Part V

Appendixes

Appendix A

ProDOS/16 File Organization

This appendix contains a detailed description of the way that ProDOS/16 stores files on disks. For most system program applications, the operating system insulates you from this level of detail. However, you must use this information if you want

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

This appendix first explains the organization of information on volumes. Next, it shows the storage of volume directories, directories, and the various stages of standard files. Finally it presents a set of diagrams that summarize all the material in this appendix. You can refer to these diagrams as you read the appendix. They will become your most valuable tool for working with file organization.

Format 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 a system program that uses ProDOS/16. This appendix discusses the organization of information on a volume in terms of logical blocks, numbered starting with zero, not tracks and sectors.

When the volume is formatted, information needed by ProDOS/16 is placed in specific logical blocks. A **loader program** is placed in blocks 0 and 1 of the volume. This program enables ProDOS/16 (or earlier versions of ProDOS) 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.

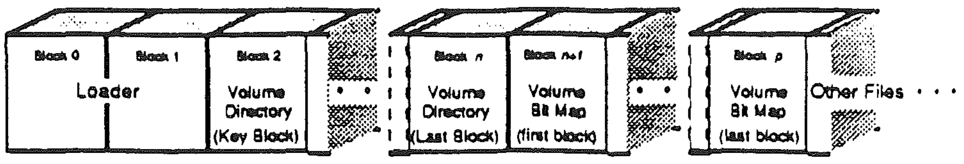

Block 0
Block 1
Loader
Block 2
Volume Directory (Key Block)
Block n
Volume Directory (Last Block)
Block n+1
Volume Bit Map (first block)
Block p
Volume Bit Map (last block)
Other Files

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 of Directory Files

The format 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 entry 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.

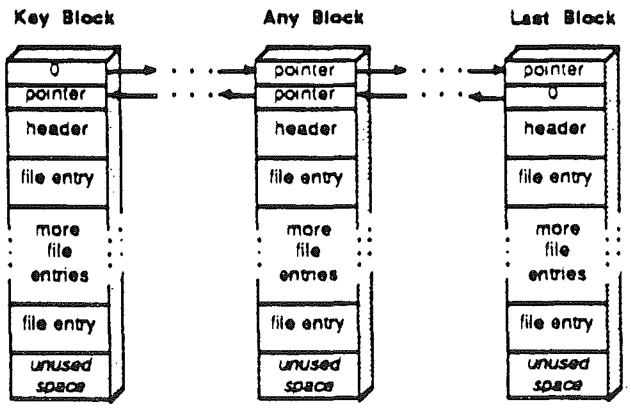


Figure A-2. Directory File Format

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

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 byte first, high 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.

> *By the Way:* All block pointers used by ProDOS/16 have the same format: low byte first, high byte last. However, because of Cortland's large memory size, all ProDOS/16 pointers except those in the disk file entries discussed here are *four* bytes long rather than two. See Chapter 2, "ProDOS/16 and Cortland 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 structure of a volume directory header. Following Figure A-3 is a description of each of its fields.

