module format segment is a *segment header*; it contains 17 fields that give the name, size, and other important information about the segment.

Restrictions on segment header values

Because OMF supports capabilities that are more general than the System Loader's needs, the System Loader permits load files to have only a subset of all possible OMF characteristics. The loader does this by restricting the values of several segment header fields:

NUMSEX:	must be 0
NUMLEN:	must be 4
BANKSIZE:	must be less than or equal to $10 000
ALIGN:	must be less than or equal to $10 000

If the System Loader finds any other values in any of the above fields, it returns error $110B ("Segment is Foreign"). The restrictions on BANKSIZE and ALIGN are enforced by the APW Linker also.

Page-aligned and bank-aligned segments

In OMF, the values of BANKSIZE and ALIGN may be any multiple of 2. But because the Memory Manager and System Loader support only two types of alignment (page- and bank-alignment) and one bank size (64K), the System Loader uses both BANKSIZE and ALIGN values to control segment alignment, as follows.

1. If BANKSIZE is 0 or $10 000, its value has no effect on segment alignment.
2. If BANKSIZE is any other value, the greater of BANKSIZE and ALIGN is called the *alignment factor*. Alignment in memory is controlled by the alignment factor in this way:
   a. If the alignment factor is 0, the segment is not aligned to any memory boundary.
   b. If the alignment factor is greater than 0 and less than or equal to $100, the segment is page-aligned.
   c. If the alignment factor is greater than $100, the segment is bank-aligned.

❖ *Note:* The Memory Manager itself does not directly support bank-alignment. The System Loader forces bank alignment where needed by requesting blocks in successive banks until it finds one that starts on a bank boundary.

Entry point and global variables

There is only one entry point needed for all System Loader calls (actually, all tool calls). It is to the Apple IIGS tool dispatcher, at the bottom of bank $E1 (address $E1 0000). Although the System Loader maintains memory space and a table of loader functions in other parts of memory, locations in those areas are not supported. Please make all System Loader calls with a JSL to $E1 0000, as explained in Chapter 17 (or with macro calls or other higher-level interface, if appropriate for your language).

The following variables are of global significance. They are defined at the system level, so any application that needs to know their values may access them. However, only USERID is important to most applications, and it should be accessed only through proper calls to the System Loader. The other variables are needed by controlling programs only, and should not be used by applications.

`SEGTBL`	Absolute address of the Memory Segment Table
`JMPTBL`	Absolute address of the Jump Table Directory
`PATHTBL`	Absolute address of the Pathname Table
`USERID`	User ID of the current application

User ID format

The User ID Manager is discussed in Chapter 5, and fully explained in *Apple IIGS Toolbox Reference*. Only the format of the User ID number, needed as a parameter for System Loader calls, is shown here.

There is a 2-byte User ID associated with every allocated memory block. It is divided into three fields: `MainID`, `AuxID`, and `TypeID`. The `MainID` field contains the unique number assigned to the owner of the block by the User ID Manager; every allocated block has a nonzero value in its MainID field. The `AuxID` field holds a user-assignable identification; it is ignored by the System Loader, ProDOS 16, and the User ID Manager. The `TypeID` field gives the general class of software to which the block belongs.

Figure D-2
User ID format

`MainID` can have any value from $01 to $FF (0 is reserved).

`AuxID` can have any value from $00 to $0F.†

`TypeID` values are defined as follows:

$00 Memory Manager
$01 application
$02 controlling program
$03 ProDOS 8 and ProDOS 16
$04 tool set†
$05 desk accessory
$06 run-time library
$07 System Loader
$08 firmware/system function
$09 Tool Locator
$0A-F (undefined)

†If type ID = $04, these values of AuxID are reserved:

$01 Miscellaneous Toolset file
$02 Scrap Manager file
$0A tool setup file

Appendix E

Error Codes

This appendix lists and describes all error codes returned by ProDOS 16 and the System Loader. Each error code is followed by the error's suggested name or screen message, and a brief description of its significance.

When an error occurs during a call, ProDOS 16 or the System Loader places the error number in the accumulator (A-register), sets the status register carry bit, and returns control to the calling routine.

If, after a call, the carry bit is clear and the accumulator contains 0, that signifies a successful completion (no error).

ProDOS 16 errors

Nonfatal errors

A nonfatal error signifies that a requested call could not be completed properly, but program execution may continue.

Number	Message and Description
	General Errors:
$00	(no error)
$01	**Invalid call number:** A nonexistent command has been issued.
$07	**ProDOS is busy:** The call cannot be made because ProDOS 16 is busy with another call.
	Device call errors:
$10	**Device not found:** There is no device on line with the given name (`GET_DEV_NUM` call).
$11	**Invalid device request:** The given device name or reference number is not in ProDOS 16's list of active devices (`VOLUME`, `READ_BLOCK` and `WRITE_BLOCK` calls).
$25	**Interrupt vector table full:** The maximum number of user-defined interrupt handlers (16) has already been installed; there is no room for another (`ALLOC_INTERRUPT` call).
$27	**I/O error:** A hardware failure has prevented proper data transfer to or from a disk device. This is a general code covering many possible error conditions.
$28	**No device connected:** There is no device in the slot and drive specified by the given device number (`READ_BLOCK`, `WRITE_BLOCK`, and `VOLUME` calls).
$2B	**Write-protected:** The specified volume is write-protected (the "write-protect" tab or notch on the diskjacket has been enabled). No operation that requires writing to the disk can be performed.
$2D	**Invalid block address:** An attempt was made to read data from a RAM disk, at an address beyond its limits.
$2E	**Disk switched:** The requested operation cannot be performed because a disk containing an open file has been removed from its drive.

Warning Apple II drives have no hardware method for detecting disk switches. This error is therefore returned only when ProDOS 16 checks a volume name during the normal course of a call. Since most disk access calls do not involve a check of the volume name, a disk-switched error can easily go undetected.

$2F **Device not on line:** A device specified in a call is not connected to the system, or has no volume mounted on it. This error may be returned by device drivers that can sense whether or not a specific device is on line.

$30 - $3F **Device-specific errors:** (error codes in this range are to be defined and used by individual device drivers.)

File call errors:

$40 **Invalid pathname or device name syntax:** The specified pathname or device name contains illegal characters (other than A–Z, 0–9, . ,/,*).

$42 **FCB table full:** The table of file control blocks is full; the maximum permitted number of open files (8) has already been reached. You may not open another file (OPEN call).

$43 **Invalid file reference number:** The specified file reference number does not match that of any currently open file.

$44 **Path not found:** A subdirectory name in the specified pathname does not exist (the pathname's syntax is otherwise valid).

$45 **Volume not found:** The volume name in the specified pathname does not exist (the pathname's syntax is otherwise valid).

$46 **File not found:** The last file name in the specified pathname does not exist (the pathname's syntax is otherwise valid).

$47 **Duplicate pathname:** An attempt has been made to create or rename a file, using an already existing pathname (CREATE, CHANGE_PATH calls).

$48 **Volume full:** An attempt to allocate blocks on a disk device has failed, due to lack of space on the volume in the device (CREATE, WRITE calls). If this error occurs during a write, ProDOS 16 writes data is until the disk is full, and still permits you to close the file.

$49 **Volume directory full:** No more space for entries is left on the volume directory (CREATE call). In ProDOS 16, a volume directory can hold no more than 51 entries. No more files can be added to this directory until others are destroyed (deleted).

$4A **Version error (incompatible file format):** The version number in the specified file's directory entry does not match the present ProDOS 8-ProDOS 16 file format version number. This error can only occur in future versions of ProDOS 16, since for all present versions of ProDOS 8 and ProDOS 16 the file format version number is zero.

❖ *Note:* The version number referred to by this error code concerns the file format only, not the version number of the operating system as a whole. In particular, it is unrelated to the ProDOS 16 version number returned by the GET_VERSION call.

$4B **Unsupported (or incorrect) storage type:** The organization of the specified file is unknown to ProDOS 16. See Appendix A for a list of valid storage types.

