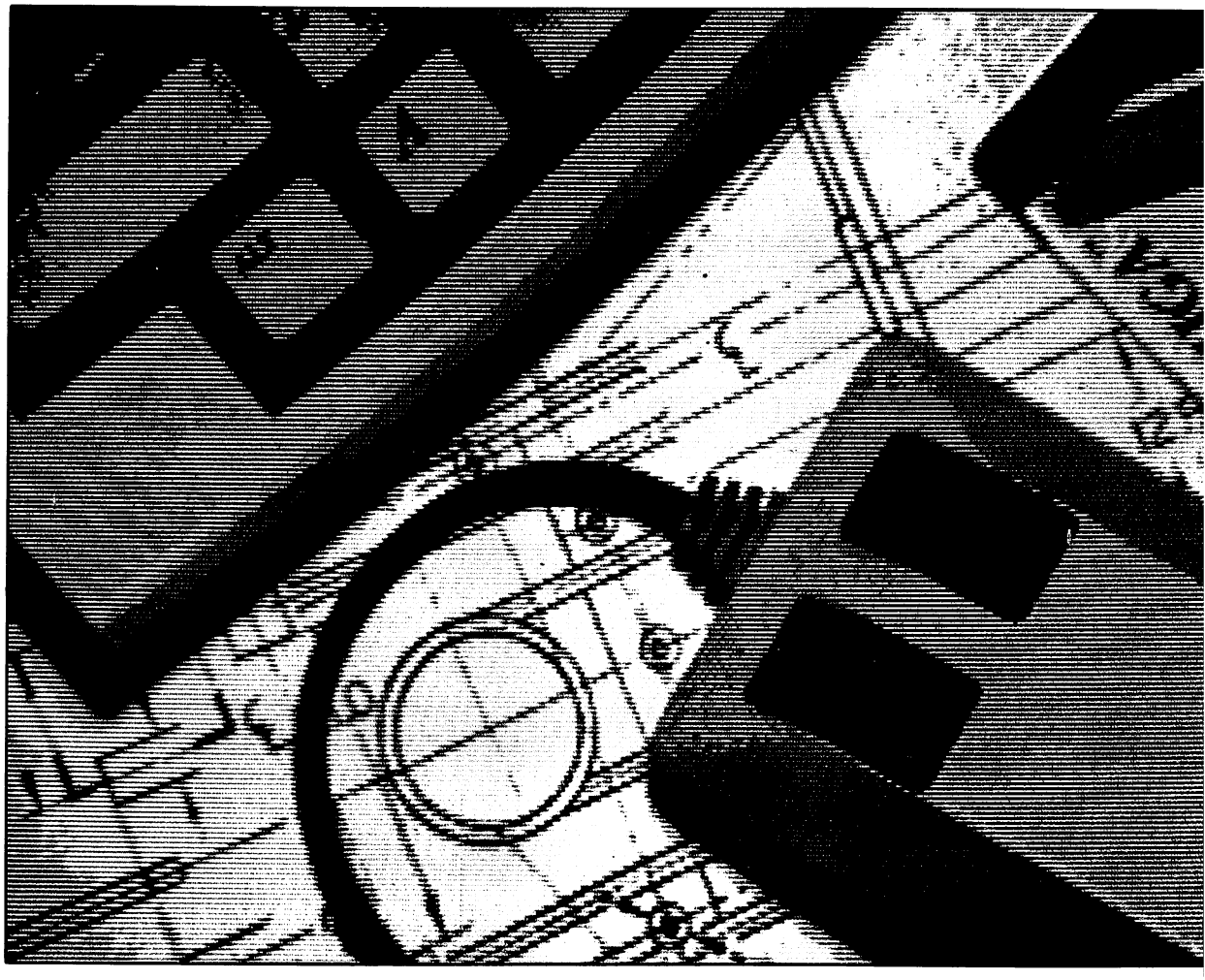
AMIGA™



AmigaDOS Technical Reference Manual

AmigaDOS Technical Reference Manual

Table of Contents

1.1 AmigaDOS File Structure

The AmigaDOS file handler uses a disk that is formatted with blocks of equal size. It provides an indefinitely deep hierarchy of directories, where each directory may contain other directories and files, or just files. The structure is a pure tree - that is, loops are not allowed.

There is sufficient redundancy in the mechanism to allow you to patch together most, if not all, of the contents of a disk after a serious hardware error, for example. To patch the contents of a disk, you use the DISKED command. For further details on the syntax of DISKED, see section 1.2, "DISKED - The Disk Editor," later in this chapter. Before you can patch together the contents a disk, you must understand the layout. The subsections below describe the layout of disk pages.

1.1.1 Root Block

The root of the tree is the Root Block, which is at a fixed place on the disk. The root is like any other directory, except that it has no parent, and its secondary type is different. AmigaDOS stores the name of the disk volume in the name field of the root block.

Each filing system block contains a checksum, where the sum (ignoring overflow) of all the words in the block is zero.

The figure on the following page describes the layout of the root block.

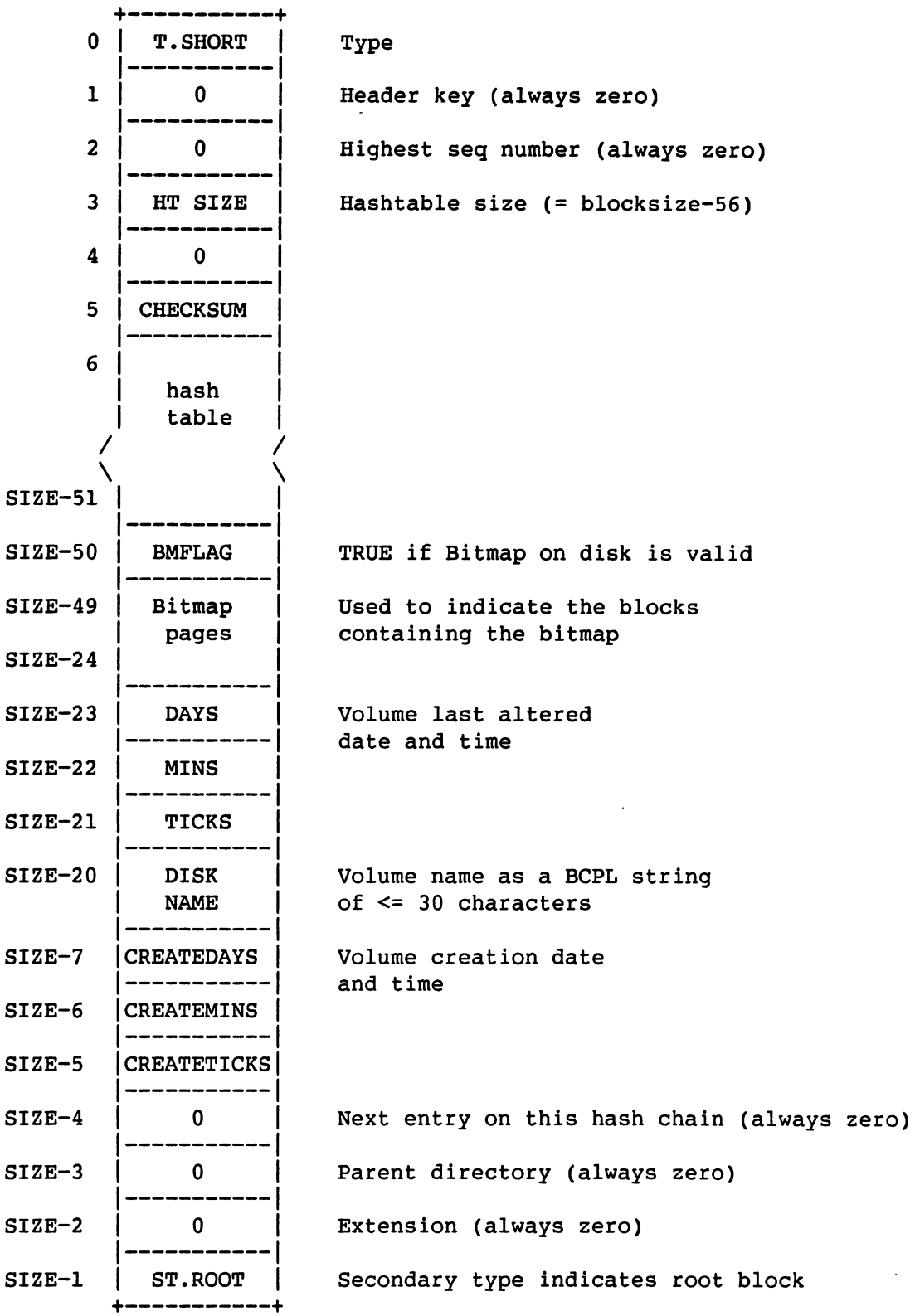


Figure 1-A: Root Block

1.1.2 User Directory Blocks

The following figure describes the layout of the contents of a user directory block.