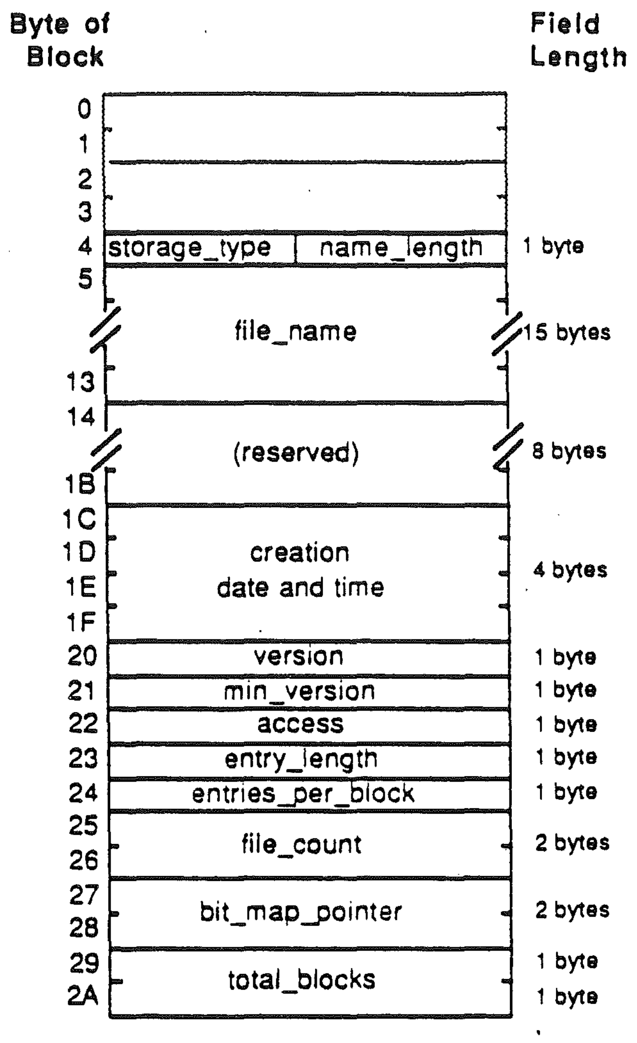


Figure A-3. The Volume Directory Header

storage_type and name_length (1 byte): Two four-bit fields are packed into this byte. A value of $F in the high four bits (the *storage_type*) identifies the current block as the key block of a volume directory file. The low four bits contain the length of the volume's name (see the *file_name* field, below). The 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.

creation date and **time**(4 bytes): The date and time at which this volume was initialized. The format of these bytes is described under "Header and Entry Fields," in this Appendix.

version (1 byte): The version number of ProDOS or ProDOS/16 under which this volume was initialized. This byte allows newer versions of ProDOS and ProDOS/16 to determine the format of the volume, and adjust their directory interpretation to conform to older volume formats. In ProDOS 1.0, version = 0.

min_version: Reserved for future use. In ProDOS 1.0, 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. *entry_length* = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. *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.

Its internal structure is quite similar to that of a volume directory header (only its last three fields are different). Fourteen fields are currently defined to be in a subdirectory header: they contain all the vital information about that subdirectory. Figure A-4 illustrates the structure of a subdirectory header. A description of all the fields in a subdirectory header follows figure A-4.

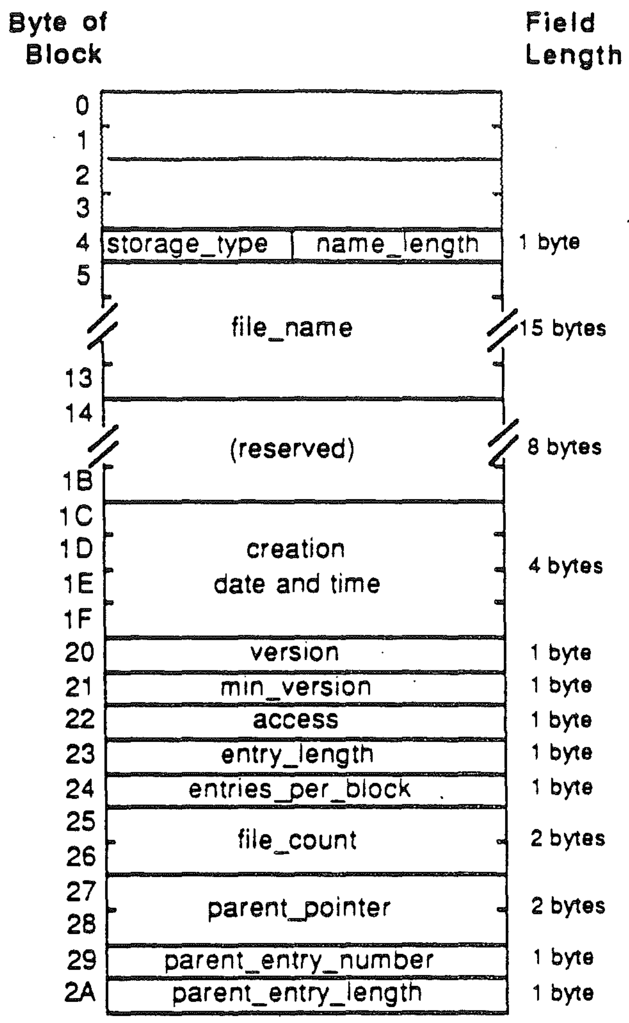


Figure A-4. The Subdirectory Header

storage_type and **name_length** (1 byte): Two four-bit fields are packed into this byte. A value of $E in the high four bits (the *storage_type*) identifies the current block as the key block of a subdirectory file. The low four bits contain the length of the subdirectory's name (see the *file_name* field, below). The *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.

creation date and **time** (4 bytes): The date and time at which this subdirectory was created. The format of these bytes is described under "Header and Entry Fields," in this Appendix.

version (1 byte): The version number of ProDOS or ProDOS/16 under which this subdirectory was created. This byte allows newer versions of ProDOS and ProDOS/16 to determine the format of the subdirectory, and to adjust their directory interpretations accordingly. ProDOS 1.0: *version* = 0.

min_version (1 byte): The minimum version number of ProDOS/16 that can access the information in this subdirectory. This byte allows older versions of ProDOS/16 to determine whether they can access newer subdirectories. *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. *entry_length* = $27.

entries_per_block (1 byte): The number of entries that are stored in each block of the directory file. *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 two-byte pointer is stored low byte first, high 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. *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.