This error may also be returned if a directory has been tampered with, or if a prefix has been set to a nondirectory file.

$4C **End-of-file encountered (out of data):** A read has been attempted, but the current file position (Mark) is equal to end-of-file (EOF), and no further data can be read.

$4D **Position out of range:** The specified file position parameter (Mark) is greater than the size of the file (EOF).

$4E **Access not allowed:** One of the attributes in the specified file's access byte forbids the attempted operation (renaming, destroying, reading, or writing).

$50 **File is open:** An attempt has been made to perform a disallowed operation on an open file (`OPEN`, `CHANGE_PATH`, `DESTROY` calls).

$51 **Directory structure damaged:** The number of entries indicated in the directory header does not match the number of entries the directory actually contains.

$52 **Unsupported volume type:** The specified volume is not a ProDOS 16, ProDOS 8, or SOS disk. Its directory format is incompatible with ProDOS 16.

$53 **Parameter out of range:** The value of one or more parameters in the parameter block is out of its range of permissible values.

$54 **Out of Memory:** A ProDOS 8 program specified by the `QUIT` call is too large to fit into the memory space available for ProDOS 8 applications.

$55 **VCB table full:** The table of volume control blocks is full; the maximum permitted number of online volumes/devices (8) has already been reached. You may not add another device to the system. The error occurs when 8 devices are on line and a `VOLUME` call is made for another device that has no open files.

$57 **Duplicate volume:** Two or more online volumes have identical volume directory names. This message is a warning; it does not prevent access to either volume. However, ProDOS 16 has no way of knowing which volume is intended if the volume name is specified in a call; it will access the first one it finds.

$58 **Not a block device:** An attempt has been made to access a device that is not a block device. Current versions of ProDOS 16 support access to block devices only.

$59 **Invalid level:** The value specified for the system file level is out of range (`SET_LEVEL` call).

$5A **Block number out of range:** The volume bit map indicates that the volume contains blocks beyond the block count for the volume. This error may indicate a damaged disk structure.

$5B **Illegal pathname change:** The pathnames on a `CHANGE_PATH` call specify two different volumes. `CHANGE_PATH` can move files among directories only on the *same* volume.

$5C **Not an executable file:** The file specified in a `QUIT` call is not a launchable type. All applications launched by the `QUIT` call must be type $B3 (ProDOS 16 application), $B5 (shell application), or $FF (ProDOS 8 system file).

$5D **Operating system/file system not available:** (1) The `QUIT` call has specified a ProDOS 8 application to be launched, but the ProDOS 8 operating system is not on the system disk. (2) The `FORMAT` call is unable to format a disk for the specified file system.

$5E: **Cannot deallocate /RAM:** In quitting from a ProDOS 8-based program and launching a ProDOS 16-based program, `PQUIT` is not able to remove the ProDOS 8 RAM disk in bank $01 (`QUIT` call).

$5F **Return stack overflow:** An attempt was made to add another User ID to the return stack maintained by `PQUIT`, but the stack already has 16 entries, its maximum permitted number (`QUIT` call).

$60 **Data unavailable:** The system has invalid information on which device was last accessed (`GET_LAST_DEV` call).

Fatal errors

A fatal error signifies the occurrence of a malfunction so serious that processing must halt. To resume execution following a fatal error, you must reboot the system.

Number	Message and Description
$01	**Unclaimed interrupt:** An interrupt signal has occurred and none of the installed handlers claimsresponsibility for it. This error may occur if interrupt-producing hardware is installed before its associated interrupt handler is allocated.
$0A	**VCB unusable:** The volume control block table has been damaged. The values of certain check bytes are not what they should be, so ProDOS 16 cannot use the VCB table.
$0B	**FCB unusable:** The file control block table has been damaged. The values of certain check bytes are not what they should be, so ProDOS 16 cannot use the FCB table.
$0C	**Block zero allocated illegally:** Write-access to block zero on a disk volume has been attempted. Block zero on all volumes is reserved for boot code.
$0D	**Interrupt occurred while I/O shadowing off:** The Apple IIGS has soft switches that control shadowing from banks $E0 and $E1 to banks $00 and $01. If an interrupt occurrs while those switches are off, the firmware interrupt-handling code will not be enabled. See *Apple IIGS Firmware Reference*.
$11	**Wrong OS version:** The version number of the file `P16` or `P8` is different from the version number of the file `PRODOS`. `PRODOS`, which loads ProDOS 16 (`P16`) and ProDOS 8 (`P8`), requires compatible versions of both.

If a `QUIT` call results in the loading of a ProDOS 16-based application that is too large to fit in the available memory or that for some other reason cannot be loaded, execution halts and the following message is displayed on the screen:

```
Can't run next application. Error=$XXXX
```

where $*XXXX* is an error code—typically a Tool Locator, Memory Manager, or System Loader error code.

Bootstrap errors

Bootstrap errors can occur when the Apple IIGS attempts to start up a ProDOS 16 system disk. Errors can occur at several points in this process:

1. If there is no disk in the startup drive, a "sliding apple" symbol (_🍎—)appears on the screen along with the message:

   ```
   Check startup device!
   ```

   Place a system disk in the drive and press Control-⏏-Reset to restart the boot procedure.

2. If there is a disk in the drive, but it is not a ProDOS 8 or ProDOS 16 system disk (that is, there is no type $FF file named `PRODOS` on it), the following message appears:

   ```
   UNABLE TO LOAD PRODOS
   ```

   Remove the disk and replace it with another containing the proper files, then press Control-⏏-Reset to restart the boot procedure.

3. If the file named `PRODOS` is found, but another essential file is missing, a message such as

   ```
   No SYSTEM/P16 file found
   ```

   or

   ```
   No x.SYSTEM or x.SYS16 file found
   ```

   may appear. Remove the disk and replace it with another containing the proper files, then press Control-⏏-Reset to restart the boot procedure.

Another type of ProDOS 16 bootstrap error occurs on other Apple II systems. If you try to boot a ProDOS 16 system disk on a standard Apple II computer (one that is *not* an Apple IIGS), the following error message is displayed:

```
PRODOS 16 REQUIRES APPLE IIGS HARDWARE
```

When this occurs the disk will not boot. You can boot an Apple IIGS System Disk only on an Apple IIGS computer.

System Loader errors

Nonfatal errors

Number	Message and Description
$0000	(no error)
$1101	**Not found:** The specified segment (in the load file) or entry (in the Pathname Table or Memory Segment Table) does not exist. If the specified load file itself is not found, a ProDOS 16 error $46 (file not found) is returned.
$1102	**Incompatible OMF version:** The object module format version of a load segment (as specified in its header) is incompatible with the current version of the System Loader. The loader will not load such a segment.
$1104	**File is not a load file:** The specified load file is not type $B3-$BE. See Appendix A or D for descriptions of these file types.
$1105	**Loader is busy:** The call cannot be made because the System Loader is busy with another call.
$1107	**File version error:** The specified file cannot be loaded because its creation date and time do not match those on its entry in the Pathname Table.
❖ *Note:* This error applies to run-time library files only.	
$1108	**User ID error:** The specified User ID either doesn't exist (Application Shutdown), or doesn't match the User ID of the specified segment (Unload Segment By Number).
$1109	**SegNum out of sequence:** The value of the SEGNUM field in the segment's header doesn't match the number by which the segment was specified (Load Segment By Number, Initial Load).
$110A	**Illegal load record found:** A record in the segment is of a type not accepted by the loader.
$110B	**Load segment is foreign:** The values in the NUMSEX and NUMLEN fields in the specified segment's header are not 0 and 4, respectively (Load Segment By Number).
$001-$05F	(ProDOS 16 I/O errors—see "ProDOS 16 Errors" in this appendix.)
$201-$20A	(Memory Manager errors—see *Apple IIGS Toolbox Reference.*)

Fatal errors

If a ProDOS 16 error or Memory Manager error occurs while the System Loader is making an internal call, it is a fatal error. The most common case is when a Jump Table Load is attempted for a dynamic load segment or run-time library segment whose volume is not on line. Control is transferred to the System Failure Manager, and the following message appears on the screen:

```
Error loading Dynamic Segment-XXXX
```

where *XXXX* is the error code of the ProDOS 16 or Memory manager error that occurred.

