Amiga Binary File Structure

Compiled by Paul René Jørgensen, paulrene@gmail.com

Last updated: 19. April, 2005

Introduction

This document details 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 document explains the use of common symbols, absolute external references, and program units.

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 16Mb, 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 byte size, word, or longword; 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, ir data block. 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). A BSS hunk may contain external definitions but no external references nor any values requiring relocation. If you place initialized data 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 to provide symbolic debugging information. They may also contain 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 resident 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 that are brought into memory as required.

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 to 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.

HUNK_UNIT (999/$3E7)

This block 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 diagrammatic form, the format is as follows:

```
-------------------
|    hunk_unit    |
|-----------------|
|        N        |
|-----------------|
|        N        |
|    longwords    |
|       of        |
|      name       |
:                 :
-------------------
```

HUNK_NAME (1000/$3E8)

This block 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 [(PC)] 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:

```
 -------------------
 |    hunk_name    |
 |-----------------|
 |        N        |
 |-----------------|
 |        N        |
 |    longwords    |
 |       of        |
 |      name       |
 :                 :
 -------------------
```

HUNK_CODE (1001/$3E9)

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

```
 -------------------
 |    hunk_code    |
 |-----------------|
 |        N        |
 |-----------------|
 |        N        |
 |    longwords    |
 |       of        |
 |      code       |
 :                 :
 -------------------
```

HUNK_DATA (1002/$3EA)

This block defines a block of initialized data that 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:

```
 -------------------
 |    hunk_data    |
 |-----------------|
 |        N        |
 |-----------------|
 |        N        |
 |    longwords    |
 |       of        |
 |      data       |
 :                 :
 -------------------
```

HUNK_BSS

This block specifies a block of uninitialized workspace that is allocated by the loader. The HUNK_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:

```
 -------------------
 |    hunk_data    |
 |-----------------|
 |        N        |
 -------------------
```

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. A HUNK_CODE contains executable machine language. A HUNK_DATA contained initialized data (constants, etc.) and a HUNK_BSS contains uninitialized data (arrays, variables, etc.). For these three hunk types, the size longword of the hunk is interpreted in a special way based on the two most significant bits:

```
+--------------Bit 31 MEMF_FAST
| +------------Bit 30 MEMF_CHIP
```

`0 0`	If neither bit is set, then the loader gets whatever memory is available (this is backwards compatible). Preference is given to Fast memory.
`1 0`	Loader must use Fast memory or fail.
`0 1`	Loader must use Chip memory or fail.
`1 1`	If bit 31 and bit 30 are both set then there is extra information available in the next longword. The lower 24 bits of the next longword are passed as a type to AllocMem(), the upper 8 bits are reserved for future expansion (not used currently).

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:

```
    -------------------
    |   HUNK_RELOC32  |
    |-----------------|
    |       N1        |
    |-----------------|
    |  Hunk Number 1  |
    |-----------------|
    |       N1        |
    |     Offsets     |
    :                 :
    -------------------

    -------------------
    |       N2        |
    |-----------------|
    |  Hunk Number 2  |
    |-----------------|
    |       N2        |
    |     Offsets     |
    :                 :
    -------------------

    -------------------
    |       Nn        |
    |-----------------|
    |  Hunk Number n  |
    |-----------------|
    |       Nn        |
    |     Offsets     |
    :                 :
    |-----------------|
    |        0        |
    -------------------
```

HUNK_RELOC32SHORT (1020/$3FC)

A HUNK_RELOC32short specifies 32-bit relocation that the linker is to perform within the current relocatable block using 16-bit quantities. It has the same format as a HUNK_RELOC32 (that is, the fields to be modified are 32 bits long), but the actual offsets and hunk numbers are 16 bits wide to save space, and make loading faster.

This is a more efficient way of encoding the relocation information in a file (HUNK_RELOC32s mostly consist of 0s, since almost all hunks are less than 64K long) and serves as an alternative to HUNK_RELOC32 for the final output of a linker. This is a new hunk available in V2.0 and later versions of AmigaDOS only.

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 [(PC)] 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.

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 [(PC)] 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.

HUNK_DRELOC32 (1015/$3F7)

A hunk_dreloc32 block specifies 32-bit data section relative relocation that the linker is to perform within the current block. This hunk type is used to implement base-relative addressing on the Amiga. The linker adds the offset of the base of the specified hunk (that is, the number of bytes from the base of hunk "_MERGED" to the base of the specified hunk) to each of the longwords in the preceding relocatable block that the list of offsets indicates. The specified link must be merged with the data hunk named "_MERGED". The hunk format is identical to "hunk_reloc32".

HUNK_DRELOC16 (1016/$3F8)

A hunk_dreloc16 block specifies 16-bit data section relative relocation that the linker is to perform within the current block. Except for relocation size, this block is identical to "hunk_dreloc32".

HUNK_DRELOC8 (1017/$3F9)

A hunk_dreloc8 block specifies 8-bit data section relative relocation that the linker is to perform within the current block. Except for relocation size, this block is identical to "hunk_dreloc32".