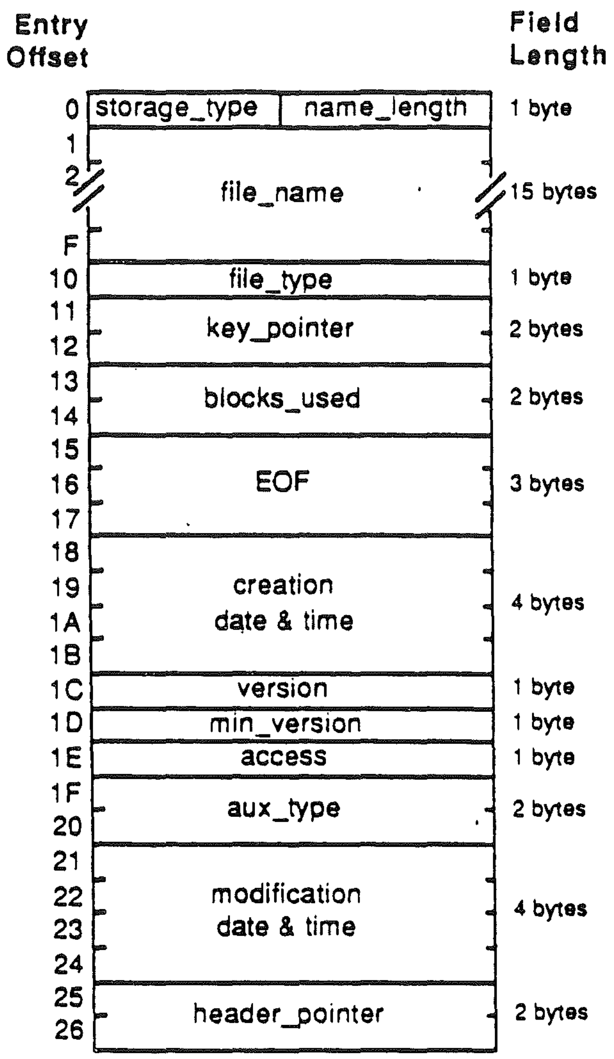


Figure A-5. The File Entry

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

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

Seedling, sapling, and tree files,are described under "Format of Standard Files," in this Appendix. The low four bits contain the length of the file's name (see the *file_name* field, below). The *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 structure of the file. Table A-1 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 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 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.

creation date and **time**: (4 bytes): The date and time at which the file pointed to by this entry was created. The format of these bytes is described under "Header and Entry Fields," below.

version (1 byte): The version number of ProDOS 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. In ProDOS 1.0, version = 0.

min_version (1 byte): The minimum version number of ProDOS or ProDOS/16 that can access the information in this file. This byte allows older versions of ProDOS and ProDOS/16 to determine whether they can access newer files. In ProDOS 1.0, 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." 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 a system program can store additional information about the internal format of a file. For example, the ProDOS 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.

last_mod (4 bytes): The date and time that the last CLOSE operation after a WRITE was performed on this file. The format of these bytes is described in Section B.4.2.2. 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 two-byte pointer is stored low byte first, high 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:

- the length of each entry in the directory (*entry_length*)
- the number of entries in each block of the directory (*entries_per_block*)
- the 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 example below). 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 the example below, 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, a system program can scan through the entries in the buffer, using a pointer to the beginning of the current entry *EntryPointer*, 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 of Standard Files

Each active entry in a directory file points to the key block (the first block) of a 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 structure of the three stages of standard file: seedling, sapling, and tree. These are the files in which all programs and data are stored.

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 1.0 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
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:

Data Block 0

	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:

Index Block 0
Data Block 0
Data Block 1

	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:

Master Index Block
Index Block 0
Index Block 1
Data Block 0
Data Block 255
Data Block 256

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 becauseEOF 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 Section B.3.6 on sparse files.

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

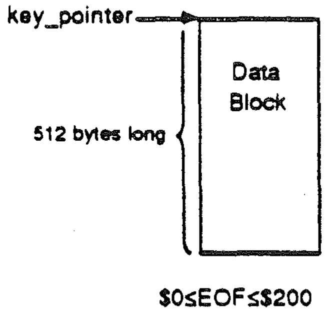


Figure A-6. Organization of a Seedling File

The file is called a seedling file because, 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. See Figure A-7.