Glossary

absolute: Characteristic of a load segment or other program code that must be loaded at a specific address in memory, and never moved. Compare **relocatable.**

access byte: An attribute of a ProDOS 16 file that determines what types of operations, such as reading or writing, may be performed on the file.

accumulator: The register in the microprocessor where most computations are performed.

address: A number that specifies the location of a single **byte** of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (in decimal) or from $00 00 00 to $FF FF FF (in hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($00 00 to $FF FF).

Apple IIGS Programmer's Workshop: The development environment for the Apple IIGS computer. It consists of a set of programs that facilitate the writing, compiling, and debugging of Apple IIGS applications.

application program (or **application**)**:** (1) A program that performs a specific task useful to the computer user, such as word processing, data base management, or graphics. Compare **controlling program, shell application, system program.** (2) On the Apple IIGS, a program that accesses ProDOS 16 and the Toolbox directly, and that can be called or exited via the QUIT call. ProDOS 16 applications are **file type** $B3.

APW: *See* **Apple IIGS Programmer's Workshop.**

APW Linker: The linker supplied with APW.

ASCII: Acronym for *American Standard Code for Information Interchange.* A code in which the numbers from 0 to 127 stand for text characters. ASCII code is used for representing text inside a computer and for transmitting text between computers or between a computer and a peripheral device.

assembler: A program that produces **object files** (programs that contain machine-language code) from **source files** written in assembly language. Compare **compiler.**

AuxID: One of three fields in the **User ID,** a number that identifies each application.

backup bit: A bit in a file's **access byte** that tells backup programs whether the file has been altered since the last time it was backed up.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

bank-switched memory: On Apple II computers, that part of the **language card** memory in which two 4K-portions of memory share the same address range ($D000–$DFFF).

binary file: (1) A file whose data is to be interpreted in binary form. Machine-language programs and pictures are stored in binary files. Compare **text file.** (2) A file in **binary file format.**

binary file format: The ProDOS 8 loadable file format, consisting of one absolute memory image along with its destination address. A file in binary file format has ProDOS file type $06 and is referred to as a BIN file. The System Loader cannot load BIN files.

bit: A contraction of *binary digit.* The smallest unit of information that a computer can hold. The value of a bit (1 or 0) represents a simple two-way choice, such as yes or no or on or off.

bit map: A set of bits that represents the positions and states of a corresponding set of items. *See,* for example, **global page bit map** or **volume bit map.**

block: (1) A unit of data storage or transfer, typically 512 bytes. (2) A contiguous, page-aligned region of computer memory of arbitrary size, allocated by the Memory Manager. Also called a *memory block.*

block device: A device that transfers data to or from a computer in multiples of one block (512 bytes) of characters at a time. Disk drives are block devices.

boot: Another way to say *start up.* A computer boots by loading a program into memory from an external storage medium such as a disk. *Boot* is short for *bootstrap load.* Starting up is often accomplished by first loading a small program, which then reads a larger program into memory. The program is said to "pull itself up by its own bootstraps."

buffer: A region of memory where information can be stored by one program or device and then read at a different rate by another; for example, a ProDOS 16 I/O buffer.

Busy word: A firmware flag, consulted by the **Scheduler,** that protects system software that is not **reentrant** from being called while processing another call.

byte: A unit of information consisting of a sequence of 8 **bits.** A byte can take any value between 0 and 255 ($0 and $FF hexadecimal). The value can represent an instruction, number, character, or logical state.

call: (v.) To request the execution of a subroutine, function, or procedure. (n.) As in operating system calls, a request from the keyboard or from a program to execute a named function.

call block: The sequence of assembly-language instructions used to call ProDOS 16 or System Loader functions.

carry flag: A status bit in the microprocessor, used as an additional high-order bit with the accumulator bits in addition, subtraction, rotation, and shift operations.

character: Any symbol that has a widely understood meaning and thus can convey information. Most characters are represented in the computer as one-byte values.

character device: A device that transfers data to or from a computer as a stream of individual characters. Keyboards and printers are character devices.

close: To terminate access to an open file. When a file is closed, its updated version is written to disk and all resources it needed when open (such as its I/O buffer) are released. The file must be opened before it can be accessed again.

compact: To rearrange allocated memory blocks in order to increase the amount of contiguous unallocated (free) memory. The Memory Manager compacts memory when needed.

compiler: A program that produces **object files** (containing machine-language code) from **source files** written in a high-level language such as C. Compare **assembler.**

controlling program: A program that loads and runs other programs, without itself relinquishing control. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell, for example, is a controlling program.

creation date: An attribute of a ProDOS 16 file; it specifies the date on which the file was first created.

creation time: An attribute of a ProDOS 16 file; it specifies the time at which the file was first created.

current application: The application program currently loaded and running. Every application program is identified by a User ID number; the current application is defined as that application whose User ID is the present value of the `USERID` global variable.

data block: A 512-byte portion of a ProDOS 16 **standard file** that consists of whatever kind of information the file may contain.

default prefix: The pathname **prefix** attached by ProDOS 16 to a **partial pathname** when no **prefix number** is supplied by the application. The default prefix is equivalent to prefix number `0/`.

dereference: To substitute a **pointer** for a **memory handle.** When you dereference a memory block's handle, you access the block directly (through its **master pointer**) rather than indirectly (through its handle).

desk accessories: Small, special-purpose programs that are available to the user regardless of which application is running—such as the Control Panel, Calculator, Note Pad, and Alarm Clock.

desktop: The visual interface between the computer and the user. In computers that support the desktop concept, the desktop consists of a menu bar at the top of the screen, and a gray area in which applications are opened as windows. The desktop interface was first developed for the Macintosh computer.

device: A piece of equipment (hardware) used in conjunction with a computer and under the computer's control. Also called a **peripheral device** because such equipment is often physically separate from, but attached to, the computer.

device driver: A program that manages the transfer of information between a computer and a peripheral device.

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (one byte) address because its high address byte is always $00 and its middle address byte is the value of the 65C816 **direct register.** Co-resident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page.** The term *direct page* is often used informally to refer to any part of the lower portion of the **direct-page/stack space.**

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's **direct page** and **stack.** Initially, the 65C816 processor's **direct register** contains the base address of the space, and its **stack register** contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

directory file: One of the two principal categories of ProDOS 16 files. Directory files contain specifically formatted entries that contain the names and disk locations of other files. Compare **standard file.** Directory files are either **volume directories** or **subdirectories.**

disk device: *See* **block device.**

disk operating system: An operating system whose principal function is to manage files and communication with one or more disk drives. **DOS** and **ProDOS** are two families of Apple II disk operating systems.

dispose: To permanently deallocate a memory block. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge.**

dormant: Said of a program that is not being executed, but whose essential parts are all in the computer's memory. A dormant program may be quickly **restarted** because it need not be reloaded from disk.

DOS: An Apple II disk operating system. *DOS* is an acronym for *Disk Operating System.*

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare **static segment.**

e flag: A flag bit in the 65C816 that determines whether the processor is in **native mode** or **emulation mode.**

8-bit Apple II: *See* **standard Apple II.**

emulation mode: The 8-bit configuration of the 65C816 processor, in which it functions like a 6502 processor in all respects except clock speed.