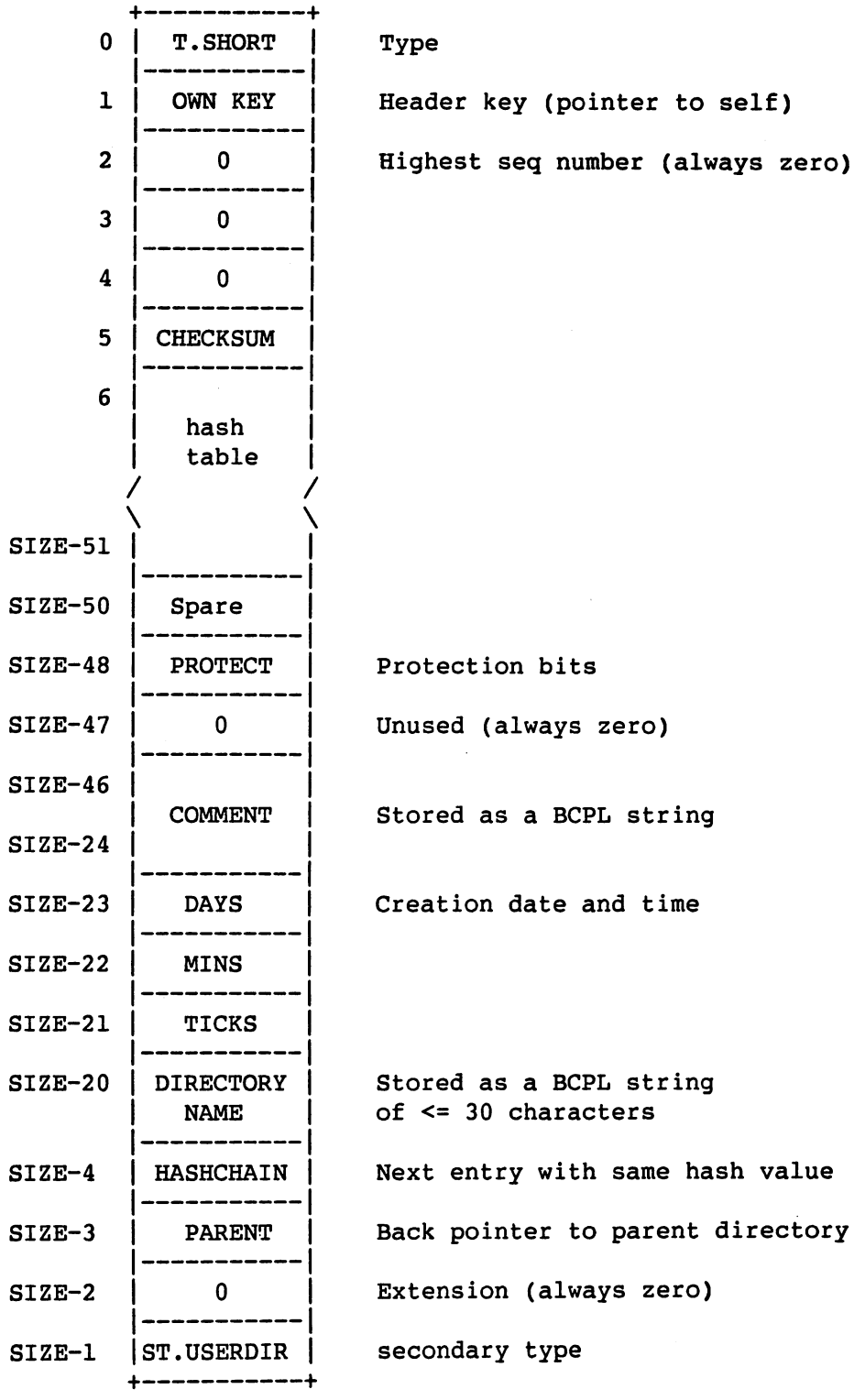


Figure 1-B: User Directory Blocks

User directory blocks have type T.SHORT and secondary type ST.USERDIRECTORY. The six information words at the start of the block also indicate the block's own key (that is, the block number) as a consistency check and the size of the hash table. The 50 information words at the end of the block contain the date and time of creation, the name of the directory, a pointer to the next file or directory on the hash chain, and a pointer to the directory above.

To find a file or sub-directory, you must first apply a hash function to its name. This hash function yields an offset in the hash table, which is the key of the first block on a chain linking those with the same hash value (or zero, if there are none). AmigaDOS reads the block with this key and compares the name of the block with the required name. If the names do not match, it reads the next block on the chain, and so on.

1.1.3 File Header Block

The following figure describes the layout of the file header block.

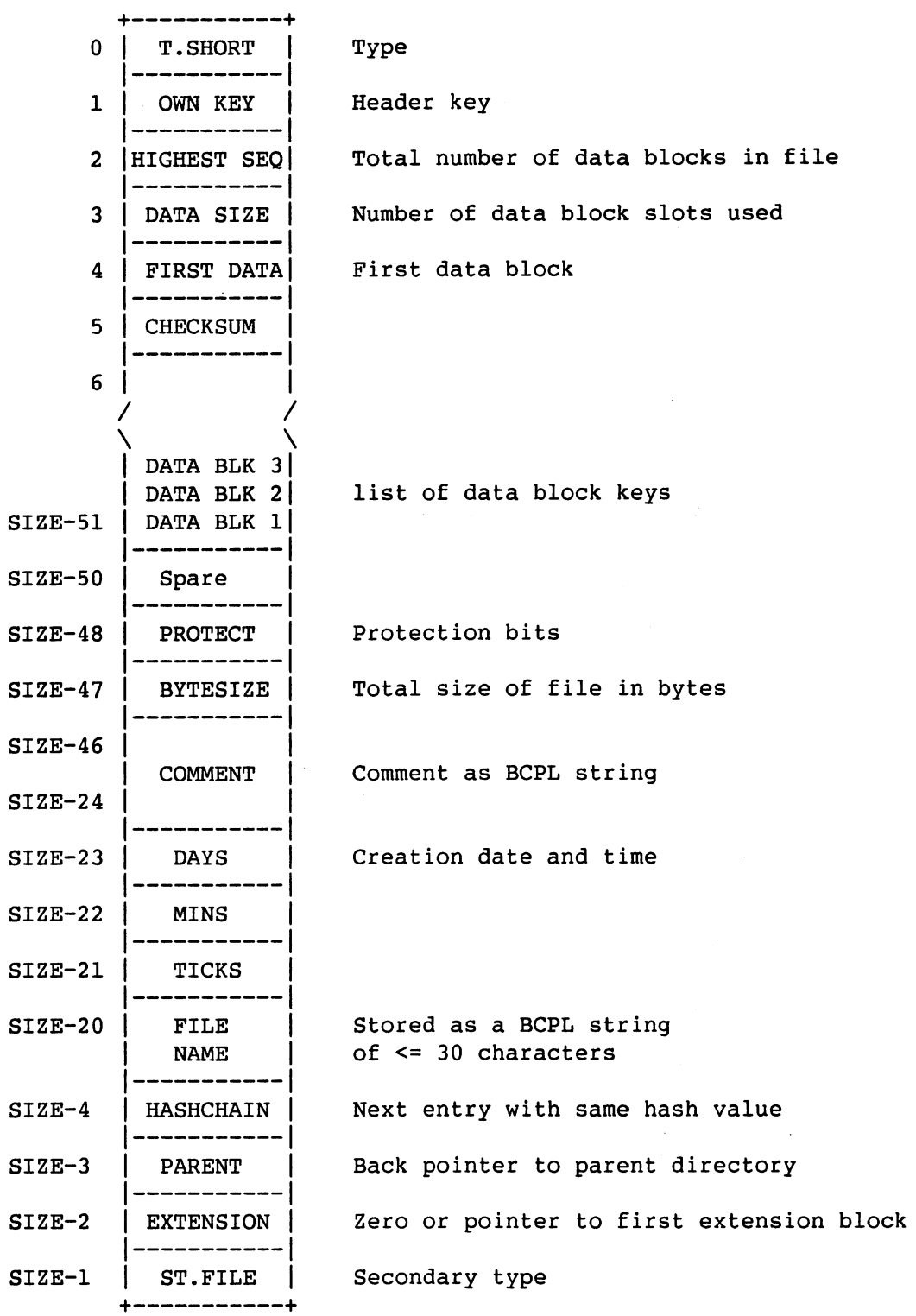


Figure 1-C: File Header Block

Each terminal file starts with a file header block, which has type T.SHORT and secondary type ST.FILE. The start and end of the block contain name, time, and redundancy information similar to that in a directory block. The body of the file consists of Data blocks with sequence numbers from 1 upwards. AmigaDOS stores the addresses of these blocks in consecutive words downwards from offset size-51 in the block. In general, AmigaDOS does not use all the space for this list and the last data block is not full.

1.1.4 File List Block

If there are more blocks in the file than can be specified in the block list, then the EXTENSION field is non-zero and points to another disk block which contains a further data block list. The following figure explains the structure of the file list block.

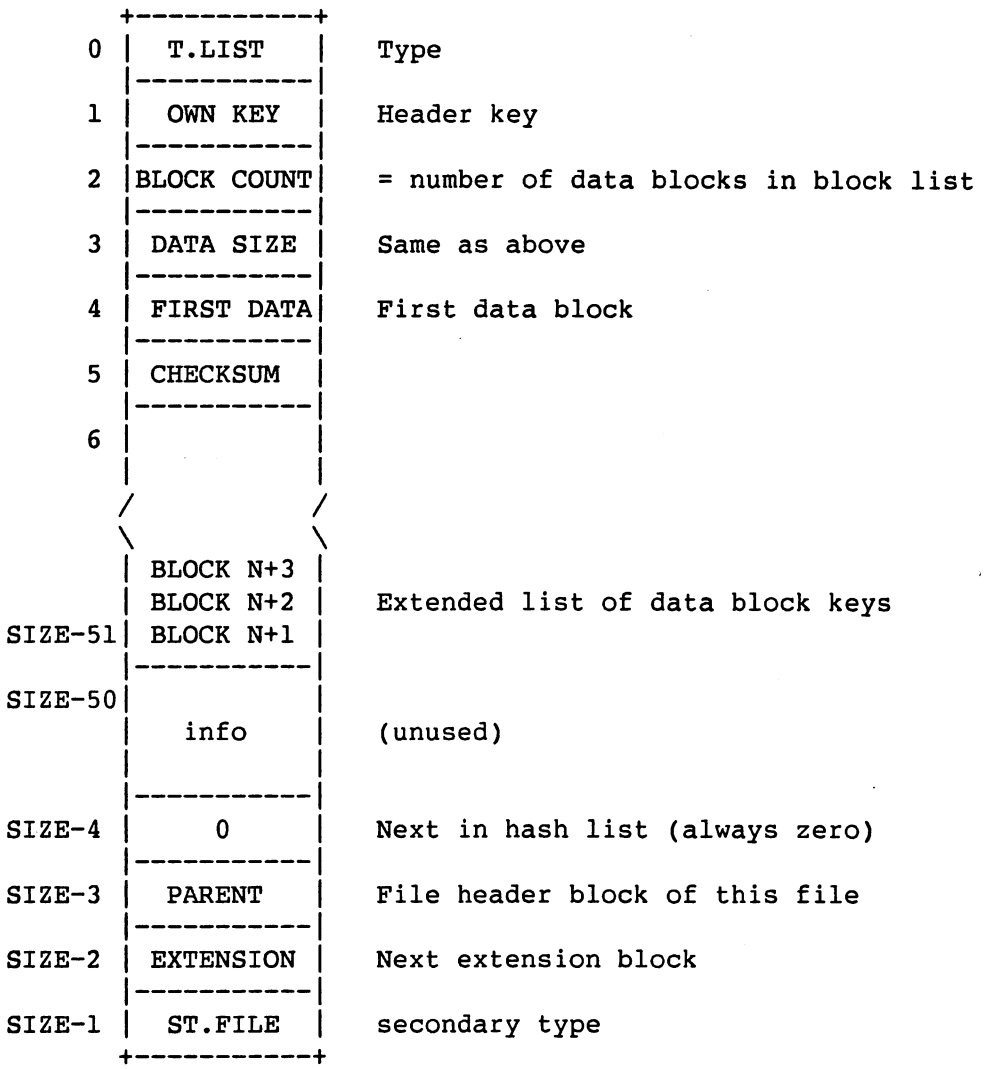


Figure 1-D: File List Block

There are as many file extension blocks as required to list the data blocks that make up the file. The layout of the block is very similar to that of a file header block, except that the type is different and the date and filename fields are not used.