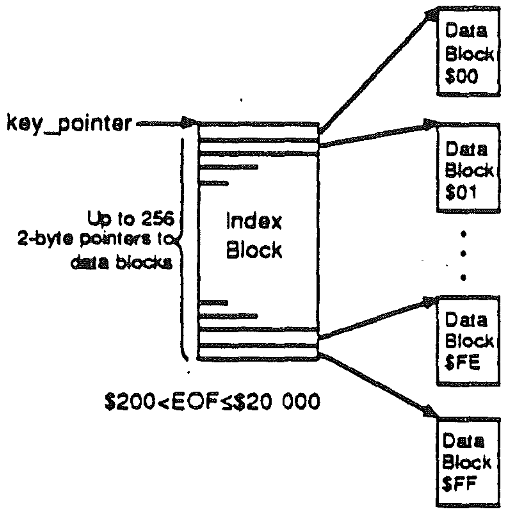


Figure A-7. Organization of a Sapling File

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 16M bytes ($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 structure of a tree file is shown in Figure A-8.

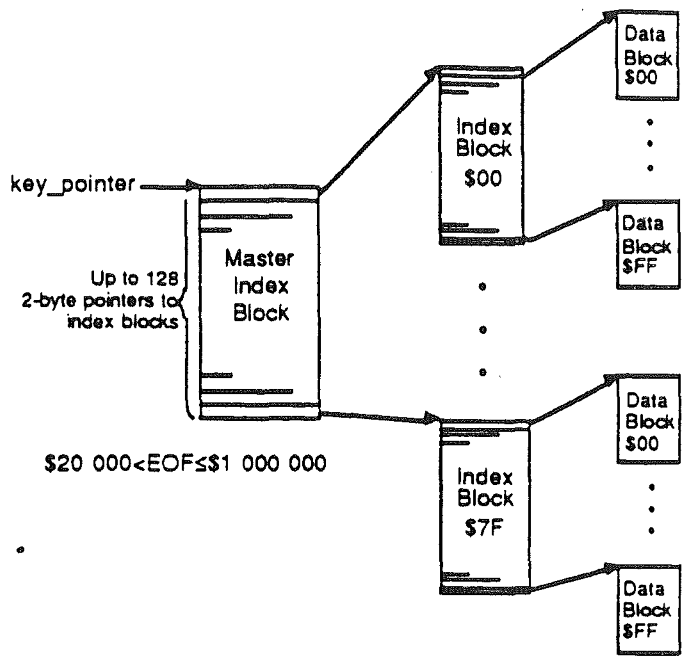


Figure A-8. Organization of a Tree File

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

A system program or 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. 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).

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.

If you now set the **EOF** to $4000 and close the file, you have a 16K file that takes up three blocks of space on the volume: two data blocks and an index block. You can read 16384 bytes of data from the file, but all the bytes before $0565 and after $0568 are nulls. Figure A-9 shows how the file is organized.

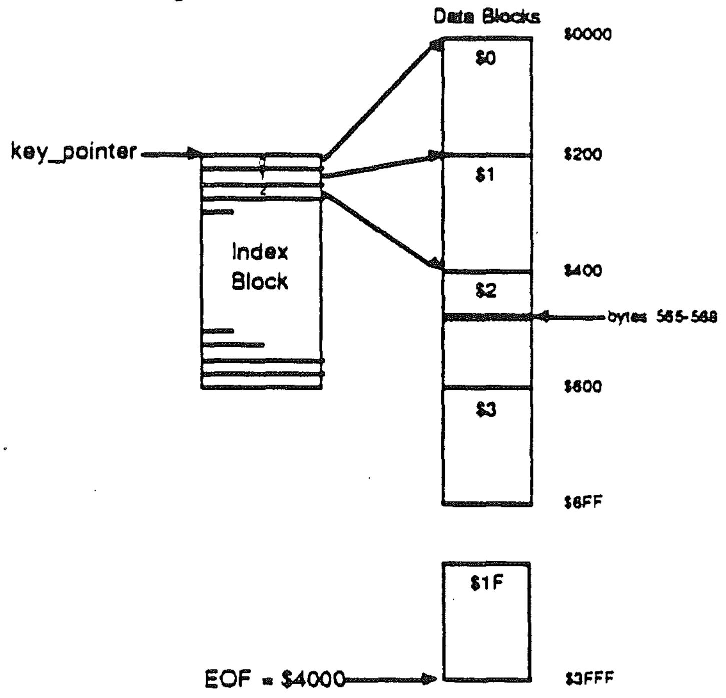


Figure A-9. An Example of Sparse File Organization

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.

On the other hand, if you CREATE a file with an EOF of $4000 (making it 16K bytes, or 32 blocks, long), ProDOS/16 allocates an index block and 32 data blocks for a sapling file, and fills the data blocks with nulls.