EOF (end-of-file): The logical size of a ProDOS 16 file; it is the number of bytes that may be read from or written to the file.

error (or **error condition**)**:** The state of a computer after it has detected a fault in one or more commands sent to it.

error code: A number or other symbol representing a type of error.

event: A notification to an application of some occurrence (such as an interrupt generated by a keypress) that the application may want to respond to.

event-driven: A kind of program that responds to user inputs in real time by repeatedly testing for events posted by interrupt routines. An event-driven program does nothing until it detects an event such as a keypress.

external device: *See* **device.**

fatal error: An error serious enough that the computer must halt execution.

file: A named, ordered collection of information stored on a disk.

file control block (FCB): A data structure set up in memory by ProDOS 16 to keep track of all open files.

file entry or **file directory entry:** The part of a ProDOS 16 directory or subdirectory that describes and points to another file. The file so described is considered to be "in" or "under" that directory.

file level: *See* **system file level.**

filename: The string of characters that identifies a particular file within its directory. ProDOS 16 filenames may be up to 15 characters long. Compare **pathname.**

file system ID: A number describing the general category of operating system to which a file or volume belongs. The file system ID is an input to the ProDOS 16 `FORMAT` call, and a result from the `VOLUME` call.

file type: An attribute in a ProDOS 16 file's directory entry that characterizes the contents of the file and indicates how the file may be used. On disk, file types are stored as numbers; in a directory listing, they are often displayed as three-character mnemonic codes.

filing calls: Operating system calls that manipulate files. In ProDOS 16, filing calls are subdivided into *file housekeeping calls* and *file access calls*.

finder: A program that performs file and disk utilities (formatting, copying, renaming, and so on) and also starts applications at the request of the user.

firmware: Programs stored permanently in the computer's read-only memory (ROM). They can be executed at any time but cannot be modified or erased.

fixed: Not movable in memory once allocated. Also called *unmovable*. Program segments that must not be moved are placed in fixed memory blocks. Opposite of **movable.**

flush: To update an open file (write any updated information to disk) without closing it.

global page: Under ProDOS 8, 256 bytes of data at a fixed location in memory, containing useful system information (such as a list of active devices) available to any application.

global page bit map: A portion of the ProDOS 8 global page that keeps track of memory use in the computer. Applications under ProDOS 8 are responsible for marking and clearing parts of the bit map that correspond to memory they have allocated or freed.

guest file system: A file system, other than ProDOS 16's, whose files can be read by ProDOS 16.

handle: *See* **memory handle.**

hexadecimal: The base-16 system of numbers, using the ten digits 0 through 9 and the six letters A through F. Hexadecimal numbers can be converted easily and directly to binary form, because each hexadecimal digit corresponds to a sequence of four bits. In Apple manuals hexadecimal numbers are usually preceded by a dollar sign ($).

hierarchical file system: A method of organization in which disk files are grouped together within **directories** and **subdirectories.** In a hierarchical file system, a file is specified by its **pathname,** rather than by a single filename.

high-order: The most significant part of a numerical quantity. In normal representation, the high-order bit of a binary value is in the leftmost position; likewise, the high-order byte of a binary **word** or **long word** quantity consists of the leftmost eight bits.

Human Interface Guidelines: A set of software development guidelines developed by Apple Computer to support the **desktop** concept and to promote uniform user interfaces in Apple II and Macintosh applications.

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory.

index block: A 512-byte part of a ProDOS 16 **standard file** that consists entirely of pointers to other parts (**data blocks**) of the file.

initial load file: The first file of a program to be loaded into memory. It contains the program's main segment and the load file tables (Jump Table segment and Pathname segment) needed to load dynamic segments and run-time libraries.

initialization segment: A segment in an initial load file that is loaded and executed independently of the rest of the program. It is commonly executed first, to perform any initialization that the program may require.

input/output: The transfer of information between a computer's memory and peripheral **devices.**

interrupt: A temporary suspension in the execution of a program that allows the computer to perform some other task, typically in response to a signal from a device or source external to the computer.

interrupt handler: A program, associated with a particular external device, that executes whenever that device sends an interrupt signal to the computer. The interrupt handler performs its tasks during the interrupt, then returns control to the computer so it may resume program execution.

interrupt vector table: A table maintained in memory by ProDOS 16 that contains the addresses of all currently active (allocated) interrupt handlers.

INTERSEG record: A part of a relocation dictionary. It contains relocation information for external (intersegment) references.

I/O: *See* **input/output.**

JML: Unconditional Long Jump; a 65C816 assembly-language op code. It takes a 3-byte address operand. A JML can reach any address in the Apple IIGS memory space.

JMP: Unconditional Jump; a 6502 and 65C816 assembly-language op code. It takes a 2-byte address operand. A JMP can reach addresses only within a single 64K **bank** of the Apple IIGS memory space.

JSL: Long Jump to Subroutine; a 65C816 assembly-language op code. It takes a 3-byte address operand. A JSL can access any address in the Apple IIGS memory space.

JSR: Jump to Subroutine; a 6502 and 65C816 assembly-language op code. It takes a 2-byte address operand. A JSR can access addresses only within a single 64K **bank** of the Apple IIGS memory space.

Jump Table: A table constructed in memory by the System Loader from all Jump Table segments encountered during a load. The Jump Table contains all references to dynamic segments that may be called during execution of the program.

Jump Table directory: A master list in memory, containing pointers to all segments that make up the Jump Table.

Jump Table segment: A segment in a load file that contains all references to dynamic segments that may be called during execution of that load file. The Jump Table segment is created by the linker. In memory, the loader combines all Jump Table segments it encounters into the **Jump Table.**

K: Kilobyte. 1024 (2^{10}) bytes.

kernel: The central part of an operating system. ProDOS 16 is the kernel of the Apple IIGS operating system.

key block: The first block in any ProDOS 16 file.

kind: *See* **segment kind.**

language card: Memory with addresses between $D000 and $FFFF on any Apple II–family computer. It includes two RAM banks in the $D*xxx* space, called **bank-switched memory.** The *language card* was originally a peripheral card for the 48K Apple II or Apple II Plus that expanded its memory capacity to 64K and provided space for an additional dialect of BASIC.

level: *See* **system file level.**

library file: An object file containing program segments, each of which can be used in any number of programs. The linker can search through the library file for segments that have been referenced in the program source file.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

load file: The output of the linker. Load files contain memory images that the system loader can load into memory, together with **relocation dictionaries** that the loader uses to relocate references.

load segment: A segment in a load file.

lock: To prevent a memory block from being moved or purged. A block may be locked or unlocked by the Memory Manager, or by an application through a call to the System Loader.

long word: A double-length word. For the Apple IIGS, a long word is 32 bits (4 bytes) long.

low-order: The least significant part of a numerical quantity. In normal representation, the low-order bit of a binary number is in the rightmost position; likewise, the low-order byte of a binary **word** or **long word** quantity consists of the rightmost eight bits.

m flag: A flag in the 65C816 processor that determines whether the accumulator is 8 bits wide or 16 bits wide.

macro: A single predefined assembly-language pseudo-instruction that an assembler replaces with several actual instructions. Macros are almost like higher-level instructions that can be used inside assembly-language programs, making them easier to write.

MainID: One of three fields in the **User ID,** a number that identifies each application.

main segment: The first static segment (other than initialization segments) in the initial load file of a program. It is loaded at startup and never removed from memory until the program terminates.

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

Mark List: A table maintained in memory by the System Loader to help it perform relocation rapidly.

master index block: The key block in a ProDOS 16 **tree file,** the largest organization of a **standard file** that ProDOS 16 can support. The master index block consists solely of pointers to one or more **index blocks.**

master pointer: A pointer to a memory block; it is kept by the Memory Manager. Each allocated memory block has a master pointer, but the block is normally accessed through its memory handle (which points to the master pointer), rather than through the master pointer itself.

Mb: Megabyte. 1,048,576 (2^{20}) bytes.

memory block: *See* **block** (2).

memory handle: The identifying number of a particular block of memory. It is a pointer to the master pointer to the memory block. A handle rather than a simple pointer is needed to reference a movable memory block; that way the handle will always be the same though the value of the pointer may change as the block is moved around.

Memory Manager: A program in the Apple IIGS Toolbox that manages memory use. The Memory Manager keeps track of how much memory is available, and allocates memory **blocks** to hold program segments or data.

Memory Segment Table: A linked list in memory, created by the loader, that allows the loader to keep track of the segments that have been loaded into memory.