1.1.5 Data Block

The following figure explains the layout of a data block.

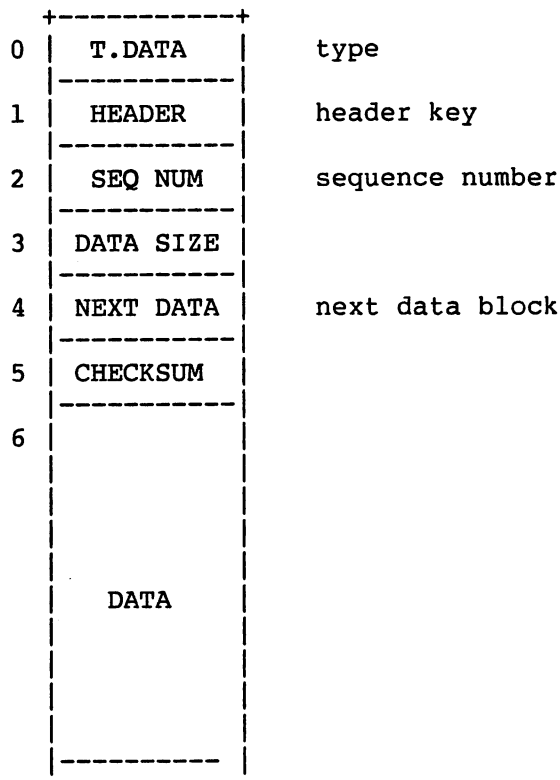


Figure 1-E: Data Block

Data blocks contain only six words of filing system information. These six words refer to the following:

- type (T.DATA)
- pointer to the file header block
- sequence number of the data block
- number of words of data
- pointer to the next data block
- checksum

Normally, all data blocks except the last are full (that is, they have a size = blocksize-6). The last data block has a forward pointer of zero.

1.2 DISKED - The Disk Editor

To inspect or patch disk blocks, you may use the AmigaDOS disk editor, DISKED. Because DISKED writes to the disk directly, you should use it with care. Nevertheless, you can use it to good effect in recovering information from a corrupt floppy disk, for example. A disk does not have to be inserted to be examined by DISKED.

You should only use DISKED with reference to the layout of an AmigaDOS disk. (For a description of the layout, see subsections 1.1.1 through 1.1.5 in the first part of the chapter.) DISKED knows about this structure - for example, the R (Root block) command prints the key of the root block. The G (Get block) command followed by this key number reads the block into memory, whereupon the I (Information) command prints out the information contained in the first and last locations, which indicate the type of block, the name, the hash links, and so on. If you specify a name after an H (Hash) command, DISKED gives you the offset on a directory page that stores as the first key headers with names that hash to the name you supplied. If you then type the number that DISKED returns followed by a slash (/), DISKED displays the key of that header page. You can then read this with further G commands, and so on.

Consider deleting a file that, due to hardware errors, makes the filing system restart process fail. First, you must locate the directory page that holds the reference to the file. You do this by searching the directory structure from the root block, using the hash codes. Then, you must locate the slot that references the file - this is either the directory block or a header block on the same hash chain. This slot should contain the key of the file's header block. To set the slot to zero, you type the slot offset, followed by a slash (/) followed by zero (that is, <offset>/0). Then correct the checksum with the K (checKsum) command. You should disable the write protection with X and write back the updated block with P (for Put block) or W (for Windup). There is no need to do anything else, as the blocks that the file used in error become available once more after the RESTART process has sucessfully scanned the disk.

DISKED commands are all single characters, sometimes with arguments.

The following is a complete list of the available commands.

Command	Function
B n	Set logical block number base to n
C n	Display n characters from current offset
G [n]	Get block n from disk (default is the current block number)
H name	Calculate hash value of name
I	Display block information
K	Check block checksum (and correct if wrong)
L [lwb upb]	Locate words that match Value under Mask (lwb and upb restrict search)
M n	Set Mask (for L and N commands) to n
N [lwb upb]	Locate words that do not match Value under Mask
P n	Put block in memory to block n on disk (default is the current block number)
R	Display block number of Root Block
Q	Quit (do not write to disk)
S char	Set display Style char = C -> characters S -> string O -> octal X -> hex D -> decimal
T lwb upb	Type range of offsets in block
V n	Set Value for L and N commands
W	Windup (=PQ)
X	Invert write protect state
Y n	Set cYlinder base to n
Z	Zero all words of buffer
number	Set current word offset in block = Display values set in program
/ [n]	Display word at current offset or update value to n
'chars'	Put chars at current offset
"chars"	Put string at current offset

Table 1.A: DISKED Commands

To indicate octal or hex, you can start numbers with # or #X (that is, # for octal, #X for hex). You can also include include BCPL string escapes (*N and so forth) in strings.

Chapter 2: Amiga Binary File Structure

This chapter describes the structure of binary object files for the Amiga, as produced by assemblers and compilers. It also describes the format of binary load files, which are produced by the linker and read into memory by the loader.

2.1 Introduction

This chapter describes the structure of binary object files for the Amiga, as produced by assemblers and compilers. It also describes the format of binary load files, which are produced by the linker and read into memory by the loader. The format of load files supports overlaying. Apart from describing the format of load files, this chapter explains the use of common symbols, absolute external references, and program units.

2.1.1 Terminology

Some of the technical terms used in this chapter are explained below.

External References

You can use a name to specify a reference between separate program units. The data structure lets you have a name longer than 16Mbytes, although the linker restricts names to 255 characters. When you link the object files into a single load file, you must ensure that all external references match corresponding external definitions. The external reference may be of size byte, word, or long; external definitions refer to relocatable values, absolute values, or resident libraries. Relocatable byte and word references refer to PC relative address modes and these are entirely handled by the linker. However, if you have a program containing longword relocatable references, relocation may take place when you load the program.

Note that these sizes only refer to the length of the relocation field; it is possible to load a word from a long external address, for example, and the linker makes no attempt to check that you are consistent in your use of externals.

Object File

An assembler or compiler produces a binary image, called an object file. An object file contains one or more program units. It may also contain external references to other object files.

Load File

The linker produces a binary image from a number of object files. This binary image is called a load file. A load file does not contain any unresolved external references.

Program Unit

A program unit is the smallest element the linker can handle. A program unit can contain one or more hunks; object files can contain one or more program units. If the linker finds a suitable external reference within a program unit when it inspects the scanned libraries, it includes the entire program unit in the load file. An assembler usually produces a single program unit from one assembly (containing one or more hunks); a compiler such as FORTRAN produces a program unit for each subroutine, main program, or BLOCK DATA. Hunk numbering starts from zero within each program unit; the only way you can reference other program units is through external references.

Hunks

A hunk consists of a block of code or data, relocation information, and a list of defined or referenced external symbols. Data hunks may specify initialized data or uninitialized data (bss). bss hunks may contain external definitions but no external references nor any values requiring relocation. If you place initialized data blocks in overlays, the linker should not normally alter these data blocks, since it reloads them from disk during the overlay process. Hunks may be named or unnamed, and they may contain a symbol table in order to provide symbolic debugging information. They may also contain a further debugging information for the use of high level language debugging tools. Each hunk within a program unit has a number, starting from zero.

Resident Library

Load files are also known as 'libraries'. Load files may be resident in memory; alternatively, the operating system may load them as part of the 'library open' call. You can reference resident libraries through external references; the definitions are in a hunk containing no code just a list of resident library definitions. Usually, to produce these hunks, you assemble a file containing nothing but absolute external definitions and then pass it through a special software tool to convert the absolute definitions to resident library definitions. The linker uses the hunk name as the name of the resident library, and it passes this through into the load file so that the loader can open the resident library before use.

Scanned library

A scanned library consists of object files that contain program units which are only loaded if there are any outstanding external references to them. You may use object files as libraries and provide them as primary input to the linker, in which case the input includes all the program units the object files contain. Note that you may concatenate object files.

Node

A node consists of at least one hunk. An overlaid load file contains a root node, which is resident in memory all the time that the program is running, and a number of overlay nodes which are brought into memory as required.

2.2 Object File Structure

An object file is the output of the assembler or a language translator. To use an object file, you must first resolve all the external references. To do this, you pass the object file through the linker. An object file consists of one or more program units. Each program unit starts with a header and is followed by a series of hunks joined end to end, each of which contains a number of 'blocks' of various types. Each block starts with a longword which defines its type, and this is followed by zero or more additional longwords. Note that each block is always rounded up the nearest longword boundary. The program unit header is also a block with this format.

The format of a program unit is as follows:

. Program unit header block
. Hunks

The basic format of a hunk is as follows:

. Hunk name block
. Relocatable block
. Relocation information block
. External symbol information block
. Symbol table block
. Debug block
. End block

You may omit all these block types, except the end block.