> *By the Way:* 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

The algorithm for finding a specific byte within a standard file is given below.

The 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 seven bits of the MARK determine the number (0 to 127) of the index block that points to the byte. The value of the seven bits indicates 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 indicated by the value of these seven bits plus 256.

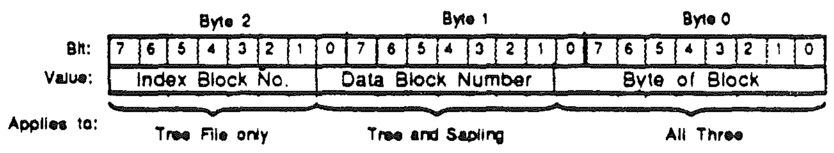


Figure A-10. MARK Format

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. This 8-bit value indicates 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 this offset plus 256.

For tree, sapling, and seedling files, the low nine bits of the MARK are the absolute position 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.

$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

The *name_length*, the low-order four bits of the first byte, specifies the number of characters in the *file_name* field.

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.

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

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 analysis.

The Access Attribute

The access attribute field (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.

Bit:	7	6	5	4	3	2	1	0
Value:	D	RN	B	reserved			W	R

where

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

Figure A-12. The Access Attribute Field

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 system programs to guarantee file compatibility from one system program to the next. The currently recognized values of this byte are shown in Table A-1.

Table A-1. Apple II File Types

File Type	Preferred Use
$00	Typeless file (SOS and ProDOS)
$01	Bad block file
$02 †	Pascal code file
$03 †	Pascal text file
$04	ASCII text file (SOS and ProDOS)
$05 †	Pascal data file
$06	General binary file (SOS and ProDOS)
$07 †	Font file
$08	Graphics screen file
$09 †	Business BASIC program file
$0A †	Business BASIC data file
$0B †	Word Processor file
$0C †	SOS system file
$0D,$0E †	SOS reserved
$0F	Directory file (SOS and ProDOS)
$10 †	RPS data file
$11 †	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	AppleWorks Data Base file
$1A	AppleWorks Word Proc. file
$1B	AppleWorks Spreadsheet file
$1C-$EE	Reserved
$B3	ProDOS/16 system file
$B4	ProDOS/16 run-time library file
$B5	ProDOS/16 shell load file
$B6	ProDOS/16 startup load file
$EF	Pascal area on a partitioned disk
$F0	ProDOS CI added command file

\$F1-\$F8	ProDOS user defined files 1-8
\$F9	ProDOS reserved
\$FA	Integer BASIC program file
\$FB	Integer BASIC variable file
\$FC	Applesoft program file
\$FD	Applesoft variables file
\$FE	Relocatable code file (EDASM)
\$FF	ProDOS system file

†apply to Apple III only

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, and Pascal), as well as one developed for the Apple III (SOS). 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.

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. The final section describes the changes made for version 1.2 of ProDOS.

History

DOS

DOS stands for *Disk Operating System*. It is Apple's first operating system; before DOS, the firmware **System Monitor** 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 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.

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 (below), which makes it impractical on computers other than the Apple III.

ProDOS

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

ProDOS 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 developed specifically for the Apple IIe and IIc is ProDOS 1.1.1. An even more recent version, developed for the Cortland but compatible with the IIe and IIc, is ProDOS 1.2

ProDOS/16

ProDOS/16 is an extended version of ProDOS, developed specifically for the Cortland (it will not run on other Apple II's). The "16" refers to the 16-bit internal registers in the Cortland's 65816 microprocessor.

ProDOS/16 permits access to Cortland's entire 16 Mbyte addressable memory space (ProDOS is restricted to addressing 64K bytes) and it has more "SOS-like" features than ProDOS 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 an interim system, consisting of a ProDOS 1.2 core surrounded by a "ProDOS/16-like" user interface. Version 2.0 is the first complete implementation of the ProDOS/16 design.

Pascal

The Pascal operating system for the Apple II and Apple III was modified and extended from UCSD Pascal, developed at the University of California at San Diego. The latest version, written for the Apple IIe/IIc, is Pascal 1.3.

File Compatibility

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

ProDOS: ProDOS/16 and ProDOS have identical file system organizations and support the same file types, with these exceptions:

- ProDOS/16 does not recognize ProDOS system files (type $FF) or binary files (type $06)[***true??]
- ProDOS does not recognize the file types $B3, $B4, $B5, $B6; these file types are specific to ProDOS/16.

SOS: The SOS file types that are recognized by ProDOS/16 are directory files, text files, and binary files[***?]. These three types are adequate for transferring programs and data between SOS and ProDOS/16.