MLI: Machine Language Interface—the part of ProDOS 8 that processes operating system calls.

modification date: An attribute of a ProDOS 16 file; it specifies the date on which the content of the file was last changed.

modification time: An attribute of a ProDOS 16 file; it specifies the time at which the content of the file was last changed.

monitor: *See* **video monitor.**

Monitor program: A program built into the firmware of Apple II computers, used for directly inspecting or changing the contents of main memory and for operating the computer at the machine–language level.

move: To change the location of a memory block. The Memory Manager may move blocks to consolidate memory space.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block. Opposite of **fixed.** Only **position-independent** program segments may be in movable memory blocks. A block is made movable or fixed through Memory Manager calls.

native mode: The 16-bit operating configuration of the 65C816 processor.

newline mode: A file-reading mode in which each character read from the file is compared to a specified character (called the *newline character);* if there is a match, the read is terminated. Newline mode is typically used to read individual lines of text, with the newline character defined as a carriage return.

nibble: A unit of information consisting of one-half of a **byte,** or 4 **bits.** A nibble can take on any value between 0 and 15 ($0 and $F hexadecimal).

NIL: Pointing to a value of 0. A memory handle is NIL if the address it points to is filled with zeros. Handles to purged memory blocks are NIL.

null: Zero.

null prefix: A prefix of zero length (and therefore nonexistent).

object file: The output from an assembler or compiler, and the input to a linker. It contains machine-language instructions. Also called *object program* or *object code.* Compare **source file.**

object module format: The general format used in Apple IIGS object files, library files, and load files.

OMF file: Any file in object module format.

op code: *See* **operation code.**

operating system call: A request to execute a named operating system function; also, the name of the function itself. `OPEN`, `GET_FILE_INFO`, and `QUIT` are ProDOS 16 operating system calls

open: To allow access to a file. A file may not be read from or written to until it is open.

operand: The part of an assembly language instruction that follows the **operation code.** The operand is used as a value or an address, or to calculate a value or an address.

operating environment: The overall hardware and software setting within which a program runs. Also called *execution environment.*

operating system: A program that organizes the actions of the various parts of the computer and its peripheral devices. *See also* **disk operating system.**

operation code: The part of a machine-language instruction that specifies the operation to be performed. Often called *op code*.

page: (1) A portion of memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page-aligned*. (2) An area of main memory containing text or graphical information being displayed on the screen.

parameter: A value passed to or from a function or other routine.

parameter block: A set of contiguous memory locations, set up by a calling program to pass parameters to and receive results from an operating system function that it calls. Every call to ProDOS 16 must include a pointer to a properly constructed parameter block.

partial pathname: A portion of a **pathname** including the **filename** of the desired file but excluding the volume directory name (and possibly one or more of the subdirectories in the pathname). It is the part of a pathname following a **prefix**—a prefix and a partial pathname together constitute a full pathname. A partial pathname does not begin with a slash because it has no volume directory name.

patch: To replace one or more bytes in memory or in a file with other values. The address to which the program must jump to execute a subroutine is *patched* into memory at load time when a file is **relocated.**

pathname: The complete name by which a file is specified. It is a sequence of **filenames** separated by slashes, starting with the filename of the volume directory and following the path through any subdirectories that a program must follow to locate the file. A complete pathname always begins with a slash (/), because volume directory names always begin with a slash.

Pathname segment: segment in a load file that contains the cross-references between load files referenced by number (in the Jump Table segment) and their pathnames (listed in the file directory). The Pathname segment is created by the linker.

Pathname Table: A table constructed in memory from all individual Pathname segments encountered during loads. It contains the cross-references between load files referenced by number (in the Jump Table) and their pathnames (listed in the file directory).

pointer: An item of information consisting of the memory address of some other item. For example, the 65C816 stack register contains a pointer to the top of the stack.

position-independent: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be relocated.

prefix: A portion of a **pathname** starting with a volume name and ending with a subdirectory name. It is the part of a pathname before the **partial pathname**—a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

prefix number: A code used to represent a particular prefix. Under ProDOS 16, there are nine prefix numbers, each consisting of a number (or asterisk) followed by a slash: `0/`, `1/`,..., `8/`, and `*/`.

ProDOS: A family of disk operating systems developed for the Apple II family of computers. *ProDOS* stands for *Professional Disk Operating System*, and includes both ProDOS 8 and ProDOS 16.

ProDOS 8: A disk operating system developed for standard Apple II computers. It runs on 6502-series microprocessors. It also runs on the Apple IIGS when the 65C816 processor is in 6502 **emulation mode.**

ProDOS 16: A **disk operating system** developed for 65C816 **native mode** operation on the Apple IIGS. It is functionally similar to **ProDOS 8** but more powerful.

pull: To remove the top entry from a **stack,** moving the stack pointer to the entry below it. Synonymous with *pop*. Compare **push.**

purge: To temporarily deallocate a memory block. The Memory Manager purges a block by setting its master pointer to NIL (0). All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose.**

purge level: An attribute of a memory block that sets its priority for purging. A purge level of 0 means that the block is unpurgeable.

purgeable: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels,** or priorities for purging; these levels are set by Memory Manager calls.

push: To add an item to the top of a **stack,** moving the stack pointer to the next entry above the top. Compare **push.**

queue: A list in which entries are added at one end and removed at the other, causing entries to be removed in first-in, first-out (FIFO) order. Compare **stack.**

quit return stack: A stack maintained in memory by ProDOS 16. It contains a list of programs that have terminated but are scheduled to return when the presently executing program is finished.

random-access device: *See* **block device.**

record: A component of an load segment. All OMF file segments are composed of records, some of which are program code and some of which contain cross-reference or relocation information.

reentrant: Said of a routine that is able to accept a call while one or more previous calls to it are pending, without invalidating the previous calls. Under certain conditions, the **Scheduler** manages execution of programs that are not reentrant.

reference: (n) The name of a segment or entry point to a segment; same as *symbolic reference*. (v) To refer to a symbolic reference or to use one in an expression or as an address.

Reload segment: a load-file segment that is always loaded from the file at program startup, regardless of whether the rest of the program is loaded from file or restarted from memory. Reload segments contain initialization information, without which certain types of programs would not be restartable.

RELOC record: A part of a relocation dictionary that contains relocation information for local (within-segment) references.

relocate: To modify a file or segment at load time so that it will execute correctly at its current memory location. Relocation consists of **patching** the proper values onto address operands. The loader relocates load segments when it loads them into memory. *See also* **relocatable.**

relocatable: Characteristic of a load segment or other program code that includes no absolute addresses, and so can be relocated at load time. A relocatable segment can be static, dynamic, or position independent. It consists of a code **image** followed by a **relocation dictionary.** Compare **absolute.**

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image portion of the segment. *See* **relocate.**

restart: To reactivate a **dormant** program in the computer's memory. The System Loader can restart dormant programs if all their static segments are still in memory. If any critical part of a dormant program has been purged by the Memory Manager, the program must be reloaded from disk instead of restarted.

restartable: Said of a program that reinitializes its variables and makes no assumptions about machine state each time it gains control. Only restartable programs can be executed from a **dormant** state in memory.

result: An item of information returned to a calling program from a function. Compare **value.**

RTL: Return from subroutine Long; a 65C816 assembly-language instruction. It is used in conjunction with a `JSL` instruction.