The following subsections describe the format of each of these blocks. The value of the type word appears in decimal and hex after the type name, for example, hunk_unit has the value 999 in decimal and 3E7 in hex.

2.2.1 hunk_unit (999/3E7)

This specifies the start of a program unit. It consists of a type word, followed by the length of the unit name in longwords, followed by the name itself padded to a longword boundary with zeros, if required. In diagramatic form, the format is as follows:

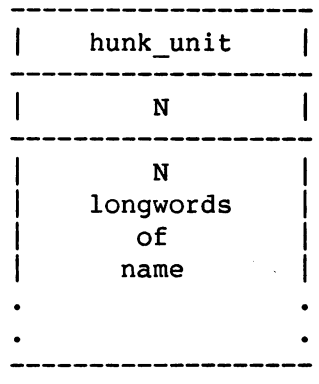


Figure 2-A: Hunk_unit (999/3E7)

2.2.2 hunk_name (1000/3E8)

This defines the name of a hunk. Names are optional; if the linker finds two or more named hunks with the same name, it combines the hunks into a single hunk. Note that 8 or 16 bit program counter relative external references can only be resolved between hunks with the same name. Any external references in a load format file are between different hunks and require 32 bit relocatable references; although, as the loader scatter loads the hunks into memory, you cannot assume that they are within 32K of each other. Note that the length is in longwords and the name block, like all blocks, is rounded up to a longword boundary by padding with zeros. The format is as follows:

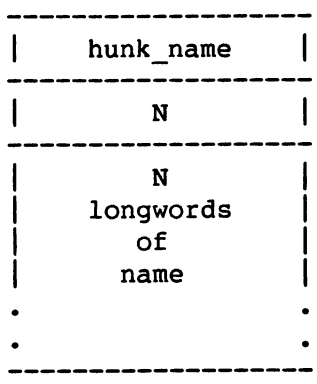


Figure 2-B: Hunk_name (1000/3E8)

2.2.3 hunk_code (1001/3E9)

This defines a block of code that is to be loaded into memory and possibly relocated. Its format is as follows:

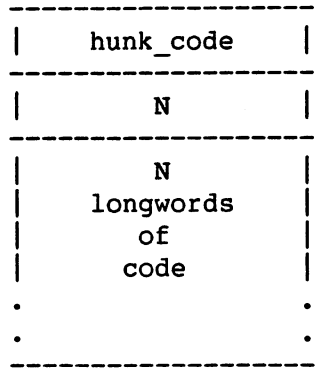


Figure 1-C: Hunk_code (1001/3E9)

2.2.4 hunk_data (1002/3EA)

This defines a block of initialized data which is to be loaded into memory and possibly relocated. The linker should not alter these blocks if they are part of an overlay node, as it may need to reread them from disk during overlay handling. The format is as follows:

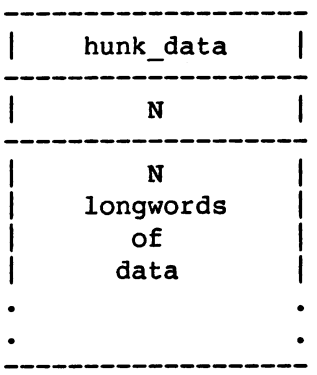


Figure 1-D: Hunk_data (1002/3EA)

2.2.5 hunk_bss (1003/3EB)

This specifies a block of uninitialized workspace which is allocated by the loader. bss blocks are used for such things as stacks and for FORTRAN COMMON blocks. It is not possible to relocate inside a bss block, but symbols can be defined within one. Its format is as follows:

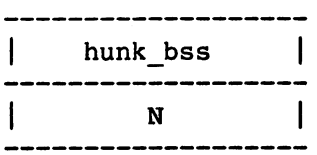


Figure 1-E: Hunk_bss (1003/3EB)

where N is the size of block you require in longwords. The memory used for bss blocks is zeroed by the loader when it is allocated.

The relocatable block within a hunk must be one of hunk_code, hunk_data or hunk_bss.

2.2.6 hunk_reloc32 (1004/3EC)

A hunk_reloc32 block specifies 32 bit relocation that the linker is to perform within the current relocatable block. The relocation information is a reference to a location within the current hunk or any other within the program unit. Each hunk within the unit is numbered, starting from zero. The linker adds the address of the base of the specified hunk to each of the longwords in the preceding relocatable block that the list of offsets indicates. The offset list only includes referenced hunks and a count of zero indicates the end of the list. Its format is as follows:

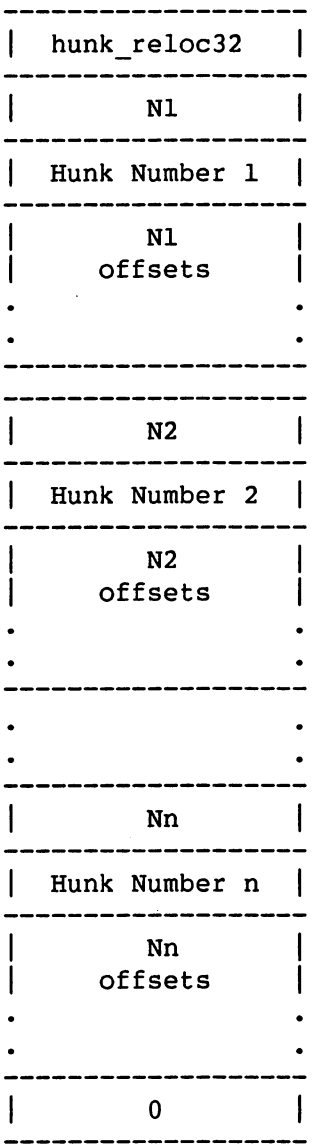


Figure 2-F: Hunk_reloc32 (1004/3EC)

2.2.7 hunk_reloc16 (1005/3ED)

A hunk_reloc16 block specifies 16 bit relocation that the linker should perform within the current relocatable block. The relocation information refers to 16 bit program counter relative references to other hunks in the program unit. The format is the same as hunk_reloc32 blocks. These references must be to hunks with the same name, so that the linker can perform the relocation while it coagulates (that is, gathers together) similarly named hunks.

2.2.8 hunk_reloc8 (1006/3EE)

A hunk_reloc8 block specifies 8 bit relocation that the linker should perform within the current relocatable block. The relocation information refers to 8 bit program counter relative references to other hunks in the program unit. The format is the same as hunk_reloc32 blocks. These references must be to hunks with the same name, so that the linker can perform the relocation while it coagulates similarly named hunks.

2.2.9 hunk_ext (1007/3EF)

This block contains external symbol information. It contains entries both defining symbols and listing references to them. Its format is as follows:

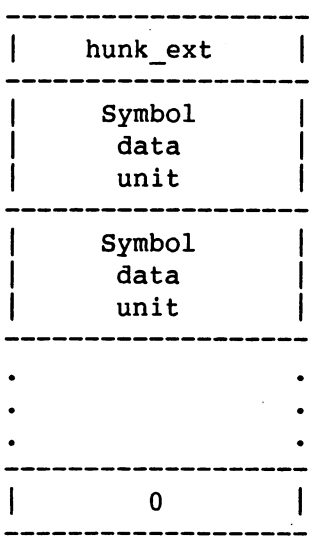


Figure 2-G: Hunk_ext (1007/3EF)

where there is one 'symbol data unit' for each symbol used, and the block ends with a zero word.

Each symbol data unit consists of a type byte, the symbol name length (three bytes), the symbol name itself, and further data. You specify the symbol name length in longwords, and pad the name field to the next longword boundary with zeros.

The type byte specifies whether the symbol is a definition or a reference, etc. AmigaDOS uses values 0-127 for symbol definitions, and 128-255 for references.

At the moment, the values are as follows:

Name	Value	Meaning
ext_symb	0	Symbol table - see symbol block below
ext_def	1	Relocatable definition
ext_abs	2	Absolute definition
ext_res	3	Resident library definition
ext_ref32	129	32bit reference to symbol
ext_common	130	32bit reference to COMMON
ext_ref16	131	16bit reference to symbol
ext_ref8	132	8bit reference to symbol

Table 2-A: External Symbols

The linker faults all other values. For ext_def there is one data word, the value of the symbol. This is merely the offset of the symbol from the start of the hunk. For ext_abs there is also one data value, which is the absolute value to be added into the code. The linker treats the value for ext_res in the same way as ext_def, except that it assumes the hunk name is the library name and it copies this name through to the load file. The type bytes ext_ref32, ext_ref16 and ext_ref8 are followed by a count and a list of references, again specified as offsets from the start of the hunk.

The type ext_common has the same structure except that it has a COMMON block size before the count. The linker treats symbols specified as common in the following way: if it encounters a definition for a symbol referenced as common, then it uses this value (the only time a definition should arise is in the FORTRAN Block Data case). Otherwise, it allocates suitable bss space using the maximum size you specified for each common symbol reference.