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:

```
------------------
|     hunk_ext    |
|-----------------|
|      Symbol     |
|       Data      |
|       Unit      |
|-----------------|
|      Symbol     |
|       Data      |
|       Unit      |
------------------

------------------
|        0        |
------------------
```

Where there is one "symbol data unit" for each symbol used, and the blockends with a zero word.

Each symbol data unit consists of a type byte, the symbol name length (3 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 zeroes.

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

At the time of writing, the values are as follows:

External symbols

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	32-bit reference to symbol
ext_common	130	32-bit reference to COMMON
ext_ref16	131	16-bit reference to symbol
ext_ref8	132	8-bit reference to symbol
ext_dref32	133	32-bit base relative reference to symbol
ext_dref16	134	16-bit base relative reference to symbol
ext_dref8	135	8-bit base relative reference to symbol

The linker faults all other values. For ext_def there is one data word, the value f 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 nam and it copies thiss 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 rom 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 longword, or byte field and gives an error if the signed value does not fit. Relocatable 32-bit references have the symbol value added to the field and a relocation record is produced for the loader. 16- and 8-bit 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 is 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 32-bit references, which it then handles as relocatable 32-bit references. The symbol data unit formats are as follows:

EXT_DEF/ABS/RES

```
    -----------------------
    | typ | Name Length NL |
    |----------------------
    |     NL Longwords     |
    |    of Symbol Name    |
    :                      :
    |----------------------|
    |     Symbol Value     |
    -----------------------
```

EXT_REF32/16/8

```
    -----------------------
    | typ | Name Length NL |
    |----------------------
    |     NL Longwords     |
    |    of Symbol Name    |
    :                      :
    |----------------------|
    |Count of references NR|
    |----------------------|
    |     NR Longwords     |
    | of Symbol References |
    :                      :
    -----------------------
```

EXT_COMMON

```
    -----------------------
    | 130 | Name Length NL |
    |----------------------
    |     NL Longwords     |
    |    of Symbol Name    |
    :                      :
    |----------------------|
    | Size of Common Block |
    |----------------------|
    |Count of references NR|
    |----------------------|
    |     NR Longwords     |
    | of Symbol References |
    :                      :
    -----------------------
```

HUNK_SYMBOL (12108/$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 symbol is the offset of the symbol from the start of the hunk. Thus the format is as follows:

```
------------------------
|      hunk_symbol     |
|-----------------------
|        Symbol        |
|         Data         |
|         Unit         |
------------------------
:                      :
------------------------
|          0           |
------------------------
```

Where each symbol data unit has the following format:

```
------------------------
|  0 | Name Length NL  |
|-----------------------
|     NL Longwords     |
|       of Symbol      |
|         Name         |
:                      :
------------------------
|     Symbol Value     |
------------------------
```

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:

```
------------------------
|      hunk_debug      |
|----------------------|
|          N           |
|----------------------|
|          N           |
|       Longwords      |
|          of          |
|      Debug Data      |
:                      :
------------------------
```

HUNK END (1010/$3F2)

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

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 with 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.

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.

The format of the hunk_header is described [below]. 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 longword 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 [that] 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 [that] 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.

```
 -----------------------
|      hunk_header      |
|-----------------------|
|          N1           |
|-----------------------|
|     N1 Longwords      |
|        of Name        |
:                       :
 -----------------------
|          N2           |
|-----------------------|
|     N2 Longwords      |
|        of Name        |
:                       :
 -----------------------
|           0           |
|-----------------------|
|      Table Size       |
|-----------------------|
|     First Hunk F      |
|-----------------------|
|      Last Hunk L      |
|-----------------------|
|       L - F + 1       |
|         Sizes         |
:                       :
 -----------------------
```

The header block continues with L - F + 1 longwords that indicate the size of each hunk that 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 that is required as they are loaded.

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:

```
 -----------------------
|     hunk_overlay      |
|-----------------------|
|      Table Size       |
|-----------------------|
|         M + 2         |
|-----------------------|
|         M + 1         |
|         Zeros         |
:                       :
 -----------------------
|        Overlay        |
|         Data          |
|         Table         |
:                       :
 -----------------------
```

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

M is the maximum level of the overlay tree used 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 0 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.

HUNK BREAK (1014/$3F6)

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

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 longword
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  #$AAAAAA,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
```

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 7$000, 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.

Amiga Library File Structure

There are two kinds of Amiga library file structures: the original format used with both ALINK and BLink, and the new indexed format used with Blink versions 7.2 and later.

The original Amiga library file structure is essentially one or more object modules concatenated together into one file. This structure has the appeal of simplicity. More object modules can be added to a library by appending them to the end of the library file.

In this format, the initial pass performed by a linker must process the library file sequentially to find the program units that it needs to link in.

Example Library File

Thus, a typical library might look as follows:

```
HUNK_UNIT,             2, "First PU"
HUNK_NAME,             3  "First Hunk"
HUNK_CODE.            20,    20 longwords of code...
HUNK_RELOC32,          3,  3, 12, 22, 48
                       2,  2,  4, 34
                       0
HUNK_EXT,      EXT_DEF|2, "FirstDef",   0
               EXT_DEF|3, "SecondDef", 38
             EXT_REF32|2, "ThirdDef",   2, 12, 48
             EXT_REF32|3, "FourthDef",  1,  4
                       0
HUNK_DEBUG,            7,    7 longwords of debugging information...
HUNK_END
HUNK_NAME,             3, "Second Hunk"
```