RTS: Return from Subroutine; a 6502 and 65C816 assembly-language instruction. It is used in conjunction with a `JSR` instruction.

run-time library file: A load file containing program segments—each of which can be used in any number of programs—that the System Loader loads dynamically when they are needed.

sapling file: An organizational form of a ProDOS 16 **standard file**. A sapling file consists of a single **index block** and up to 256 **data blocks.**

Scheduler: A firmware program that manages requests to execute **interrupted** software that is not **reentrant**. If, for example, an interrupt handler needs to make ProDOS 16 calls, it must do so through the Scheduler because ProDOS 16 is not reentrant. Applications need not use the Scheduler because ProDOS 16 is not in an interrupted state when it processes applications' system calls.

sector: A division of a **track** on a disk. When a disk is formatted, its surface is divided into tracks and sectors.

seedling file: An organizational form of a ProDOS 16 **standard file.** A seedling file consists of a single **data block.**

segment: A component of an OMF file, consisting of a header and a body. In load files, each segment incorporates one or more subroutines.

segment kind: A numerical designation used to classify a segment in object module format. It is the value of the `KIND` field in the segment's header.

sequential-access device: *See* **character device.**

shadowing: The process whereby any changes made to one part of the Apple IIGS memory are automatically and simultaneously copied into another part. When shadowing is on, information written to bank $00 or $01 is automatically copied into equivalent locations in bank $E0 or $E1. Likewise, any changes to bank $E0 or $E1 are immediately reflected in bank $00 or $01.

shell application: A type of program that is launched from a **controlling program** and runs under its control. Shell applications are ProDOS 16 file type $B5.

soft switch: A location in memory that produces some specific effect whenever its contents are read or written.

source file: An ASCII file consisting of instructions written in a particular language, such as Pascal or assembly language. An assembler or compiler converts source files into **object files.**

sparse file: A variation of the organizational forms of ProDOS 16 **standard files.** A sparse file may be either a **sapling file** or a **tree file;** what makes it sparse is the fact that its logical size (defined by its **EOF**) is greater than its actual size on disk. This occurs when one or more **data blocks** contain nothing but zeros. Those data blocks are considered to be part of the file, but they are not actually allocated on disk until nonzero data is written to them.

special memory: On an Apple IIGS, all of banks $00 and $01, and all display memory in banks $E0 and $E1. So called because it is the memory directly accessed by **standard Apple II** programs running on the Apple IIGS.

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. The term *the stack* usually refers to the particular stack pointed to by the 65C816's **stack register.** Compare **queue.**

stack register: A hardware register in the 65C816 processor that contains the address of the top of the processor's **stack.**

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

standard file: One of the two principal categories of ProDOS 16 files. Standard files contain whatever data they were created to hold; they have no predefined internal format. Compare **directory file.**

start up: To get the system running. It involves loading system software from disk, and then loading and running an application. Also called *boot*.

static segment: A segment that is loaded only at program boot time, and is not unloaded during execution. Compare **dynamic segment.**

storage type: An attribute of a ProDOS 16 file that describes the file's organizational form (such as directory file, seedling file, or sapling file).

subdirectory: A ProDOS 16 directory file that is not the volume directory.

switcher: A controlling program that rapidly transfers execution among several applications.

system: A coordinated collection of interrelated and interacting parts organized to perform some function or achieve some purpose—for example, a computer system comprising a processor, keyboard, monitor, disk drive, and software.

system call: *See* **operating system call.**

system disk: A disk that contains the operating system and other system software needed to run applications.

System Failure Manager: A firmware program that processes fatal errors by displaying a message on the screen and halting execution.

system file: *See* **system program.**

system file level: A number between $00 and $FF associated with each open ProDOS 16 file. Every time a file is opened, the current value of the system file level is assigned to it. If the system file level is changed (by a `SET_LEVEL` call), all subsequently opened files will have the new level assigned to them. By manipulating the system file level, a controlling program can easily close or flush files opened by its subprograms.

System Loader: The program that manages the loading and relocation of load segments (programs) into the Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

system program: (1) A software component of a computer system that supports **application programs** by managing system resources such as memory and I/O devices. Also called *system software*. (2) Under ProDOS 8, a stand-alone and potentially self-booting application. A ProDOS 8 system program is of file type $FF; if it is self-booting, its filename has the extension .SYSTEM.

system software: The components of a computer system that support **application programs** by managing system resources such as memory and I/O devices.

tool: *See* **tool set.**

tool set: A group of related routines (usually in firmware), available to applications and system software, that perform necessary functions or provide programming convenience. The Memory Manager, the System Loader, and QuickDraw II are tool sets.

toolbox: A collection of built-in routines on the Apple IIGS that programs can call to perform many commonly-needed functions. Functions within the toolbox are grouped into **tool sets.**

track: One of a series of concentric circles on a disk. When a disk is formatted, its surface is divided into tracks and **sectors.**

tree file: An organizational form of a ProDOS 16 **standard file.** A tree file consists of a single **master index block,** up to 127 **index blocks,** and up to 32,512 **data blocks.**

TypeID: One of three fields in the **User ID,** a number that identifies each application.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides. The loader then modifies the Memory Segment Table to reflect the fact that the segment is no longer in memory.

unmovable: *See* **fixed.**

unpurgeable: Having a **purge level** of zero. the Memory Manager is not permitted to purge memory blocks whose purge level is zero.

User ID: An identification number that specifies the owner of every memory block allocated by the Memory Manager. User ID's are assigned by the User ID Manager.

User ID Manager: A tool set that is responsible for assigning User ID's to every block of memory allocated by the Memory Manager.

value: An item of information passed from a calling routine to a function. Compare **result.**

video monitor: A display device that receives video signals by direct connection only.

version: A number indicating the release edition of a particular piece of software. Version numbers for most system software (such as ProDOS 16 and the System Loader) are available through function calls.

volume: An object that stores data; the source or destination of information. A volume has a name and a volume directory with the same name; information on a volume is stored in **files.** Volumes typically reside in **devices;** a device such as a floppy disk drive may contain one of any number of volumes (disks).

volume bit map: A portion of every ProDOS 16-formatted disk that keeps track of free disk space.

volume control block (VCB): A data structure set up in memory by ProDOS 16 to keep track of all volumes/devices connected to the computer.

volume directory: A ProDOS 16 directory file that is the principal directory of a volume. It has the same name as the volume. The pathname of every file on the volume starts with the volume directory name.

volume name: The name by which a particular volume is identified. It is the same as the filename of the **volume directory** file.

word: A group of bits that is treated as a unit. For the Apple IIGS, a word is 16 bits (2 bytes) long.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

Index

ProDOS 16 Calls

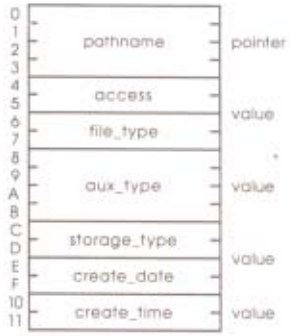
Call Block:

```
JSL     PRODOS
DC      I2 'CALLNUM'
DC      I4 'PARMBLOCK'
BCS     ERROR
```

PRODOS entry point
= $E1 00A8

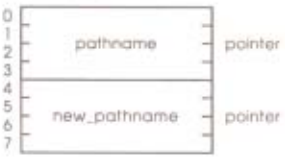
Each minor division in a parameter block diagram represents one byte