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

Pascal: Pascal does not have a hierarchical file system. ProDOS/16 cannot read Pascal files.

Reading DOS 3.3 Disk Files

Both DOS 3.3 and ProDOS 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

Input/Output

ProDOS/16 can perform I/O operations on disk files (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).

ProDOS: Like ProDOS/16, ProDOS 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, or ProDOS/16.

Pascal: Pascal provides access to both block devices and character devices, through their *volume* names.

Filing Calls

SOS, ProDOS, 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 and SOS (ProDOS/16 calls have a completely different numbering system from either ProDOS or SOS). Some differences are these:

- With ProDOS and ProDOS/16, unlike SOS, you don't specify the file size when creating a file. File sizes are automatically extended when necessary.

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

- The SOS VOLUME call corresponds to the ProDOS ON_LINE 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/16 VOLUME call is functionally identical to the SOS VOLUME call.

- The SOS calls SET_MARK and SET_EOF can use a displacement from the current position in the file. ProDOS and ProDOS/16 accept only absolute positions in the file for these 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, a Cortland ROM-based Tool. 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 Cortland Memory Manager is similar to the SOS memory manager, except that it is more sophisticated and is not considered part of the operating system.

ProDOS: A ProDOS system program is responsible for its own memory management. It must find free memory, and then allocate it by marking it off in the ProDOS Global Page's memory bit map. ProDOS 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 calls).

SOS: SOS has a fairly sophisticated memory manager that is part of the operating system itself. A system program or application requests memory from SOS, either by location or by the anmount 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:

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. Interrupt routines in ProDOS/16 have no priority; ProDOS/16 must poll them in succession until one of them claims the interrupt.

ProDOS: ProDOS 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 do not.

Revised ProDOS for the Cortland

Both ProDOS and ProDOS/16 operating systems are delivered with the Cortland computer. Cortland ProDOS (version 1.2) has some minor revisions, necessary to make it compatible with the Cortland hardware configuration.

1. The system clock:

   For Apple II computers, the only supported system clock is the ThunderClock™, an option on an accessory board. Cortland has a built-in system clock, and ProDOS has been modified to support that clock. At boot time, ProDOS senses whether it is on a Cortland; if so, it replaces the ThunderClock routine with the built-in clock routine.

2. QUIT mechanism:

***

3. Cold and warm start:

   A ProDOS cold start on Cortland is the same as on other Apple II's. A ProDOS warm start is performed by taking the proper system file name from a prearranged memory location, then loading and running that program. [***not completely defined yet?]

4. NEWLINE bug:

   A minor error in the way ProDOS handles NEWLINE mode in reading and writing has been corrected. See ***.

5. Zeroing index blocks:

   A previous revision to ProDOS causes all of a file's index blocks to be zeroed when the file is destroyed. That revision has [***?]been removed.

For more details on Cortland ProDOS, see *ProDOS 1.2 Delta Guide* or the *ProDOS Technical Reference Manual.*

Appendix C

The ProDOS/16 Exerciser

(version 0.0)

information not yet available

Appendix D

System Loader Technical Data

This appendix assembles some specific technical information 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 Cortland **Object module Format**. Object Module format is described in detail in *Cortland Programmer's Worrkshop*.

File Types

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

File type	Description
$B0	Source file (*aux_type* defines language)
$B1	Object file
$B2	Library file
$B3	Load file
$B4	Run-time library file
$B5	Shell load file
$B6	Initialization system file

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. Each bit in the KIND field describes some attribute of the segment; different combinations of these attributes yield different values for the segment kind. Some specific values are

Segment Kind	Description
$02	Segment Jump Table
$04	Pathname Table
$08	Library Segment Dictionary
$10	Initialization segment

Record Codes

Load segments, like all OMF segments, are made up of **records**. For a complete list of record types, see*Cortland Programmer's Worrkshop*. 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)
$F2	LCONST	long-constant record (the actual code and data for each segment)

If the loader encounters any other type of record in a load segment, it will not load the segment.

Load File Numbers

Load files processed by the linker at any one time are numbered consecutively from one. Load file 1 is called the **initial load file**. It must fulfill these requirements:

- It must be static.
- If there is an Initialization segment, it must [***???] be in load file 1.
- if there is a segment jump table, it must be in load file 1.
- If there is a Pathname table, it must [***???]be in load file 1.