The linker handles external references differently according to the type of the corresponding definition. It adds absolute values to the long, word, or byte field and gives an error if the signed value does not fit. Relocatable 32bit references have the symbol value added to the field and a relocation record is produced for the loader. 16 and 8bit references are handled as PC relative references and may only be made to hunks with the same name so that the hunks are coagulated by the linker before they are loaded. It also possible for PC relative references to fail if the reference and the definition are too far apart. The linker may only access resident library definitions with 32bit references, which it then handles as relocatable 32bit references. The symbol data unit formats are as follows:

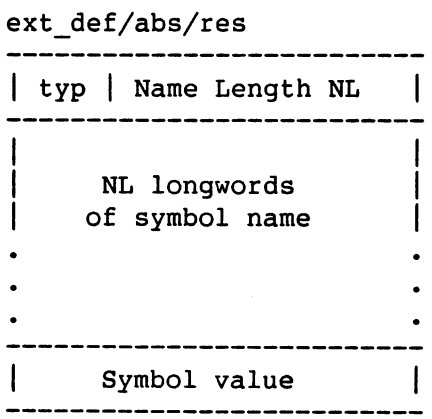
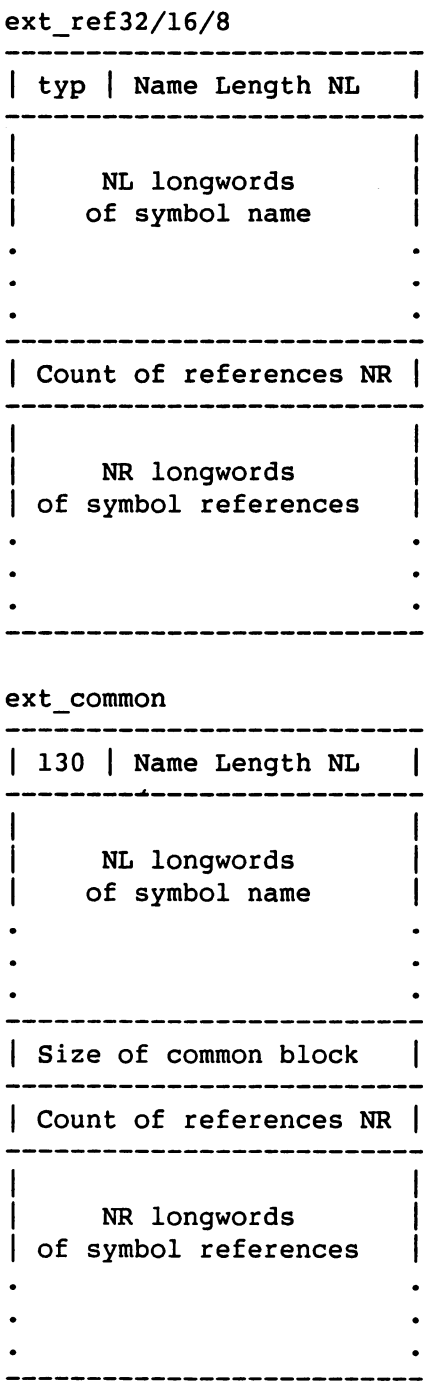


Figure 2-H: Symbol Data Unit



(continuation of Figure 2-H)

2.2.10 hunk_symbol (1008/3F0)

You use this block to attach a symbol table to a hunk so that you can use a symbolic debugger on the code. The linker passes symbol table blocks through attached to the hunk and, if the hunks are coagulated, coagulates the symbol tables. The loader does not load symbol table blocks into memory; when this is required, the debugger is expected to read the load file. The format of the symbol table block is the same as the external symbol information block with symbol table units for each name you use. The type code of zero is used within the symbol data units. The value of the symbol is the offset of the symbol from the start of the hunk. Thus the format is as follows:

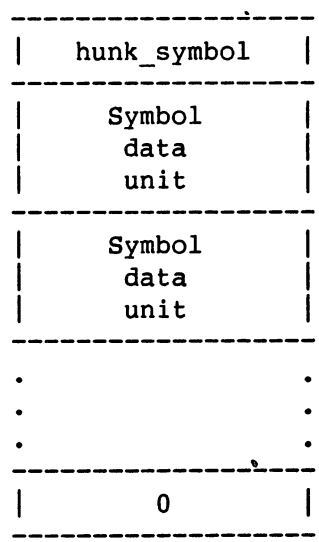


Figure 2-I: Hunk_symbol (1008/3F0)

where each symbol data unit has the following format.

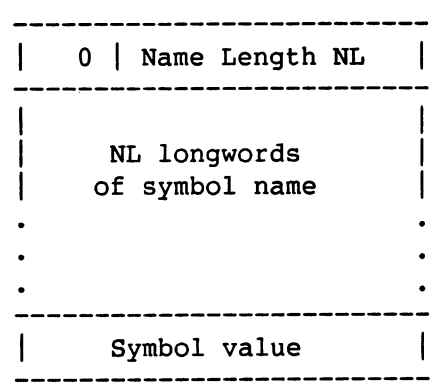


Figure 2-J: Symbol Data Unit

2.2.11 hunk_debug (1009/3F1)

AmigaDOS provides the debug block so that an object file can carry further debugging information. For example, high level language compilers may need to maintain descriptions of data structures for use by high level debuggers. The debug block may hold this information. AmigaDOS does not impose a format on the debug block except that it must start with the hunk_debug longword and be followed by a longword that indicates the size of the block in longwords. Thus the format is as follows:

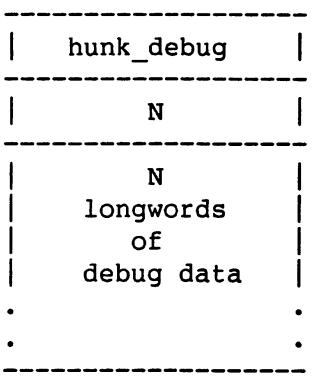


Figure 2-K: Hunk_debug (1009/3F1)

2.2.12 hunk_end (1010/3F2)

This specifies the end of a hunk. It consists of a single longword, hunk_end.

2.3 Load Files

The format of a load file (that is, the output from the linker) is similar to that of an object file. In particular, it consists of a number of hunks with a similar format to those in an object file. The main difference is that the hunks never contain an external symbol information block, as all external symbols have been resolved, and the program unit information is not included. In a simple load file that is not overlaid, the file contains a header block which indicates the total number of hunks in the load file and any resident libraries the program referenced. This block is followed by the hunks, which may be the result of coagulating a number of input hunks if they had the same name. This complete structure is referred to as a node. Load files may also contain overlay information. In this case, an overlay table follows the primary node, and a special break block separates the overlay nodes. Thus the load file structure can be summarized as follows, where the items marked with an asterisk (*) are optional.

. Primary node
. Overlay table block (*)
. Overlay nodes separated by break blocks (*)

The relocation blocks within the hunks are always of type hunk_reloc32, and indicate the relocation to be performed at load time. This includes both the 32 bit relocation specified with hunk_reloc32 blocks in the object file and extra relocation required for the resolution of external symbols.

Each external reference in the object files is handled as follows. The linker searches the primary input for a matching external definition. If it does not find one, it searches the scanned library and includes in the load file the entire program unit where the definition was defined. This may make further external references become outstanding. At the end of the first pass, the linker knows all the external definitions and the total number of hunks that it is going to use. These include the hunks within the load file and the hunks associated with the resident libraries. On the second pass, the linker patches the longword external references so that they refer to the required offset within the hunk which defines the symbol. It produces an extra entry in the relocation block so that, when the hunks are loaded, it adds to each external reference the base address of the hunk defining the symbol. This mechanism also works for resident libraries.

Before the loader can make these cross hunk references, it needs to know the number and size of the hunks in the nodes. The header block provides this information, as described below. The load file may also contain overlay information in an overlay table block. Break blocks separate the overlay nodes.

2.3.1 hunk_header (1011/3F3)

This block gives information about the number of hunks that are to be loaded, and the size of each one. It also contains the names of any resident libraries which must be opened when the node is loaded.

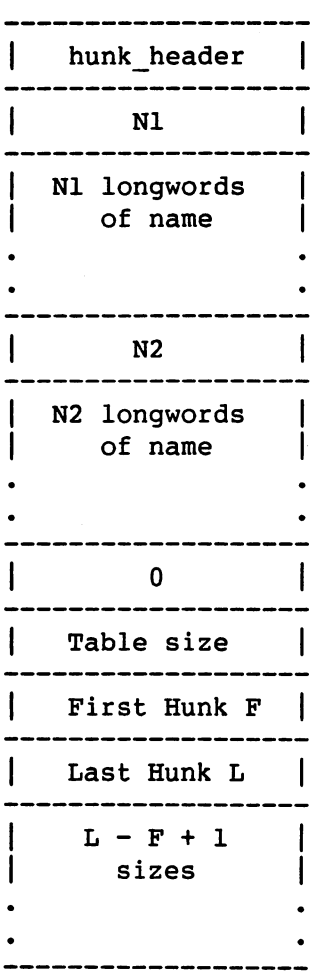


Figure 2-L: Hunk_header (1011/3F3)

The format of the hunk_header is described in Figure 2-L. The first part of the header block contains the names of resident libraries that the loader must open when this node is loaded. Each name consists of a long word indicating the length of the name in longwords and the text name padded to a longword boundary with zeros. The name list ends with a longword of zero. The names are in the order in which the loader is to open them.

When it loads a primary node, the loader allocates a table in memory which it uses to keep track of all the hunks it has loaded. This table must be large enough for all the hunks in the load file, including the hunks in overlays. The loader also uses this table to keep a copy of the hunk tables associated with any resident libraries. The next longword in the header block is therefore this table size, which is equal to the maximum hunk number referenced plus one.

The next longword F refers to the first slot in the hunk table the loader should use when loading. For a primary node that does not reference a resident library, this value is zero; otherwise, it is the number of hunks in the resident libraries. The loader copies these entries from the hunk table associated with the library following a library open call. For an overlay node, this value is the number of hunks in any resident libraries plus the number of hunks already loaded in ancestor nodes.

The next longword L refers to the last hunk slot the loader is to load as part of this loader call. The total number of hunks loaded is therefore L - F + 1.

The header block continues with L - F + 1 longwords which indicate the sizes of each hunk which is to be loaded as part of this call. This enables the loader to preallocate the space for the hunks and hence perform the relocation between hunks which is required as they are loaded. One hunk may be the bss hunk with a size given as zero; in this case the loader uses an operating system variable to give the size as described in hunk_bss above.

2.3.2 hunk_overlay (1013/3F5)

The overlay table block indicates to the loader that it is loading an overlaid program, and contains all the data for the overlay table. On encountering it, the loader sets up the table, and returns, leaving the input channel to the load file still open. Its format is as follows:

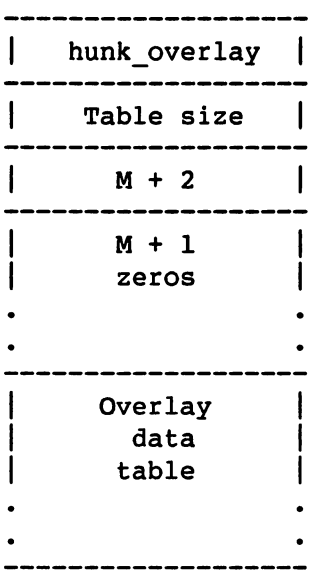


Figure 2-M: Hunk_overlay (1013/3F5)

The first longword is the upper bound of the complete overlay table (in longwords).