$01 CREATE

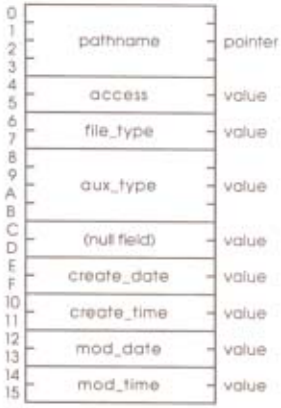


$02 DESTROY

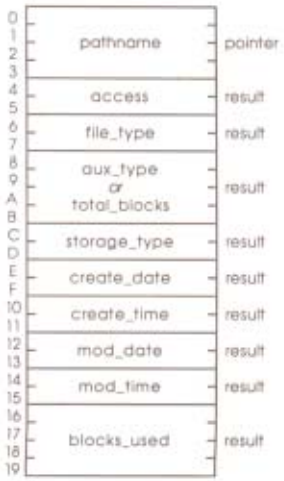
$04 CHANGE_PATH



$05 SET_FILE_INFO

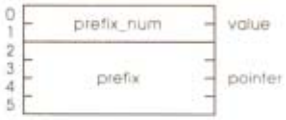


$06 GET_FILE_INFO

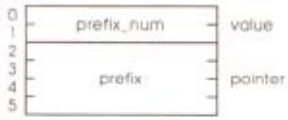


$08 VOLUME

$09 SET_PREFIX

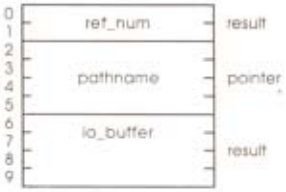


$0A GET_PREFIX

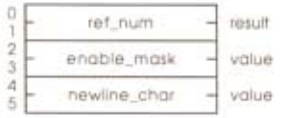


$0B CLEAR_BACKUP_BIT

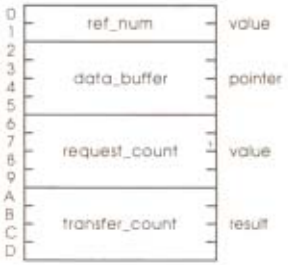
$10 OPEN



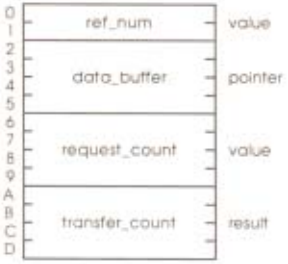
$11 NEWLINE



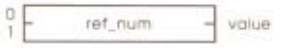
$12 READ



$13 WRITE

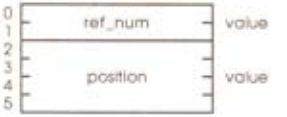


$14 CLOSE



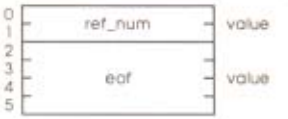
$15 FLUSH

$16 SET_MARK



$17 GET_MARK

$18 SET_EOF



P_BIT

pointer

value

pointer

value

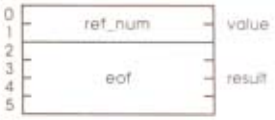
result

value

value

value

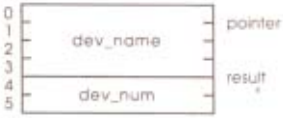
$19 GET_EOF



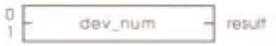
$1A SET_LEVEL

$1B GET_LEVEL

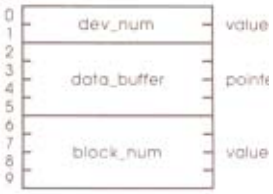
$20 GET_DEV_NUM



$21 GET_LAST_DEV

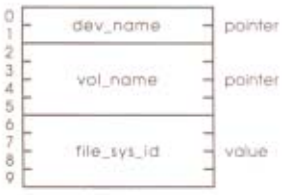


$22 READ_BLOCK



$23 WRITE_BLOCK

$24 FORMAT



$27 GET_NAME

$28 GET_BOOT_VOL

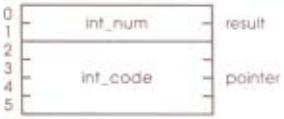
$29 QUIT

$2A GET_VERSION

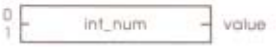
value

value

$31 ALLOC_INTERRUPT



$32 DEALLOC_INTERRUPT

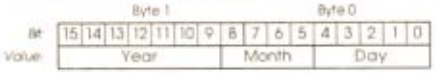


Fields

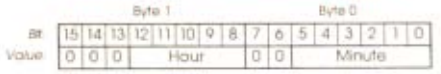
Access Byte

D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

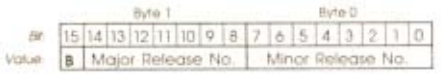
Creation/Modification Date



Creation/Modification Time

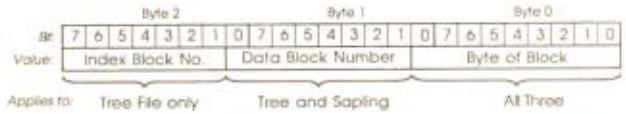


Version Word

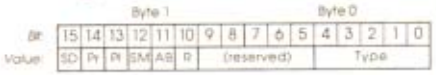


B = 0 for final releases
B = 1 for prototype releases

File Mark



Segment KIND



SD = 1: segment is dynamic
Pr = 1: segment is private
PI = 1: segment is position-independent
SM = 1: segment may not be in special memory
AB = 1: segment is absolute-bank
R = 1: segment is a Reload segment

Type	Description
$00	code segment
$01	data segment
$02	Jump Table segment
$04	Pathname segment
$08	library dictionary segment
$10	initialization segment
$12	direct-page/stack segment

User ID Word

TypeID	Description
$0	Memory Manager
$1	application
$2	controlling program
$3	ProDOS 8 and ProDOS 16
$4	tool set†
$5	desk accessory
$6	run-time library
$7	System Loader
$8	firmware/system function
$9	Tool Locator
$A-$F	(undefined)

† if TypeID = $04, these values of AuxID are reserved:

$1	Miscellaneous toolset file
$2	Scrap Manager file
$A	tool setup file

Error Codes, Files, and Records

ProDOS 16 Error Codes

Nonfatal:

$01	Invalid call number
$07	ProDOS is busy
$10	Device not found
$11	Invalid device request
$25	Interrupt vector table full
$27	I/O error
$28	No device connected
$2B	Write-protected
$2D	Invalid block address
$2E	Disk switched
$2F	Device not on line
$30 - $3F	Device-specific errors
$40	Invalid pathname/device name syntax
$42	FCB table full
$43	Invalid file reference number
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$48	Volume full
$49	Volume directory full
$4A	Version error (incompatible file format)
$4B	Unsupported (or incorrect) storage type
$4C	End-of-file encountered (out of data)
$4D	Position out of range
$4E	Access not allowed
$50	File is open
$51	Directory structure damaged
$52	Unsupported volume type
$53	Parameter out of range
$54	Out of memory
$55	VCB table full
$57	Duplicate volume
$58	Not a block device
$59	Invalid level
$5A	Block number out of range
$5B	Illegal pathname change
$5C	Not an executable file
$5D	Operating system/file system not available
$5E	Cannot deallocate /RAM
$5F	Return stack overflow
$60	Data unavailable

Fatal:

$01	Unclaimed interrupt
$0A	VCB unusable
$0B	FCB unusable
$0C	Block zero allocated illegally
$0D	Interrupt occurred while I/O shadowing off
$11	Wrong OS version

System Loader Error Codes

$1101	Not found
$1102	Incompatible OMF version
$1104	File is not a load file
$1105	Loader is busy
$1107	File version error
$1108	User ID error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

ProDOS 16 Filenames

A filename may contain any combination of

- capital letters A through Z
- numbers 0 through 9
- periods (.)

It must begin with a letter.

It may be up to 15 characters long.

A *volume name* follows the same rules, but also must have a preceding slash (/).

ProDOS 16 Pathnames

A pathname is a sequence of filenames separated by slashes

A full pathname begins with a slash (and a volume name)

A partial pathname begins with a filename or a prefix number

A full or partial pathname may be up to 64 characters long (including slashes)

ProDOS 16 Prefixes

0/	default prefix
1/	application subdirectory prefix
2/	system library prefix
3/ - 7/	(user-assigned)
*/	boot prefix

An application can change the pathname assigned to any prefix (except */)