```
    HUNK_DATA,            30,    30 longwords of data...
    HUNK_EXT,      EXT_DEF|3, "FirstConst", 0
                   EXT_DEF|3, "FourthDef",  4
                   EXT_DEF|3, "LongString", 8
                           9
    HUNK_END

    HUNK_BSS,             40
    HUNK_EXT,      EXT_DEF|2, "workStr", 0
                           0
    HUNK_END

    HUNK_UNIT,             3, "Second PU"
    HUNK_NAME,             3, "Third Hunk"
    HUNK_CODE,            64,    64 longwords of code,
    HUNK_RELOC32,          2,  0, 14, 54
                           4,  1,  4, 22, 28, 44
                           3,  2, 10, 38, 100
                           0
    HUNK_EXT,    EXT_REF32|2, "FirstDef",    2, 14, 54
                 EXT_REF32|3, "LongString",  3, 22, 28, 44
                   EXT_DEF|2, "ThirdDef",    0
                           0
    HUNK_END
```

The New Library File Structure

The new library file format is very much like the old, except that there is an extra level of encapsulation, through the use of two new hunk types. Users may still merge libraries by simply concatenating files and old or new format libraries can be appended together.

The new format is more compact and faster for the linker to process. It achieves its performance and flexibility by adding two additional hunk types: hunk_lib and hunk_index. Like all basic Amiga hunk types, these consist of a longword type value, followed by a 32-bit value for the number of subsequent longwords in the hunk. Further, they always occur in pairs, hunk_lib first, hunk_index following. Nothing comes between.

HUNK_LIB (1019/$3FB)

The format of hunk_lib is shown [below].

```
    ------------------------
    |       hunk_lib       |
    |----------------------|
    |          N           |
    |----------------------|
    |      N Longwords     |
    |          of          |
    |    Contained Hunks   |
    ------------------------
```

The size field (N) of the hunk_lib structure must be a count of ALL of the longwords belonging to the structure, excluding the type and size field. Thus, the longword count given in the size field can be greater than 65,535; however, note that the offset, in longwords, to the last code, data, or bss hunk be no greater than 65,535 (see hunk_index, below). If the contained hunk (or its constituent hunks) extend beyond that point, the hunk_lib size field MUST still include them in the count.

HUNK_INDEX (1020/$3FC)

The hunk_index provides an index to all the hunks concatenated in hunk_lib. hunk_index format is shown [below].

```
-----------------
|   hunk_index  |
|---------------|
|       N       |
|---------------|
|    Size of    |
|  String Block |
|    in Bytes   |
|---------------|
|  String Block |
|   (Up to 64K  |  -----------------
|  of Strings)  |  |  16-bit Byte  |
|---------------|  |  Offset Into  |  -----------------
|    Program    |  |  String Block |  |  16-bit byte  |
|      Unit     |  |  for Program  |  |  Offset Into  |
|       1       |  |  Unit N1 Name |  |  String Block |
|---------------|  |---------------|  |to hunk name N2|
|    Program    |  |16-bit Longword|  |---------------|
|      Unit     |  |   Offset to   |  |  hunk N2 Size |
|       2       |  | First Hunk of |  |---------------|
|---------------|  | Prog. Unit N1 |  |  hunk N2 Type |
|    Program    | /|---------------|  |---------------|
|      Unit     |< |   Hunk Count  |  | hunkRef Count |
|       N1      | \|  for Program  |  |  (EXT_REF32s  |
-----------------  |    Unit N1    |  |and EXT_REF16s)|
                   |---------------|  |---------------|
                   |  Hunk Entry 1 |  |   hunkRef 1   |
                   |---------------|  |---------------|
                   |  Hunk Entry 2 |  |   hunkRef 2   |
                   |---------------| /|---------------| /-----------------
                   | Hunk Entry N2 |< |   hunkRef N3  |< |  16-bit Byte  |
                   ----------------- \|---------------|\|  Offset Into
                                      | DefEntry Count|  |  String Block |
                                      |   (EXT_DEFS,  |  | Symbol N3 Name|
                                      |      etc.)    |  -----------------
                                      |---------------|
                                      |   DefEntry 1  |
                                      |---------------|
                                      |   DefEntry 2  |
                                      |---------------| /-----------------
                                      |  DefEntry N4  |< |  16-bit Byte  |
                                      ----------------- \|  Offset Into  |
                                                         |  String Block |
                                                         | for Symbol N4 |
                                                         |      Name     |
                                                         |---------------|
                                                         |  16-bit Byte  |
                                                         |  from Base of |
                                                         |Hunk for Symbol|
                                                         |       N4      |
                                                         |---------------|
                                                         | 16-bit Type of|
                                                         |   Symbol N4   |
                                                         -----------------
```

Example of HUNK_LIB

Here's an example of a new format library, based on the previous example given in "Example Library File", above. The library is formed by pairing of hunk_lib and a hunk_index. Here's the hunk_lib:

```
    HUNK_LIB,            191,
    HUNK_CODE,            20,    20 longwords of code...
    HUNK_RELOC32,          3,  3, 12, 22, 48
                           2,  2,  4, 34
                           0
    HUNK_EXT,    EXT_REF32|2, "ThirdDef",  2, 12, 48
                 EXT_REF32|2, "FourthDef", 1, 4
                           0
    HUNK_DEBUG,            7,    7 longwords of debugging information...
    HUNK_END
    HUNK_DATA,            30,    30 longwords of data...
    HUNK_END
    HUNK_BSS,             40
    HUNK_END
    HUNK_CODE,            64,    64 longwords of code,
    HUNK_RELOC32,          2,  0, 14, 54
                           4,  1,  4, 22, 28, 44
                           3,  2, 10, 38, 100
                           0
    HUNK_EXT,    EXT_REF32|2, "FirstDef",    2, 14, 54
                 EXT_REF32|3, "LongString",  3, 22, 28, 44
                           0
    HUNK_END
```

Example of HUNK_INDEX

The hunk_index for the library is more complicated. It follows the general format:

- hunk_index
- Size
- 16-bit word aligned string block
- one or more punit structures

Where the string block consists of a 16-bit word value, representing the size of the rest of the block, in bytes, and the rest of the block consists of null-terminated (C-style) strings, where the first string must be the null string. Strings are NOT word-boundary aligned. If necessary, the block is padded on the end with a single 0 byte, to align to a word boundary. Thus the string block for the above example would resemble the following:

```
    122
    ""                     at offset  0
    "First PU"                        1
    "First Hunk"                     10
    "FirstDef"                       21
    "SecondDef"                      30
    "ThirdDef"                       40
    "FourthDef"                      49
    "Second Hunk"                    59
    "FirstConst"                     71
    "LongString"                     82
    "workStr"                        93
    "Second PU"                     101
    "Third Hunk"                    111
```

This block needed no trailing 0 byte for alignment to a 16-bit word boundary. Note that this block, excluding its length field, can be no larger than 65,534 bytes (64K - 2 bytes). The trailing pad byte, if present, is included in the size field for the block.

What follow the string block is one or more punit structures with the following format:

- Punit header
- One or more hunk entries
- If necessary, a padding 16-bit 0 value, to realign the hunk_index hunk to a longword boundary.

Where a punit header consists of:

1. A 16-bit offset of a program unit name string in preceding string block (0 is the offset to the first string; -2 is the offset to the length of the block). This offset is in bytes, and is signed. Thus, the total string space available for any one hunk_lib's symbol names is 65,534 bytes.
2. A 16-bit offset of first hunk (code, data, or bass) to a program unit within the preceding hunk_lib structure. This offset is in longwords, meaning that no hunk in the corresponding hunk_lib can begin beyond a byte offset of 262,140.
3. A 6-bit count of the number of hunks in the preceding hunk_lib structure (code, data, and bss).

And a hunk entry consists of:

1. A 16-bit offset to hunk name string in string block (or 0 - the null string).
2. The 16-bit size of the hunk, in longwords.
3. A 16-bit type of the hunk (hunk_code, hunk_data, hunk_bss), with any Fast or Chip flag settings moved into the upper 2 bits of the type word.
4. A 16-bit count of the number of references. This information is duplicated from the EXT_REFs of any hunk_ext associated with the hunk. This particular field is followed by the 16-bit string offsets of the symbols being referenced in the string block (the string itself if a 32-bit reference).
5. The 16-bit count of the number of definitions. This information is moved completely out of the hunk_exts of the hunk (which is why they are so much shorter in the example of the hunk_lib above). This field is followed by 0 or more entries of three words:
   a. A 16-bit offset to defined symbol name string in string block (most significant bit always clear).
   b. A 16-bit offset (in bytes) of symbol from base of hunk.
   c. A 16-bit type of the symbol definition. Note that type has been extended. In some instances of EXT_ABS values, most notably the _CIA references in amiga.lib, the ABS value has significant bits which take up to 25 bits to store.

Since the type field value will fit comfortably into 1 byte, the upper byte is reserved for bits 16-23 of EXT_ABS values, and bit 6 in the type byte is used to note the state of the uppermost 8 bits of the original 32-bit value of the EXT_ABS (that is, all 1s, or all 0s). This permits 25 bits' worth of EXT_ABS information to be stored in the existing structures. Thus, for EXT_ABS data, the following is the format:

```
original EXT_ABS values:  abs1 = $c709d3
                          abs2 = -14872941 ($$ff1d0e93)
resultant EXT_ABS values: abs1 = $09d3          (word)
                                 $c7            (byte)
                                 EXT_ABS        (byte)
                          abs2 = $0e93          (word)
                                 $1d            (byte)
                                 EXT_ABS | 64   (byte)
```

Note that in all hunk_index structures, a 16-bit value of 0 for the count of array elements of a given type means that NO array elements of that type are present in the structure.

Thus, the hunk_index for the above given hunk_lib is:

```
    hunk_index, 57
      122
      ""                      at offset  0
      "First PU"                         1
      "First Hunk"                      10
      "FirstDef"                        21
      "SecondDef"                       30
      "ThirdDef"                        40
      "FourthDef"                       49
      "Second Hunk"                     59
      "FirstConst"                      71
      "LongString"                      82
      "workStr"                         93
      "Second PU"                      101
      "Third Hunk"                     111
      1, 0, 3                 program unit with 3 hunks...
        10, 20, HUNK_CODE        hunk info
            2, 40, 49               2 refs...
            2, 21, 0, EXT_DEF       2 defs...
            30, 38, EXT_DEF
        59, 30, HUNK_DATA        hunk info
             0                      no refs
             3, 71, 0, EXT_DEF      3 defs...
                49, 4, EXT_DEF
                82, 4, EXT_DEF
         0, 40, HUNK_BSS         hunk info
            0                      no refs
            1, 93, 0, EXT_DEF       1 def...
       101, 92, 1             program unit with 1 hunk...
         111, 64, HUNK_CODE     hunk info
           2, 21, 82              2 refs...
           1, 40, 0, EXT_DEF      1 def...
      0                         16-bit pad for longword alignment of hunk
```

Note from the examples that the lib_lib structure still contained the hunk_ends, the hunk_reloc32s, the hunk_debug, and part of some of hunk_exts; if a hunk_symbol had been present, it would also have to be in the hunk_lib with its corresponding code, data, or bss hunk.

These hunks - hunk_code, hunk_data, hunk_bss, hunk_reloc32, hunk_reloc16, hunk_reloc8, hunk_symbol, hunk_debug, and hunk_end - must be present exactly as if they weren't in a hunk_lib. A hunk_unit, hunk_name must be removed entirely, replaced by program unit entries in the hunk_index associated with the hunk_lib. The hunk_ext must lose any EXT_DEFs, the information for which is instead found in the hunk entries in the hunk_index. EXT_REF32s and/or EXT_REF16s must be present in a hunk_ext in order for the hunk_ext to remain at all in the hunk_lib, and EXT_REF32s and EXT_REF16s must be noted as well in the reference list in the hunk entry found in the hunk_index for the hunk. EXT_REF8s are not supported.

Hunk Overlay Table – Overview

When overlays are used, the linker basically produces one very large file containing all the object modules as hunks of relocatable code. The hunk overlay table contains a data structure that describes the hunks and their relationship to each other.

When you are designing a program to use overlays, you must keep in mind how the overlay manager (also called the overlay supervisor) handles the interaction between the various segments of the file. What you must do, basically, is build a tree that reflects the relationships between the various code modules that are a part of the overall program and tell the linker how this tree should be constructed.

The hunk overlay table is generated as a set of 8 longwords, each describing a particular overlay node that is part of the overall file. Each 8 longword entry is comprised of the following data:

```
    Hunk Overlay Symbol - Table Entry Data Structure
    long seekOffset;          /* where in the file to find this node */
    long dummy1;              /* a value of 0...compatibility item */
    long dummy2;              /* a value of 0...compatibility item */
    long level;               /* level in the tree */
    long ordinate;            /* item number at that level */
    long firstHunk;           /* hunk number of the first hunk containing
                                 this node */
    long symbolHunk;          /* the hunk number in which this symbol is
                                 located */
    long symbolOffsetX;       /* (offset + 4), where offset is the offset
                                 within the symbol hunk at which this
                                 symbol's entry is located */
```

Each of these items is explained further in the sections that follow.

Designing an Overlay Tree

Let's say that you have, for example, the files main, a, b, c, d, e, f, g, h, i, and j, and that main can call a, b, c, and d and that each of these files can call main. Additionally let's say that routine e can be called from a, b, c, d, or main, but has no relationship to routine f. Thus, if a routine in e is to be run, then a, b, c, and d need to be memory-resident as well. Routine f is like e; that is, it needs nothing in e to be present, but can be called from a, b, c, or d. This means that the overlay manager can share the memory space between routines e and f, since neither need ever be memory-coresident with the other to run.

If you consider routine g to share the same space as the combination of a, b, c, and d and routines h, i, and j sharing the same space, you have the basis for constructing the overlay tree for this program structure:

```
                 main (root level of the tree)
                   |                        |
          a,b,c,d(1,1)            +-----g(1,2)-----+
          |         |             |        |       |
       e(2,1)    f(2,2)        h(2,1)  i(2,2)  j(2,3)
```

Not only have we drawn the tree, but we have also labelled its branches to match the hunk overlay (level, ordinate) numbers that are found in the hunk overlay table that matches the nodes to which they are assigned.

From the description above, you can see that if main is to call any routine in program segment a-d, then all of those segments should be resident in memory at the same time. Thus they have all been assigned to a single node by the linker. While a-d are resident, if you call routines in e, the linker will automatically load routine e from disk, and reinitialize the module (each time it is again brought in) so that its subroutines will be available to be run. If any segment a-d calls a routine in f, the linker replaces e with the contents of f and initializes it. Thus a-d are at level 1 in the overlay tree, and routines e and f are at level 2, requiring that a-d be loaded before e or f can be accessed and loaded for execution.

Note: A routine can only perform calls to routines in other nodes that either are currently memory-resident (the ancestors of the node in which the routine now in use is located), or a routine in a direct child node. That is, main cannot call e directly, but e can call routines in main since main is an ancestor.

Note also that within each branch of each sub-node, the ordinate numbers begin again with number 1 for a given level.

Describing the Tree

You create the tree by telling the overlay linker about its structure. The numerical values, similar to those noted in the figure above, are assigned sequentially by the linker itself and appear in the hunk node table. Here is the sequence of overlay link statements that cause the figure above to be built:

```
OVERLAY
a,b,c,d
*e
*f
g
*h
*i
*j
```

This description tells the linker that a, b, c, [and] d are part if a single node at a given level (in this case level 1), and the asterisk in front of e and f each say that these are one each on the next level down from a-d, and accessible only through a-d or anything closer towards the root of the tree. The name g has no asterisk, so it is considered [to be] on the same level as a-d, telling the linker that either a-d or g will be resident, but not both simultaneously. Names h, i, and j are shown to be related to g, one level down.

The above paragraphs have explained the origin of the hunk node level and the hunk ordinate in the hunk overlay symbol table.

seekOffset Amount

The first value for each node in the overlay table is the seek offset. As specified earlier, the overlay linker builds a large single file containing all of the overlay nodes. The seek offset number is that value that can be given to the Seek(file, byte_offset) routine to point to the first byte of the hunk header of a node.

firstHunk

The firstHunk value in the overlay symbol table is used by the overlay manager when unloading a node. It specifies the initial hunk that must have been loaded in order to have loaded the node that contains this symbol. When a routine is called at a different level and ordinate (unless it is a direct, next level, child of the current node), it is necessary to free the memory utilized by invalid hunks, so as to make room to overlay with the hunk(s) containing the desired symbol.

symbolHunk and symbolOffsetX

These table entries for the symbols are used by the overlay manager to actually locate the entry point once it has either determined it is already loaded or has loaded it. The symbolHunk shows in which hunk to locate the symbol. symbolOffsetX - 4 shows the offset from the start of that hunk at which the entry point is actually located.

Overlay Nodes and the Linker

While linking an overlaid program, the linker checks each symbol reference for validity. Suppose that the reference is in a tree node R, and the symbol is in a node S. Then the reference is legal if one of the following is true:

1. R and S are the same node.
2. R is a descendant of S.
3. R is the parent of S.

References of the third type above are known as overlay references. In this case, the linker enters the overlay supervisor when the program is run. The overlay supervisor then checks to see if the code segment containing the symbol is already in memory. If not, first the code segment, if any, at this level, and all its descendents are unloaded, and then the node containing the symbol is brought into memory. An overlaid code segment returns directory to its caller, and so is not unloaded from memory until another node is loaded on top of it.

For example, suppose that [this] is the tree:

```
    A
   / \\
  B   C
 /|\\
D E F
```

When the linker first loads the program, only A is in memory. When the linker finds a reference in A to a symbol in B, it loads and enters B. If B, in turn, calls D then again a new node is loaded. When B returns to A, both B and D are left in memory, and the linker does not reload them if the program requires them later. Now suppose that A calls C. First the linker unloads the code segments that it does not require, and which it may overwrite. In this case, these are B and D. Once it has reclaimed the memory for these, the linker can load C.

Thus, when the linker executes a given node, all the node's "ancestors",up to the root, are in memory, and possible some of its descendents.

The linker assumes that all overlay references are jumps or subroutine calls, and routes them the overlay supervisor. Thus, you should not use overlay symbols as data labels.

Try to avoid impure code when overlaying because the linker does not always load a node that is fresh from the load file.

The linker gives each symbol that has an overlay reference an overlay number. It uses this value, which is zero or more, to construct the overlay supervisor entry label associated with that symbol. This label is of the form "OVLYnnnn", where nnnn is the overlay number. You should not use symbols with this format elsewhere.

The linker gathers together all program sections with the same section name. It does this so that it can load them continuously in memory.

Delete all the material on "ATOM: (Alink Temporary Object Modifier)". It describes a product no longer sold or supported by Commodore.

Example code: Dumps Amiga Load Files.rexx

```
/*************************************************\
*       dalf.rexx - Dumps Amiga Load Files.       *
* C 1990 Mikael Karlsson (m...@slaka.sirius.se) *
\*************************************************/

parse arg file              /* File to examine */

signal on break_c           /* We want a nice clean break */

pl. = "s"                   /* This is how to handle plurals the ince way */
pl.1 = ""

temp = '00'x
flagtext.temp = ""
temp = '40'x                /* Bit 30 means 'Load to CHIPMEM' */
flagtext.temp = " (CHIP)"
bits. = '00'x

type. = "Unknown"

/* These are the hunk types we know about (so far) */

Hunk_unit    = '03E7'x; type.Hunk_unit    = "Hunk_unit    "
Hunk_name    = '03E8'x; type.Hunk_name    = "Hunk_name    "
Hunk_code    = '03E9'x; type.Hunk_code    = "Hunk_code    "
Hunk_data    = '03EA'x; type.Hunk_data    = "Hunk_data    "
Hunk_bss     = '03EB'x; type.Hunk_bss     = "Hunk_bss     "
Hunk_reloc32 = '03EC'x; type.Hunk_reloc32 = "Hunk_reloc32 "
Hunk_reloc16 = '03ED'x; type.Hunk_reloc16 = "Hunk_reloc16 "
Hunk_reloc8  = '03EE'x; type.Hunk_reloc8  = "Hunk_reloc8  "
Hunk_ext     = '03EF'x; type.Hunk_ext     = "Hunk_ext     "
Hunk_symbol  = '03F0'x; type.Hunk_symbol  = "Hunk_symbol  "
Hunk_debug   = '03F1'x; type.Hunk_debug   = "Hunk_debug   "
Hunk_end     = '03F2'x; type.Hunk_end     = "Hunk_end     "
Hunk_header  = '03F3'x; type.Hunk_header  = "Hunk_header  "
Hunk_overlay = '03F5'x; type.Hunk_overlay = "Hunk_overlay "
Hunk_break   = '03F6'x; type.Hunk_break   = "Hunk_break   "
Hunk_drel32  = '03F7'x; type.Hunk_drel32  = "Hunk_drel32  "
Hunk_drel16  = '03F8'x; type.Hunk_drel16  = "Hunk_drel16  "
Hunk_drel8   = '03F9'x; type.Hunk_drel8   = "Hunk_drel8   "
Hunk_libhunk = '03FA'x; type.Hunk_libhunk = "Hunk_libhunk "
Hunk_libindx = '03FB'x; type.Hunk_libindx = "Hunk_libindx "

/* These are subtypes in Hunk_ext */

Hunk_def     =   '01'x; type.Hunk_def     = "Hunk_def    "
Hunk_abs     =   '02'x; type.Hunk_abs     = "Hunk_abs    "
Hunk_res     =   '03'x; type.Hunk_res     = "Hunk_res    "
Hunk_ext32   =   '81'x; type.Hunk_ext32   = "Hunk_ext32  "
Hunk_ext16   =   '83'x; type.Hunk_ext16   = "Hunk_ext16  "
Hunk_ext8    =   '84'x; type.Hunk_ext8    = "Hunk_ext8   "
Hunk_dext32  =   '85'x; type.Hunk_dext32  = "Hunk_dext32 "
Hunk_dext16  =   '86'x; type.Hunk_dext16  = "Hunk_dext16 "
Hunk_dext8   =   '87'x; type.Hunk_dext8   = "Hunk_dext8  "

if ~open(lf, file, 'R') then do     /* Open load file */
    say "Can't open" file
    exit 10
end

index = 0
size. = 0

loop:
    type = readch(lf, 4)            /* Read hunk type */
    if type == "" then do           /* End of file */
        signal done
    end
    bits.index = bitor(bits.index, left(type, 1))   /* Check flag bits */
    type = right(type, 2)                           /* Remove flag bits */
    if type.type = "Unknown"  then do
        say "Unknown hunk type ("c2x(type)")"
        exit 10
    end
    id = type.type "("c2x(type)")"
    signal value trim(type.type)    /* Jump to hunk display routine */

Hunk_header:
    say id
    dummy = c2d(readch(lf, 4))         /* What's this? */
    count = c2d(readch(lf, 4))
    low   = c2d(readch(lf, 4))
    high  = c2d(readch(lf, 4))
    say "      "count "hunk"pl.count "("low "to" high")"
    do i=low to high
        size = readch(lf, 4)
        bits.i = left(size, 1)
        size.i = c2d(right(size, 3))*4
        bits = bits.i
        say "      Size hunk" i":" size.i "bytes" flagtext.bits
    end
    index = low
    signal loop

Hunk_end:
    say "    "id
    signal loop

Hunk_code:
    size = readch(lf, 4)
    bits = bitor(bits.index, left(size, 1))
    size = c2d(right(size, 3))*4
    temp = right(index, 2)
    temp = temp":" id
    temp = temp "("size "bytes)"flagtext.bits
    say temp
    do while size>32768
        data = readch(lf, 32768)
        size = size-32768
    end
    data = readch(lf, size)
    index = index+1
    signal loop

Hunk_reloc32:
Hunk_reloc16:
Hunk_reloc8:
    say "    "id
    count = c2d(readch(lf, 4))
    do while count~=0
        ref = c2d(readch(lf, 4))
        say "      "count "item"pl.count "for hunk" ref
        refs = readch(lf, count*4)
        count = c2d(readch(lf, 4))
    end
    signal loop

Hunk_ext:
    say "    "id
    sym_type = readch(lf, 1)
    sym_length = c2d(readch(lf, 3))*4
    do until sym_type == "00"x
        symbol = strip(readch(lf, sym_length), 'T', '00'x)
        select
            when sym_type == hunk_def |,
                 sym_type == hunk_abs |,
                 sym_type == hunk_res then do
                offset = strip(c2x(readch(lf, 4)), 'T', '00'x)
                temp = "     " type.sym_type
                temp = temp left(symbol, 32)":" "0x"offset
                say temp
            end
            when sym_type == hunk_ext32  |,
                 sym_type == hunk_ext16  |,
                 sym_type == hunk_ext8   |,
                 sym_type == hunk_dext32 |,
                 sym_type == hunk_dext16 |,
                 sym_type == hunk_dext8 then do
                count = c2d(readch(lf, 4))
                refs = readch(lf, count*4)
                temp = "     " type.sym_type
                temp = temp left(symbol, 32)":"
                temp = temp right(count, 2) "item"pl.count
                say temp
            end
            otherwise do
                say "      Unknown definition"
            end
        end
        sym_type = readch(lf, 1)
        sym_length = c2d(readch(lf, 3))*4
    end
    signal loop

Hunk_drel32:
Hunk_drel16:
Hunk_drel8:
    say "    "id
    count = c2d(readch(lf, 4))
    do while count~=0
        ref = c2d(readch(lf, 4))
        say "      "count "item"pl.count "for hunk" ref
        refs = readch(lf, count*4)
        count = c2d(readch(lf, 4))
    end
    signal loop

Hunk_data:
    size = readch(lf, 4)
    bits = bitor(bits.index, left(size, 1))
    size = c2d(right(size, 3))*4
    temp = right(index, 2)
    temp = temp":" id
    temp = temp "("size "bytes"
    if size.index-size>0 then do
        temp = temp"," size.index-size "BSS"
    end
    temp = temp")"flagtext.bits
    say temp
    data = readch(lf, size)
    index = index+1
    signal loop

Hunk_bss:
    size = readch(lf, 4)
    bits = bitor(bits.index, left(size, 1))
    size = c2d(right(size, 3))*4
    temp = right(index, 2)
    temp = temp":" id
    temp = temp "("size "bytes)"flagtext.bits
    say temp
    index = index+1
    signal loop

Hunk_unit:
Hunk_name:
    say right(index, 2)":"id
    size = c2d(readch(lf, 4))*4
    data = readch(lf, size)
    say "    " type.type":" data
    index = index+1
    signal loop

Hunk_symbol:
    say right(index, 2)":"id
   size = c2d(readch(lf, 4))*4
    do while size~=0
        data = strip(readch(lf, size), 'T', '00'x)
        say "    " left(data, 32)":" c2x(readch(lf, 4))
        size = c2d(readch(lf, 4))*4
    end
    signal loop

Hunk_libhunk:
    size = readch(lf, 4)
    bits = bitor(bits.index, left(size, 1))
    size = c2d(right(size, 3))*4
    say "    "id "("size "bytes)"flagtext.bits
    signal loop

Hunk_libindx:
    size = c2d(readch(lf, 4))*4
    say "    "id "("size "bytes)"
    count = c2d(readch(lf, 2))
    say "    " count "bytes in string block"
    string = readch(lf, count)
    do forever
        nameoffset = c2d(readch(lf, 2))
        if nameoffset=0 then leave
        parse value substr(string, nameoffset+1) with name "00"x .
        say "    PUNIT '"name"'"
        unitoffset = c2d(readch(lf, 2))
        say "     offset" unitoffset "longword"pl.unitoffset
        hunkcount = c2d(readch(lf, 2))
        say "    " hunkcount "hunk"pl.hunkcount
        do for hunkcount
            nameoffset = c2d(readch(lf, 2))
            parse value substr(string, nameoffset+1) with name "00"x .
            hunksize = c2d(readch(lf, 2))
            hunktype = readch(lf, 2)
            say "    " type.hunktype "'"name"' of" hunksize "longword"pl.hunksize
            refcount = c2d(readch(lf, 2))
            say "    " refcount "ref"pl.refcount":"
            do for refcount
                nameoffset = c2d(readch(lf, 2))
                if substr(string, nameoffset+1, 1)="00"x then do
                    nameoffset = nameoffset+1
                    temp = "16"
                end
                else do
                    temp = "32"
                end
                parse value substr(string, nameoffset+1) with name "00"x .
                say "      " temp"-bit ref '"name"'"
            end
            defcount = c2d(readch(lf, 2))
            say "    " defcount "def"pl.defcount":"
            do for defcount
                nameoffset = c2d(readch(lf, 2))
                parse value substr(string, nameoffset+1) with name "00"x .
                defoffset = readch(lf, 2)
                defdata = readch(lf, 2)
                deftype = c2d(right(defdata, 2))
                defdata = left(defdata, 2)
                select
                    when deftype=1 then do
                        say "       Define def '"name"' at" c2d(defoffset)
                    end
                    when deftype=2 then do
                        defoffset = defdata || defoffset
                        say "       Define abs '"name"' at" c2d(defoffset)
                    end
                    when deftype=3 then do
                        say "       Define res '"name"' at" c2d(defoffset)
                    end
                    when deftype=66 then do
                        defoffset = "FF"x || defdata || defoffset
                        say "       Define abs '"name"' at" c2d(defoffset)
                    end
                    otherwise do
                        say "Error in object file"
                        exit 10
                    end
                end
            end
        end
    end
    signal loop

Hunk_debug:
    size = c2d(readch(lf, 4))*4
    say "    "id "("size "bytes)"
    say "       Offset:" c2d(readch(lf, 4))
    say "       Type:  " readch(lf, 4)
    data = readch(lf, size-8)
    signal loop

Hunk_break:
    size = c2d(readch(lf, 4))*4
    say "    "id "("size "bytes)"
    data = readch(lf, size)
    index = index+1
    signal loop

Hunk_overlay:
    size = c2d(readch(lf, 4))*4
    say "    "id "("size "bytes) - Not supported"
    data = readch(lf, size)
    index = index+1
    signal loop


break_c:
done:
exit 0
```