Load Segment Numbers

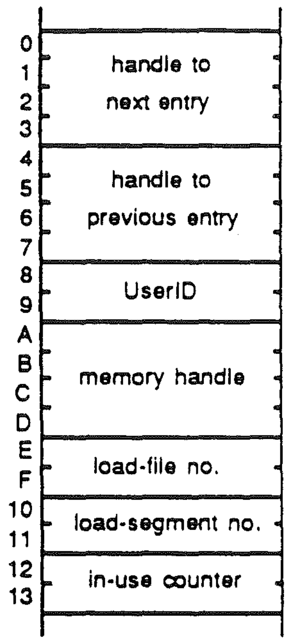
In each load file created by the linker, segments are numbered consecutively by their position in the load file, starting at 1. The loader determines a segment's number by counting up to it from the beginning of the load file. As a check, it also looks at the segment number in the segment's header.

Load Data Table Formats

The tables diagrammed here are described in Chapter 16. The width of each diagram represents one byte in memory; numbers down the left side of the diagram represent byte offsets (in hexadecimal) from the base address of the entry.

Memory Segment Table

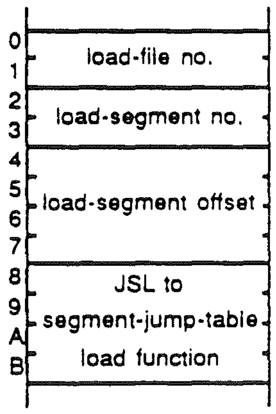
Each *entry* in the memory segment table looks like this:



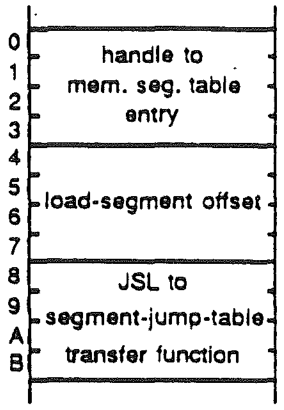
Segment Jump Table

Entries in the Segment Jump Table are in one of two states.

1. "Unloaded" Segment Jump Table entry:

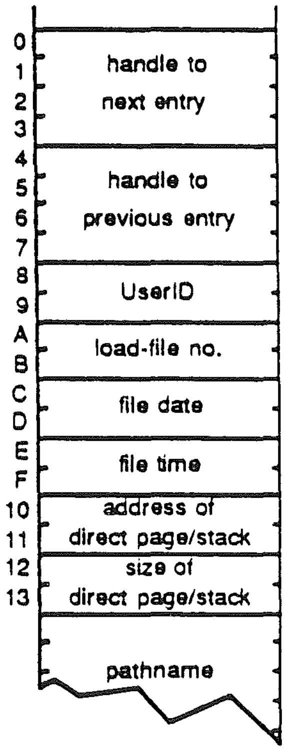


2. "Loaded" Segment Jump Table Entry:


0
1
2
3
4
5
6
7
8
9
A
B
handle to mem. seg. table entry
load-segment offset
JSL to segment-jump-table transfer function

Pathname Table

Each *entry* in the Pathname Table looks like this:



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 Cortland tool dispatcher, at the bottom of bank $E1 (address $E1 00 00). Although the System Loader maintains memory space with a table of loader functions n bank $00 and other space in bank $E0, locations in those areas are not supported. Pleas make all System Loader calls with a JSL to $E1 00 00.

The following variables are of global significance. They are defined at the system level, meaning that any system program or 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 system programs only, and should not be used by applications.

SEGTBL	Absolute address of memory segment table
JMPTBL	Absolute address of segment jump table
PATHTBL	Absolute address of pathname table
RETSTK	Absolute address of current application's return address stack
USERID	UserID of current application

UserID Format

The UserID Manager is discussed in Chapter 5, and fully explained in *Cortland Toolbox Reference: Part I*. Only the format of the UserID parameter needed by the System Loader and ProDOS/16 is shown here.

There is a 2-byte UserID associated with every allocated memory segment. It is divided into three fields: Main ID, Aux ID, and Type ID. The Main ID is the unique number assigned to the owner of the segment by the UserID Manager; every allocated segment has a nonzero value in its Main ID field. The Aux ID is a user-assignable identification; it is ignored by the System Loader, ProDOS/16, and the UserID Manager. The Type ID gives the general class of software to which the segment belongs.

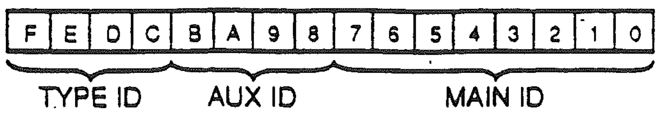


Figure D-1. UserID Format

The Main ID can-have any value from $01 to $FF (0 is reserved).

The Aux ID can have any value from $00 to $0F.