ProDOS File Types

Type	Code	Description
$00		Uncategorized file (SOS/ProDOS)
$01	BAD	Bad block file
$02 †	PCD	Pascal code file
$03 †	PTX	Pascal text file
$04	TXT	ASCII text file (SOS/ProDOS)
$05 †	PDA	Pascal data file
$06	BIN	Gen. binary file (SOS/ProDOS 8)
$07 †	FNT	Font file
$08	FOT	Graphics screen file
$09 †	BA3	Business BASIC program file
$0A †	DA3	Business BASIC data file
$0B †	WPF	Word Processor file
$0C †	SOS	SOS system file
$0D–$0E †		(SOS reserved)
$0F	DIR	Directory file (SOS/ProDOS)
$10 †	RPD	RPS data file
$11 †	RPI	RPS index file
$12 †		AppleFile discard file
$13 †		AppleFile model file
$14 †		AppleFile report format file
$15 †		Screen Library file
$16–$18 †		(SOS reserved)
$19	ADB	AppleWorks Data Base file
$1A	AWP	AppleWorks Word Proc. file
$1B	ASP	AppleWorks Spreadsheet file
$1C–$AF		(reserved)
$B0	SRC	APW source file
$B1	OBJ	APW object file
$B2	LIB	APW library file
$B3	S16	ProDOS 16 application prog. file
$B4	RTL	APW run-time library file
$B5	EXE	ProDOS 16 shell application file
$B6		ProDOS 16 permanent init. file
$B7		ProDOS 16 temporary init. file
$B8		New desk accessory
$B9		Classic desk accessory
$BA		Tool set file
$BB–$BE		(reserved for ProDOS 16 load files)
$BF		ProDOS 16 document file
$C0–$EE		(reserved)
$EF	PAS	Pascal area on a partitioned disk
$F0	CMD	ProDOS 8 CI added command file
$F1–$F8		ProDOS 8 user defined files 1-8
$F9		(ProDOS 8 reserved)
$FA	INT	Integer BASIC program file
$FB	IVR	Integer BASIC variable file
$FC	BAS	Applesoft program file

†SOS (Apple III) only

ProDOS File Types (continued)

Type	Code	Description
$FD	VAR	Applesoft variables file
$FE	REL	Relocatable code file (EDASM)
$FF	SYS	ProDOS 8 system program file

Storage Type

$0	inactive file entry
$1	seedling file entry (EOF <= 256 bytes)
$2	sapling file entry (256 < EOF <= 128K bytes)
$3	tree file entry (128K < EOF < 16M bytes)
$4	Pascal operating system area on a partitioned disk
$D	subdirectory file entry
$E	subdirectory header
$F	volume directory header

File System ID

0	(reserved)
1	ProDOS/SOS
2	DOS 3.3
3	DOS 3.2, 3.1
4	Apple II Pascal
5	Macintosh
6	Macintosh (HFS)
7	LISA
8	Apple CP/M
9-255	(reserved)

Load File Records

Code	Name	Description
$E2	RELOC	intrasegment relocation record
$E3	INTERSEG	intersegment relocation record
$F1	DS	zero-fill record
$F2	LCONST	long-constant record (includes all code and data)
$F5	cRELOC	compressed intrasegment relocation record
$F6	cINTERSEG	compressed intersegment relocation record
$F7	SUPER	super-compressed relocation record
$00	END	end of the segment

System Loader Calls

1. Push result space (as shown on *Stack Before Call*) onto the stack
2. Push input parameters (in order shown on *Stack Before Call*) onto the stack
3. Execute call block:

```
LDX     #$11+FuncNum!8
JSL     Dispatcher
```

FuncNum = number of function being called

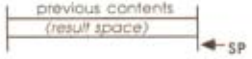
Dispatcher = Tool dispatcher (address = $E1 0000)

4. On completion, results will be in order shown on *Stack After Call*

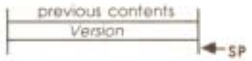
Each minor division in a stack diagram represents 1 word (2 bytes)

$04 Loader Version

Stack Before Call:

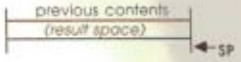


Stack After Call:

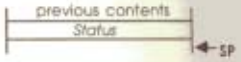


$06 Loader Status

Stack Before Call:

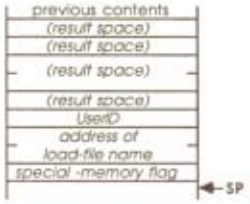


Stack After Call:

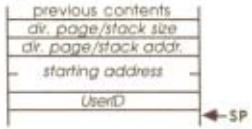


$09 Initial Load

Stack Before Call:

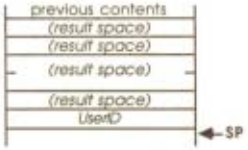


Stack After Call:

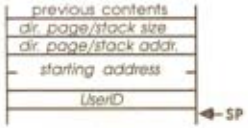


$0A Restart

Stack Before Call:

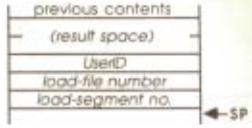


Stack After Call:

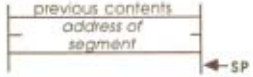


$0B Load Segment By No.

Stack Before Call:

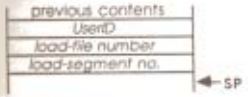


Stack After Call:

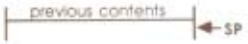


$0C Unload Seg. By No.

Stack Before Call:

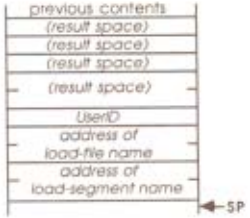


Stack After Call:

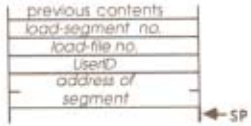


$0D Load Seg. By Name

Stack Before Call:

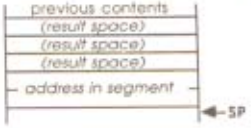


Stack After Call:

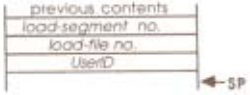


$0E Unload Segment

Stack Before Call:

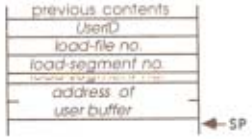


Stack After Call:



$0F Get Load Segment Info

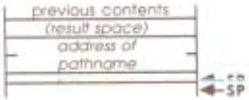
Stack Before Call:



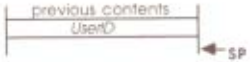
Stack After Call:

$10 Get User ID

Stack Before Call:

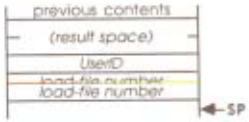


Stack After Call:

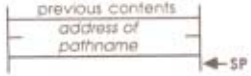


$11 Get Pathname

Stack Before Call:

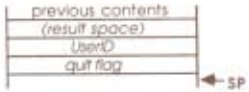


Stack After Call:

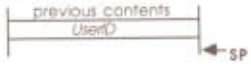


$12 User Shutdown

Stack Before Call:



Stack After Call:



The Official Publication from Apple Computer, Inc.

Written by the people at Apple Computer, Inc., this is the authoritative guide to the new Apple IIgs™ operating system. ProDOS®16 is an advanced ProDOS with extended file-management, device-management, and interrupt-handling capabilities. It can launch both standard Apple® II programs and new Apple IIgs programs.

This manual gives an overview of the operating system and a detailed documentation of its programming features. Specialized topics include:

- Using the QUIT call to pass execution from one application program to another.
- Switching rapidly among applications by making them dormant and restarting them.
- Writing controlling programs such as shells and switchers.
- Writing interrupt handlers.
- Working with multiple pathname prefixes.
- Converting applications based on ProDOS 8 to work with ProDOS 16.

The *Apple IIgs ProDOS 16 Reference* is organized into four parts:

- Part I shows how ProDOS 16 works and explains how it differs from its predecessor, ProDOS 8.
- Part II describes all ProDOS 16 commands (system calls) in detail.
- Part III documents the System Loader, a flexible programming tool that loads, unloads, and manipulates program segments in memory.
- Part IV consists of appendixes, a glossary, and an index. The appendixes describe the ProDOS 16 file structure, outline the history of Apple II operating systems, explain the ProDOS 16 Exerciser disk, list all ProDOS 16 and System Loader error codes, and provide additional System Loader technical information.

A quick-reference card bound into the manual tabulates ProDOS 16 and System Loader calls, errors, and data structures. The Exerciser disk in the back pocket allows you to practice making ProDOS 16 calls before actually writing an application program.

Written for assembly-language programmers and advanced users, the *Apple IIgs ProDOS 16 Reference* is indispensable for understanding and designing Apple IIgs application programs.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

030-3126-A
Printed in U.S.A.

Addison-Wesley Publishing Company, Inc.

ISBN 0-201-17754-4