M is the maximum level the overlay tree uses with the root level being zero. The next M+1 words form the ordinate table section of the overlay table.

The rest of the block is the overlay data table, a series of eight-word entries, one for each overlay symbol. If O is the maximum overlay number used, then the size of the overlay data table is (O+1)*8, since the first overlay number is zero. So, the overlay table size is equal to (O+1)*8 + M+1.

2.3.3 hunk_break (1014/3F6)

A break block indicates the end of an overlay node. It consists of a single longword, hunk_break.

2.4 Examples

The following simple sections of code show how the linker and loader handle external symbols. For example,

```
                              IDNT   A
                              XREF   BILLY,JOHN
                              XDEF   MARY
* The next longword requires relocation
0000' 0000 0008               DC.L   FRED
0004' 123C 00FF               MOVE.B #$FF,D1
0008' 7001               FRED MOVEQ  #1,D0
* External entry point
000A' 4E71               MARY NOP
000C' 4EB9 0000 0000          JSR    BILLY   Call external
0012' 2239 0000 0000          MOVE.L JOHN,D1 Reference external
                              END
```

produces the following object file

```
hunk_unit
00000001        Size in longwords
41000000        Name, padded to lword
hunk_code
00000006        Size in longwords
00000008 123C00FF 70014E71 4EB90000 00002239 00000000
hunk_reloc32
00000001        Number in hunk 0
00000000        Hunk 0
00000000        Offset to be relocated
00000000        Zero to mark end
hunk_ext
01000001        XDEF, Size 1 longword
4D415259        MARY
0000000A        Offset of definition
81000001        XREF, Size 1 longword
4A4F484E        JOHN
00000001        Number of references
00000014        Offset of reference
81000002        XREF, Size 2 longwords
42494C4C        BILLY
59000000        (zeros to pad)
00000001        Number of references
0000000E        Offset of reference
00000000        End of external block
hunk_end
```

The matching program to this is as follows:

```
                                    IDNT    B
                                    XDEF    BILLY,JOHN
                                    XREF    MARY
0000' 2A3C  AAAA  AAAA              MOVE.L  #$AAAAAAAA,D5
* External entry point
0006' 4E71                BILLY     NOP
* External entry point
0008' 7201                JOHN      MOVEQ   #1,D1
* Call external reference
000A' 4EF9  0000  0000              JMP     MARY
                                    END
```

and the corresponding output code would be

```
hunk_unit
00000001        Size in longwords
42000000        Unit name
hunk_code
00000004        Size in longwords
2A3CAAAA AAAA4E71 72014EF9 00000000
hunk_ext
01000001        XDEF, Size 1 longword
4A4F484E        JOHN
00000008        Offset of definition
01000002        XDEF, Size 2 longwords
42494C4C        BILLY
59000000        (zeros to pad)
00000006        Offset of definition
81000001        XREF, Size 1 longword
4D415259        MARY
00000001        Number of references
0000000C        Offset of reference
00000000        End of external block
hunk_end
```

Once you passed this through the linker, the load file would have the following format.

```
hunk_header
00000000        No hunk name
00000002        Size of hunk table
00000000        First hunk
00000001        Last hunk
00000006        Size of hunk 0
00000004        Size of hunk 1
hunk_code
00000006        Size of code in longwords
00000008 123C00FF 70014E71 4EB90000 00062239 00000008
hunk_reloc32
00000001        Number in hunk 0
00000000        Hunk 0
00000000        Offset to be relocated
00000002        Number in hunk 1
00000001        Hunk 1
00000014        Offset to be relocated
0000000E        Offset to be relocated
00000000        Zero to mark end
hunk_end
hunk_code
00000004        Size of code in longwords
2A3CAAAA AAAA4E71 72014EF9 0000000A
hunk_reloc32
00000001        Number in hunk 0
00000000        Hunk 0
0000000C        Offset to be relocated
00000000        Zero to mark end
hunk_end
```

When the loader loads this code into memory, it reads the header block and allocates a hunk table of two longwords. It then allocates space by calling an operating system routine and requesting two areas of sizes 6 and 4 longwords respectively. Assuming the two areas it returned were at locations 3000 and 7000, the hunk table would contain 3000 and 7000.

The loader reads the first hunk and places the code at 3000; it then handles relocation. The first item specifies relocation with respect to hunk 0, so it adds 3000 to the longword at offset 0 converting the value stored there from 00000008 to 00003008. The second item specifies relocation with respect to hunk 1. Although this is not loaded, we know that it will be loaded at location 7000, so this is added to the values stored at 300E and 3014. Note that the linker has already inserted the offsets 00000006 and 00000008 into the references in hunk 0 so that they refer to the correct offset in hunk 1 for the definition. Thus the longwords specifying the external references end up containing the values 00007006 and 00007008, which is the correct place once the second hunk is loaded.

In the same way, the loader loads the second hunk into memory at location 7000 and the relocation information specified alters the longword at 700C from 0000000A (the offset of MARY in the first hunk) to 0000300A (the address of MARY in memory).

The loader handles references to resident libraries in the same way, except that, after it has opened the library, it copies the locations of the hunks comprising the library into the start of the hunk table. It then patches references to the resident library to refer to the correct place by adding the base of the library hunks.

Chapter 3: AmigaDOS Data Structures

This chapter describes AmigaDOS data structures in memory and in files. It does not describe the layout of a disk, which is described in Chapter 1.

Table of Contents

AmigaDOS process values are as follows:

Value	Function	Description
BPTR	SegArray	Array of SegLists used by this process
LONG	StackSize	Size of process stack in bytes
APTR	GlobVec	Global Vector for this process
LONG	TaskNum	CLI Task number or zero if not a CLI
BPTR	StackBase	Pointer to high memory end of process stack
LONG	IoErr	Value of secondary result from last call
BPTR	CurrentDir	Lock associated with current directory
BPTR	CIS	Current CLI input stream
BPTR	COS	Current CLI output stream
APTR	CoHand	Console handler process for current window
APTR	FiHand	File handler process for current drive
BPTR	CLIStruct	Pointer to additional CLI information
APTR	ReturnAddr	Pointer to previous stackframe
APTR	PktWait	Function to be called when awaiting message
APTR	WindowPtr	Pointer to window

To identify the segments that a particular process uses, you use SegArray. SegArray is an array of longwords with its size in SegArray[0]. Other elements are either zero or a BPTR to a SegList. CreateProc creates this array with the first two elements of the array pointing to resident code and the third element being the SegList passed as argument. When a process terminates, FreeMem is used to return the space for the SegArray.

StackSize indicates the size of the process stack, as supplied by the user when calling CreateProc. Note that the process stack is not the same as the command stack a CLI uses when it calls a program. The CLI obtains its command stack just before it runs a program and you may alter the size of this stack with the STACK command. When you create a process, AmigaDOS obtains the process stack and stores the size in StackSize. The pointer to the space for the process control block and the stack is also stored in the MemEntry field of the task structure. When the process terminates this space is returned via a call to FreeMem. You can also chain any memory you obtain into this list structure so that it, too, gets put back when the task terminates.

If a call to CreateProc creates the process, GlobVec is a pointer to the Shared Global Vector. However, some internal handler processes use a private GlobVec.

The value of TaskNum is normally zero; a CLI process stores the small integer that identifies the invocation of the CLI here.

The pointer StackBase points to the high-memory end of the process stack. This is the end of the stack when using languages such as C or Assembler; it is the base of the stack for languages such as BCPL.

The values of IoErr and CurrentDir are those handled by the similarly named AmigaDOS calls. CIS and COS are the values Input and Output returns and refer to the file handles you should use when running a program under the CLI. In other cases CIS and COS are zero.

CoHand and Fihand refer to the console handler for the current window and the file handler for the current device. You use these values when attempting to open the * device or a file by a relative path name.

The CLIStruct pointer is nonzero only for CLI processes. In this case it refers to a further structure the CLI uses with the following format:

Value	Function	Description
LONG	Result2	Value of IoErr from last command
BSTR	SetName	Name of current directory
BPTR	CommandDir	Lock associated with command directory
LONG	ReturnCode	Return code from last command
BSTR	CommandName	Name of current command
LONG	FailLevel	Fail level (set by FAILAT)
BSTR	Prompt	Current prompt (set by PROMPT)
BPTR	StandardIn	Default (terminal) CLI input
BPTR	CurrentIn	Current CLI input
BSTR	CommandFile	Name of EXECUTE command file
LONG	Interactive	Boolean; True if prompts required
LONG	Background	Boolean; True if CLI created by RUN
BPTR	CurrentOut	Current CLI output
LONG	DefaultStack	Stack size to be obtained (in lwords)
BPTR	StandardOut	Default (terminal) CLI output
BPTR	Module	SegList of currently loaded command

The Exit function uses the value of ReturnAddr which points to just above the return address on the currently active stack. If a program exits by performing an RTS on an empty stack, then control passes to the code address pushed onto the stack by CreateProc or by the CLI. If a program terminates with a call to Exit, then AmigaDOS uses this pointer to extract the same return address.

The value of PktWait is normally zero. If it is nonzero, then AmigaDOS calls PktWait whenever a process is about to go to sleep to await a signal indicating that a message has arrived. In the same way as GetMsg, the function should return a message when one is available. Usually, you use this function to filter out any private messages arriving at the standard process message port that are not intended for AmigaDOS.

The value of WindowPtr is used when AmigaDOS detects an error that normally requires the user to take some action. Examples of these errors are attempting to write to a write-protected disk, or when the disk is full. If the value of WindowPtr is -1, then the error is returned to the calling program as an error code from the AmigaDOS call of Open, Write, or whatever. If the value is zero, then AmigaDOS places a request box on the WorkBench screen informing the user of the error and providing the opportunity to retry the operation or to cancel it. If the user selects cancel then AmigaDOS returns the error code to the calling program. If the user selects retry, or inserts a disk, then AmigaDOS attempts the operation once more.

If you place a positive value into the WindowPtr field, then AmigaDOS takes this to be a pointer to a Window structure. Normally you would place the Window structure of the window you are currently using here. In this case, AmigaDOS displays the error message within the window you have specified, rather than use the WorkBench screen. You can always leave the WindowPtr field as zero, but if you are using another screen, then the messages AmigaDOS displays appear on the WorkBench screen, possibly obscured by your own screen.

The initial value of WindowPtr is inherited from the process that created the current one. If you decide to alter WindowPtr from within a program that runs under the CLI, then you should save the original value and restore it when you finish; otherwise, the CLI process contains a WindowPtr that refers to a window that is no longer present.

3.3 Global Data Structure

This data structure only exists once; however, all AmigaDOS processes use it. If you make a call to OpenLibrary, you can obtain the library base pointer. The base of the data structure is a positive offset from the library base pointer. The library base pointer points to the following structure:

Library Node structure
APTR to DOS RootNode
APTR to DOS Shared Global Vector
DOS private register dump

All internal AmigaDOS calls use the Shared Global Vector, which is a jump table. You should not normally use it, except through the supplied interface calls, as it is liable to change without warning.

The RootNode structure is as follows:

Value	Function	Description
BPTR	TaskTable	Array of CLI processes currently running
BPTR	CLISegList	SegList for the CLI
LONG	Days	Number of days in current time
LONG	Mins	Number of minutes in current time
LONG	Ticks	Number of ticks in current time
BPTR	RestartSeg	SegList for the disk validator process
BPTR	Info	Pointer to the Info substructure

The TaskTable is an array with the size of the array stored in TaskTable[0]. The processid (in other words, the MsgPort associated with the process) for each CLI is stored in the array. The processid for the CLI with TaskNum n is stored in TaskTable[n]. An empty slot is filled with a zero. The commands RUN and NEWCLI scan the TaskTable to identify the next free slot, and use this as the TaskNum for the CLI created.

The CLISegList is the SegList for the code of the CLI. RUN and NEWCLI use this value to create a new instance of a CLI.

The rootnode stores the current date and time; normally you should use the AmigaDOS function DateStamp to return a consistent set of values. The values Days, Mins, and Ticks specify the date and time. The value of Days is the number of days since January 1st, 1978. The value of Mins is the number of minutes since midnight. A tick is one fiftieth of a second, but the time is only updated once per second.

The RestartSeg is the SegList for the code of the disk validator, which is a process that AmigaDOS creates whenever you insert a new disk into a drive.

3.3.1 Info Substructure

To access a further substructure with the following format, you use the Info pointer.

Value	Function	Description
BPTR	McName	Network name of this machine; currently zero
BPTR	DevInfo	Device list
BPTR	Devices	Currently zero
BPTR	Handlers	Currently zero
APTR	NetHand	Network handler processid, currently zero

Most of the fields in the Info substructure are empty at the moment, but Commodore-Amiga intend to use them for expanding the system.

The DevInfo structure is a linked list. You use it to identify all the device names that AmigaDOS knows about; this includes ASSIGNed names and disk volume names. There are two possible formats for the list entries depending on whether the entry refers to a disk volume or not. For an entry describing a device or a directory (via ASSIGN) the entry is as follows:

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
LONG	Type	List entry type (device or dir)
APTR	Task	Handler process or zero
BPTR	Lock	File system lock or zero
BSTR	Handler	File name of handler or zero
LONG	StackSize	Stack size for handler process
LONG	Priority	Priority for handler process
LONG	Startup	Startup value to be passed to handler process
BPTR	SegList	SegList for handler process or zero
BPTR	GlobVec	Global vector for handler process or zero
BSTR	Name	Name of device or ASSIGNed name

The Next field links all the list entries together, and the name of the logical device name is held in the Name field.

The Type field is 0 (dt_device) or 1 (dt_dir). You can make a directory entry with the ASSIGN command. This command allocates a name to a directory that you can then use as a device name. If the list entry refers to a directory, then the Task refers to the file system process handling that disk, and the Lock field contains a pointer to a lock on that directory.

If the list entry refers to a device, then the device may or may not be resident. If it is resident, the Task identifies the handler process, and the Lock is normally zero. If the device is not resident, then the Task is zero and AmigaDOS uses the rest of the list structure.

If the SegList is zero, then the code for the device is not in memory. The Handler field is a string specifying the file containing the code (for example, SYS:L/RAM-HANDLER). A call to LoadSeg loads the code from the file and inserts the result into the SegList field.

AmigaDOS now creates a new handler process with the SegList, StackSize, and Pri values. The new process is a BCPL process and requires a Global Vector; this is either the value you specified in GlobVec or a new private global vector if GlobVec is zero.

The new process is passed a message containing the name originally specified, the value stored in Startup and the base of the list entry. The new handler process may then decide to patch into the Task slot the process id or not as required. If the task slot is patched, then subsequent references to the device name use the same handler task; this is what the RAM: device does. If the task slot is not patched, then further references to the device result in new process invocations; this is what the CON: device does.

If the type field within the list entry is equal to 2 (dt_volume), then the format of the list structure is slightly different.

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
LONG	Type	List entry type (volume)
APTR	Task	Handler process or zero
BPTR	Lock	File system lock
LONG	VolDays	Volume creation date
LONG	VolMins	
LONG	VolTicks	
BPTR	LockList	List of active locks for this volume
LONG	DiskType	Type of disk
LONG	Spare	Not used
BSTR	Name	Volume name

In this case, the name field is the name of the volume, and the Task field refers to the handler process if the volume is currently inserted; or to zero if the volume is not inserted. To distinguish disks with the same name, AmigaDOS timestamps the volume on creation and then saves the timestamp in the list structure. AmigaDOS can therefore compare the timestamps of different volumes whenever necessary.

If a volume is not currently inserted, then AmigaDOS saves the list of currently active locks in the LockList field. It uses the DiskType field to identify the type of disk, currently this is always an AmigaDOS disk. The disk type is up to four characters packed into a long word and padded on the right with nulls.

3.4 Memory Allocation

AmigaDOS obtains all the memory it allocates by calling the AllocMem function provided by Exec. In this way, AmigaDOS obtains structures such as locks and file handles; it usually places them back in the free pool by calling FreeMem. Each memory segment allocated by AmigaDOS is identified by a BPTR to the second longword in the structure. The first longword always contains the length of the entire segment in bytes. Thus the structure of allocated memory is as follows.

Value	Function	Description
LONG	BlockSize	Size of memory block
LONG	FirstData	First data segment, BPTR to block points here

3.5 Segment Lists

To obtain a segment list, you call LoadSeg. The result is a BPTR to allocated memory, so that the length of the memory block containing each list entry is stored at -4 from the BPTR. This length is 8 more than the size of the segment list entry, allowing for the link field and the size field itself.

The SegList is a list linked together by BPTRs and terminated by zero. The remainder of each segment list entry contains the code loaded. Thus the format is

Value	Function	Description
LONG	NextSeg	BPTR to next segment or zero
LONG	FirstCode	First value from binary file

3.6 File Handles

File handles are created by the AmigaDOS function Open, and you use them as arguments to other functions such as Read and Write. AmigaDOS returns them as a BPTR to the following structure.

Value	Function	Description
LONG	Link	Not used
LONG	Interact	Boolean, TRUE if interactive
LONG	ProcessID	Process id of handler process
BPTR	Buffer	Buffer for internal use
LONG	CharPos	Character position for internal use
LONG	BufEnd	End position for internal use
APTR	ReadFunc	Function called when buffer exhausted
APTR	WriteFunc	Function called when buffer full
APTR	CloseFunc	Function called when handle closed
LONG	Arg1	Argument; depends on file handle type
LONG	Arg2	Argument; depends on file handle type

Most of the fields are only used by AmigaDOS internally; normally Read or Write uses the file handle to indicate the handler process and any arguments to be passed. Values should not be altered within the file handle by user programs, except that the first field may be used to link file handles into a singly linked list.

3.7 Locks

The filing system extensively uses a data structure called a lock. This structure serves two purposes. First, it serves as the mechanism to open files for multiple reads or a single write. Note that obtaining a shared read lock on a directory does not stop that directory being updated.

Second, the lock provides a unique identification for a file. Although a particular file may be specified in many ways, the lock is a simple handle on that file. The lock contains the actual disk block location of the directory or file header and is thus a shorthand way of specifying a particular file system object.

The structure of a lock is as follows.

Value	Function	Description
BPTR	NextLock	BPTR to next in chain, else zero
LONG	DiskBlock	Block number of directory or file header
LONG	AccessType	Shared or exclusive access
APTR	ProcessID	Process ID of handler task
BPTR	VolNode	Volume entry for this lock

Because AmigaDOS uses the NextLock field to chain locks together, you should not alter it. The filing system fills in DiskBlock field to represent the location on disk of the directory block or the file header block. The AccessType serves to indicate whether this is a shared read lock, when it has the value -2, or an exclusive write lock when it has the value -1. The ProcessID field contains a pointer to the handler process for the device containing the file to which this lock refers. Finally the VolNode field points to the node in the DevInfo structure that identifies the volume to which this lock refers. Volume entries in the DevInfo structure remain there if a disk is inserted or if there are any locks open on that volume.

Note that a lock can also be a zero. The special case of lock zero indicates that the lock refers to the root of the initial filing system, and the FiHand field within the process data structure gives the handler process.

3.8 Packets

Packet passing handles all communication performed by AmigaDOS between processes. A packet is a structure built on top of the message passing mechanism provided by the Exec kernel.

An Exec message is a structure, described elsewhere, that includes a Name field. AmigaDOS uses this field as an APTR to another section of memory called a packet. A packet must be long word aligned, and has the following general structure.

Value	Function	Description
APTR	MsgPtr	Pointer back to message structure
APTR	MsgPort	Message port where the reply should be sent
LONG	PktType	Packet type
LONG	Res1	First result field
LONG	Res2	Second result field
LONG	Arg1	Argument; depends on packet type
LONG	Arg2	Argument; depends on packet type
	...	
LONG	ArgN	Argument; depends on packet type

The format of a specific packet depends on its type; but in all cases, it contains a back pointer to the Message structure, the MsgPort for the reply, and two result fields. When AmigaDOS sends a packet, the reply port is overwritten with the process identifier of the sender so that the packet can be returned. Thus, when sending a packet to an AmigaDOS handler process, you must fill in the reply MsgPort each time; otherwise, when the packet returns, AmigaDOS has overwritten the original port. AmigaDOS maintains all other fields except the result fields.

All AmigaDOS packets are sent to the message port created as part of a process; this message port is initialized so that arriving messages cause signal 8 to be set. An AmigaDOS process which is waiting for a message waits for signal 8 to be set. When the process wakes up because this event has occurred,

GetMsg takes the message from the message port and extracts the packet address. If the process is an AmigaDOS handler process, then the packet contains a value in the PktType field which indicates an action to be performed, such as reading some data. The argument fields contain specific information such as the address and size of the buffer where the characters go.

When the handler process has completed the work required to satisfy this request, the packet returns to the sender, using the same message structure. Both the message structure and the packet structure must be allocated by the client and must not be deallocated before the reply has been received. Normally AmigaDOS is called by the client to send the packet, such as when a call to Read is made. However, there are cases when asynchronous IO is required, and in this case the client may send packets to the handler processes as required. The packet and message structures must be allocated, and the processid field filled in with the message port where this packet must return. A call to PutMsg then sends the message to the destination. Note that many packets may be sent out returning to either the same or different message ports.

3.8.1 Packet Types

AmigaDOS supports the following packet types. Not all types are valid to all handlers, for example a rename request is only valid to handlers supporting a filing system. For each packet type the arguments and results are described. The actual decimal code for each type appears next to the symbolic name. In all cases, the Res2 field contains additional information concerning an error (indicated by a zero value for Res1 in most cases). To obtain this additional information, you can call IoErr when making a standard AmigaDOS call.

```
Open Old File
Type      LONG  Action.FindInput (1005)

Arg1      BPTR  FileHandle
Arg2      BPTR  Lock
Arg3      BSTR  Name

Res1      LONG  Boolean
```

Attempts to open an existing file for input or output (see the function Open in Chapter 2, "Calling AmigaDOS," of the *AmigaDOS Developer's Manual* for further details on opening files for I/O). To obtain the value of lock, you call DeviceProc to obtain the handler processid and then IoErr which returns the lock. Alternatively the lock and processid can be obtained directly from the DevInfo structure. Note that the lock refers to the directory owning the file, not to the file itself.

The caller must allocate and initialize FileHandle. This is done by clearing all fields to zero except for the CharPos and BufEnd fields which should be set to -1. The ProcessID field within the FileHandle must be set to the processid of the handler process.

The result is zero if the call failed, in which case the Res2 field provides more information on the failure and the FileHandle should be released.

Open New File

```
Type      LONG  Action.FindOutput (1006)

Arg1      BPTR  FileHandle
Arg2      BPTR  Lock
Arg3      BSTR  Name

Res1      LONG  Boolean
```

Arguments as for previous entry.

Read

```
Type      LONG  Action.Read (82)

Arg1      BPTR  FileHandle Arg1
Arg2      APTR  Buffer
Arg3      LONG  Length

Res1      LONG  Actual Length
```

To read from a file handle, the process id is extracted from the ProcessID field of the file handle, and the Arg1 field from the handle is placed in the Arg1 field of the packet. The buffer address and length are then placed in the other two argument fields. The result indicates the number of characters read - see the function Read for more details. An error is indicated by returning -1 whereupon the Res2 field contains more information.

Write

```
Type      LONG  Action.Write (87)

Arg1      BPTR  FileHandle Arg1
Arg2      APTR  Buffer
Arg3      LONG  Length

Res1      LONG  Actual Length
```

The arguments are the same as those for Read. See the Write function for details of the result field.

Close

```
Type      LONG  Action.End (1007)

Arg1      BPTR  FileHandle Arg1

Res1      LONG  TRUE
```

You use this packet type to close an open file handle. The process id of the handler is obtained from the file handle. The function normally returns TRUE. After a file handle has been closed, the space associated with it should be returned to the free pool.

Seek

```
Type      LONG  Action.Seek (1008)

Arg1      BPTR  FileHandle Arg1
Arg2      LONG  Position
Arg3      LONG  Mode

Res1      LONG  OldPosition
```

This packet type corresponds to the SEEK call. It returns the old position, or -1 if an error occurs. The process id is obtained from the file handle.

WaitChar

```
Type      LONG  Action.WaitChar (20)

Arg1      LONG  Timeout

Res1      LONG  Boolean
```

This packet type implements the WaitForChar function. You must send the packet to a console handler process, with the timeout required in Arg1. The packet returns when either a character is waiting to be read, or when the timeout expires. If the result is TRUE, then at least one character may be obtained by a subsequent READ.

ExamineObject

```
Type      LONG  Action.ExamineObject (23)

Arg1      BPTR  Lock
Arg2      BPTR  FileInfoBlock

Res1      LONG  Boolean
```

This packet type implements the Examine function. It extracts the process id of the handler from the ProcessID field of the lock. If the lock is zero, then it uses the default file handler, which is kept in the FiHand field of the process. The result is zero if it fails, with more information in Res2. The FileInfoBlock returns with the name and comment fields as BSTRs.

ExamineNext

```
Type      LONG  Action.ExamineNext (24)

Arg1      BPTR  Lock
Arg2      BPTR  FileInfoBlock

Res1      LONG  Boolean
```

This call implements the ExNext function, and the arguments are similar to those for Examine above. Note that the BSTR representing the filename must not be disturbed between calls of ExamineObject and different calls to ExamineNext, as it uses the name as a place saver within the directory being examined.

DiskInfo

```
Type      LONG  Action.DiskInfo (25)

Arg1      BPTR  ParameterBlock

Res1      LONG  TRUE
```

This implements the Info function. A suitable lock on the device would normally obtain the process id for the handler. This packet can also be sent to a console handler process, in which case the Volume field in the ParameterBlock contains the window pointer for the window opened on your behalf by the console handler.

Parent

```
Type      LONG  Action.Parent (29)

Arg1      BPTR  Lock

Res1      LONG  ParentLock
```

This packet returns a lock representing the parent of the specified lock, as provided by the ParentDir function call. Again it must obtain the process id of the handler from the lock, or from the Fihand field of the current process if the lock is zero.

DeleteObject

```
Type      LONG  Action.DeleteObject (16)

Arg1      BPTR  Lock
Arg2      BSTR  Name

Res1      LONG  Boolean
```

This packet type implements the Delete function. It must obtain the lock from a call to IoErr() immediately following a successful call to DeviceProc which returns the process id. The lock actually refers to the directory owning the object to be deleted, as in the the Open New and Open Old requests.

CreateDir

```
Type      LONG  Action.CreateDir (22)

Arg1      BPTR  Lock
Arg2      BSTR  Name

Res1      BPTR  Lock
```

This packet type implements the CreateDir function. Arguments are the same as for DeleteObject. The result is zero or a lock representing the new directory.

LocateObject

```
Type      LONG  Action.LocateObject (8)

Arg1      BPTR  Lock
Arg2      BSTR  Name
Arg3      LONG  Mode

Res1      BPTR  Lock
```

This implements the Lock function and returns the lock or zero. Arguments as for CreateDir with the addition of the Mode as arg3.

CopyDir

```
Type      LONG  Action.CopyDir (19)

Arg1      BPTR  Lock

Res1      BPTR  Lock
```

This implements the DupLock function. If the lock requiring duplication is zero, then the duplicate is zero. Otherwise, the process id is extracted from the lock and this packet type sent. The result is the new lock or zero if an error was detected.

FreeLock

```
Type      LONG  Action.FreeLock (15)

Arg1      BPTR  Lock

Res1      LONG  Boolean
```

This call implements the UnLock function. It obtains the process id from the lock. Note that freeing the zero lock takes no action.

SetProtect

```
Type      LONG  Action.SetProtect (21)

Arg1      Not used
Arg2      BPTR  Lock
Arg3      BSTR  Name
Arg4      LONG  Mask

Res1      LONG  Boolean
```

This implements the SetProtection function. The lock is a lock on the owning directory obtained from DeviceProc as described for DeleteObject above.

SetComment

```
Type      LONG  Action.SetComment (28)

Arg1      Not used
Arg2      BPTR  Lock
Arg3      BSTR  Name
Arg4      BSTR  Comment

Res1      LONG  Boolean
```

This implements the SetComment function. Arguments as for SetProtect above, except that arg4 is a BSTR representing the comment.

RenameObject

```
Type      LONG  Action.RenameObject (17)

Arg1      BPTR  FromLock
Arg2      BPTR  FromName
Arg3      BPTR  ToLock
Arg4      BPTR  ToName

Res1      LONG  Boolean
```

This implements the Rename function. It must contain an owning directory lock and a name for both the source and the destination. The owning directories are obtained from DeviceProc as mentioned under the entry for DeleteObject.

Inhibit

```
Type      LONG  Action.Inhibit (31)

Arg1      LONG  Boolean

Res1      LONG  Boolean
```

This packet type implements a filing system operation that is not available as an AmigaDOS call. The packet contains a Boolean value indicating whether the filing system is to be stopped from attempting to verify any new disks placed into the drive handled by that handler process. If the Boolean is true, then you may swap disks without the filesystem process attempting to verify the disk. While disk change events are inhibited, the disk type is marked as "Not a DOS disk" so that other processes are prevented from looking at the disk.

If the Boolean is false, then the filesystem reverts to normal after having verified the current disk in the drive.

This request is useful if you wish to write a program such as DISKCOPY where there is much swapping of disks that may have a half completed structure. If you use this packet request then you can avoid having error messages from the disk validator while it attempts to scan a half completed disk.

RenameDisk

```
Type      LONG  Action.RenameDisk (9)

Arg1      BPTR  NewName

Res1      BPTR  Boolean
```

Again, this implements an operation not normally available through a function call. The single argument indicates the new name required for the disk currently mounted in the drive handled by the file system process where the packet is sent. The volume name is altered both in memory and on the disk.

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4