Type ID values are defined as follows:

0	Memory Manager
1	Application
2	Controlliing Program
3	ProDOS and ProDOS/16
4	Toolset
5	Desk accessory
6	Run-time library
7	System Loader
8-F	(undefined)

Appendix E

ASCII Tables

ASCII	Dec.	Hex.	Binary	ASCII	Dec.	Hex.	Binary
NUL	0	00	00000000	SP	32	20	00100000
@	64	40	01000000	DEL	127	7F	01111111

Glossary

absolute code: Program code that must be loaded at a specific address in memory, and never moved.

access byte:

application program (or **application**): A program that performs a specific task useful to the computer user, such as word processing, data base management, or graphics. Compare with **controlling program**.

assembler: A program that produces object files from source files written in assembly language.

backup bit:

bank:

binary output file: A file of absolute code that is ProDOS file type $06. The system loader will not load binary files.

bitmap:

block device:

character device:

code segment: A segment that contains program.code. Code segments are provided for programs that differentiate between code and data segments. See the "Segment Types" section in the *Object Module Format Preliminary Notes*.

compact:

compiler: A program that produces object modules from source files written in a high-level language such as Pascal.

controlling program: the program that initially requests the system loader to load and run other programs, and that is responsible for shutting down those programs when they are finished. A finder is an example of a controlling program.

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

data code: Code that consists primarily of data.

data segment: A segment that contains data code. Data segments are provided for programs that differentiate between code and data segments; see the" Segment Types" section in the *Object Module Format Preliminary Notes*.

device:

device driver:

device information block (DIB): a memory table, maintained by the operating system, which contains pertinent information on all active devices. DIB's will be a feature of ProDOS/16 (version 2); ProDOS and ProDOS/16 (version 1) do not build or use DIB's.

direct page:

directory file:

dispose: To permanently disallocate a memory segment. The memory manager disposes of a memory segment by removing its master pointer. Any handle to that pointer will then be invalid. Compare with **purge**.

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

emulation mode:

EOF (end-of-file):

error:

external device: See **device**.

failure: See **system failure**.

fatal error:

file control block (FCB):

file name:

file type:

file number cross-reference: The part of the pathname table that contains load-file numbers and pointers to their corresponding pathnames.

Finder:

global page:

global symbol: A label in a code segment that is either the name of the segment or an entry point to it. Global symbols may be referenced by other segments. Compare with **local symbol**.

handle: See **memory handle**

Human Interface Guidelines:

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:

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 (segment jump table and pathname table) needed to load dynamic segments.

initialization segment: A segment in an initial load file that is loaded and executed first, to perform any initialization that the program may require

interrupt:

interrupt handler:

interrupt vector table:

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

kernel:

key block:

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: The 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.

load segment: A segment in a load file.

local symbol: A label defined only within an individual segment. Other segments cannot access the label. Compare with **Global Symbol.**

lock: To prevent a memory segment from being moved or purged. A segment may be locked or unlocked by the memory manager, or by an application through a call to the system loader.

main segment: The first segment in the initial load file of a program (unless the file also has an initialization segment). It is loaded first and never removed from memory until the program terminates.

MARK:

master index block:

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

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

Memory Manager:

memory segment: A contiguous, page-aligned region of memory, under control of the memory manager. Each memory segment contains a single load segment that the system loader has loaded into memory.

MLI:

move: To change the location of a segment in memory. The memory manager may move segments to consolidate memory space.

movable: A memory segment attribute, indicating that the memory manager is free to move the segment . A segment is made movable or unmovable through memory manager calls.

native mode:

object file: The output from an assembler or compiler, and the input to the linker.

object module: An object file in Object Module Format.

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

OMF File: Any file in object module format.

operating system:

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 the system loader must include a pointer to a properly-constructed parameter block

pathname:

pathname list: The part of the pathname table that contains the file pathnames.

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

pointer:

position-independent code: 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:

prefix designator:

program code: Code that consists primarily of instructions.

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

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

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

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

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

relocate: The process of modifying a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded.

relocatable segment: A load segment that can be loaded into any part of memory. A relocatable load segment contains a code **image** followed by a **relocation dictionary**.

relocation dictionary: A part of a relocatable load segment that contains relocation information necessary to modify the code-image immediately preceding it. When the code-image part of the segment is loaded into memory, the relocation dictionary is processed to recalculate the values of location-dependent addresses and operands.

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:

Scheduler:

sector:

seedling file:

segment: An individual component of an OMF file. Each file contains one or more segments.

segment jump table: A segment in a load file that contains all references to dynamic segments that will be called during execution of the program. The segment jump table is created by the linker.

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

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:

standard file:

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

storage type:

system level:

System Loader:

system program:

track:

tree-structured file or tree file:

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 segment 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.

UserID:

UserID Manager:

version:

volume:

volume bitmap:

volume control block (VCB):

volume directory:

zero page: