Chapter 8

# THE DSI-32 LINKER

## 8.1 INTRODUCTION

LN (the DSI-32 Linker) takes object modules generated by AS (the DSI-32 assembler) and produces an executable file that can be loaded onto the DSI-32 Coprocessor board and executed. The linker supports the National Semiconductor 32000 module table scheme of linking.

The features of the LN object linker are:

- Complete set of directives* available for easy and efficient linking
- Linker can be driven interactively or by command file
- Full control of memory allocation by directives
- Support of "ROM" modules
- Library module support
- Optional Link Map can be generated by directive

The programmer has complete control over the placement of code, linkage tables, static data areas and the user stack. Each module's memory allocation can be assigned to a global location or on a module by module basis. Initialized COMMON data areas are supported as well.

## 8.2 INPUT AND OUTPUT FILES USED AND GENERATED BY LINKER

LN links the user's object modules produced by AS and generates an executable file. An optional map file may be generated by issuing the MAP directive to the linker.

Object file - Input. The object code and associated tables generated by AS are in this file. The default filename extension for object files is .O32.

Indirect file - Input. The indirect (command) file contains a list of linker directives used to drive the linker without requiring interactive input. The directives supported by the indirect file are the same as those used in interactive mode, except that embedded indirect commands are not supported. See section 8.2.2 for a discussion of interactive and indirect modes of input.

Executable file - Output. The linked code is placed in an executable file which may then be loaded onto the DSI-32 Coprocessor board by the Loader. The default extension for executable files is .E32.

Map file - Output. If the MAP directive is given to the linker, the linker map information is placed in a text file called a map file. The default extension for map files is .M32.

Errors and Warnings - Output. Errors detected by the linker, whether operating in interactive or indirect (command file) mode are sent to the console immediately after the error occurs. In

---

* - for a full discussion of directives, see section 8.6

indirect mode, error messages are displayed after the directive detecting the error has been echoed to the console. Some directives will cause the linker to exit to the operating system if an error is detected, whereas other directive errors will return to the linker prompt for further directives.

## 8.3 LINKER INVOCATION

The Linker may be invoked in an interactive mode or command line mode. Interactive mode allows the user to specify several directives, one at a time to the linker. As it processes each directive it immediately report any errors. Command line mode allows the user to list on the operating system command line all the necessary directives for linking.

Interactive mode is entered by simply typing "LN" after the system prompt. Command line mode in invoked by specifying the link directives immediately after "LN". Both modes use the same directives and require the same format for each directive.

When invoked in interactive mode, the linker displays a LN> prompt on the screen and waits for the user to start entering directives. The user then enters the directives, one directive at a time followed by a carriage return (<CR>), until the full set of directives necessary for linking have been specified. If a directive is improperly specified, or a directive conflicts with a previously entered directive, the linker will display an error message immediately after the directive has been entered. When all directives have been entered, a blank line (i.e. a line containing only a <CR>) instructs the linker to link the modules specified under the constraints of the directives. If all necessary directives have not been specified, the linker will display an error message reporting that fact. Depending on the severity of the error, the linker will return to the LN> prompt, or return to the operating system.

As an example of interactive mode, the following is a session using LN to link three modules named a.o32, b.o32 and c.o32 producing the executable file named d.e32:

```
C>LN
DSI-32 Linker V1.10
Copyright (c)1985 Definicon Systems Inc.
LN> FILE=a.o32,b.o32
LN> FILE=c.o32
LN> EXEC=d.e32
LN> RAM=0..fffff
LN>

C>
```

Whether invoked in command line mode or interactive mode, each directive specified on the command or prompt line must be separated by blanks. Directives themselves, however, must not contain blanks. For example, to link the same modules in the above example in command line mode, the following MS-DOS command line is used:

```
C>LN FILE=a.o32,b.o32,c.o32 EXEC=d.e32 RAM=0..fffff
DSI-32 Linker V1.10
Copyright (c)1985 Definicon Systems Inc.

C>
```

Note that the single directive FILE, though it refers to three module names, does not contain embedded blanks, whereas a blank is necessary to separate the FILE directive from the EXEC directive. This requirement is the same in both indirect and interactive modes.
In both interactive and command line mode (unless the NODEFAULT directive is used), when directed to link, the linker will first look for a default file which is specific to the programming language for the link session. Four language types are supported by LN, the default filenames for the language types are listed below:

1) C language LNC.D32

2) Pascal language LNP.D32

3) Fortran language LNF.D32

4) Assembler language LNA.D32

The default file normally contains default libraries to search for a given language type and any hardware specific directives such as the amount of RAM, ROM, etc. If this file is not found, the linker issues a warning message. The user should place directives in this file that are invariant for each invocation of the linker. Directives in the default file are echoed to the console (in the same manner as indirect files, see section 8.7).

The minimum required set of directives for linking consists of the following:

- one or more FILE directives specifying all files to be linked
- one or more LIBRARY directives specifying all libraries (if any) necessary for linking
- a RAM directive specifying total system memory

## 8.4 RESOLVING EXTERNAL REFERENCES

LN resolves external references by searching each module specified by the FILE_LIST directive and then searching each module specified by the LIBRARY directive for symbols that match outstanding unresolved symbols. External symbols are specified in the assembler source code with IMPORT/IMPORTP and EXPORT/EXPORTP pseudo-ops. A match if found when an IMPORTed symbol has the same name as an EXPORTed symbol and their types are consistent. The following table defines consistency of import/export symbol types.

| IMPORT type | EXPORT type |
|---|---|
| Data | SB Data, ABS Data, PC Data |
| Entry Point | PC Entry, Local Entry |

If a symbol is unresolved after searching both the user modules and libraries, LN will continue processing the remaining modules to generate a map but will not generate an executable file. An error message will list all unresolved external references at the end of processing.

## 8.5 MEMORY ALLOCATION

After all external references have been resolved, LN begins to allocate memory for the executable file. The following structures require either automatic or user specified memory allocation:

- the global module table
- code blocks for each module
- link tables for each module
- static base data blocks for each module
- the stack
- the heap

By default, each structure is automatically allocated by LN using a contiguous allocation scheme. LN places the structures in the lowest available RAM address in the order listed above. If the default allocation scheme is not optimal for a particular program, it can be overridden by explicit specification of allocation addresses using the memory allocation directives.

Allocation of structures for a particular module, independent of the global specifications defined by default or by memory allocation directives, is handled by the MODULE directive. This directive specifies certain allocation requirements that are not necessarily addressed by a global allocation scheme, for example COMMON memory in a FORTRAN program.

If LN is unable to allocate a specified structure due to size limitations, an error message will be sent to the console and control will return to the operating system. The user does not have control over which order the structures are allocated.

Due to the architecture of the 32000 microprocessor series, the Global Module Table must be allocated in the lower 64k bytes of RAM. The stack is allocated a default 4096 bytes unless otherwise specified, the remaining memory is allocated to the heap. Note that it may be necessary to allocate more stack for certain programs. None of the structures can be allocated over non-contiguous blocks of memory and each structure starts at a double word boundary.

## 8.6 DIRECTIVES

Directives provide the linker with the information necessary to successfully create an executable file. The format of a directive is:

```
dir=spec
```

where dir is one of LN's directive keywords and spec is variable information that is significant in the context of the directive. Note that no spaces are allowed between any parts of the directive.

There are five forms of directive:

- INPUT
- INDIRECT FILE
- MEMORY ALLOCATION
- OUTPUT
- LINKER CONTROL

Directives fall into two categories depending on how many times they may appear in a link session. Cumulative directives can be specified more than once during a link session, whereas non-cumulative directives may be specified a maximum of one time during any link session. Specifying a non-cumulative directive more than once will cause LN to generate an error. The cumulative directives are:

RAM, ROM, FILE_LIST, MODULE and LIBRARY.

All other directives fall into the non-cumulative category and therefore can be specified only once.

LN identifies a directive by the first n characters of the directive keyword which uniquely distinguish it from other directives. This allows for abbreviated specification of the directive keyword. For example, SB uniquely identifies SB_START and distinguishes it from the STACK directive. However, S is an ambiguous keyword abbreviation and will generate an error. (NOTE: MOD_START and MODULE do not follow this rule!)

### 8.6.1 INPUT DIRECTIVES

The INPUT directives specify which object files LN is to use to produce an executable file. These directives are:

- FILE_LIST
- LIBRARY
- MAIN

#### 8.6.1.1 The FILE_LIST directive

Format: FILE_LIST=<filename>[,<filename>]...

description:

| | | |
|---|---|---|
| | FILE_LIST | is the directive keyword |
| | filename | is the name of an object file generated by AS. The default filename extension is .O32. |

The FILE_LIST directive specifies the object file(s) to use for generating an executable file. This is a required directive.

The current maximum number of files that may be linked is limited to 300 object files including library files. Note a library file is counted as a single file.

Since this directive is cumulative (see section 8.6), it may be specified more than once. This is useful when the names of the files to link do not fit on one line, and a second line is required to complete the list.

If LN determines that a file is unusable (i.e. not found, file is empty, not an object file), an error message is sent to the console.

Examples:

1. LN> FILE_LIST=TEST1,TEST2,TEST3.O32,TEST4
2. LN> F=TEST3.O

In example 1, four object files are specified. Note that no spaces exist between the filenames.

In example 2, "F" uniquely defines the FILE_LIST directive and a non-default extension is specified for the object file.

### 8.6.1.2 The LIBRARY directive

Format: LIBRARY=<filename>[,<filename>]...

description:

| | |
|---|---|
| LIBRARY | is the directive keyword |
| filename | is the name of a library file of object modules generated by AS. The default extension is .O32. |

The LIBRARY directive specifies the object file(s) used by LN for resolving external references not defined by the FILE_LIST object files.

This directive is a cumulative directive (see section 8.6) which allows the user to continue the LIBRARY specification with more than one line.

When an external reference is satisfied by a library, the complete object module is included in the executable file. When the module is included, the linker then tries to resolve all external references in the new module by searching the library modules. Modules which do not resolve external references are not included in the executable file.

If LN determines that a file is unusable (i.e. not found, file is empty, not an object file), an error message is sent to the console.

Examples:

1. LN> LIBRARY=LIB1,LIB2,LIB3.O32,LIB4
   LN> LIBRARY=LIB5,LIB0
2. LN> LIB=LIB0.O

In example 1, the linker is given two library directives.

In example 2, the abbreviated form of the LIBRARY directive is used and a non default extension is specified for LIB0 object file.

### 8.6.1.3 The MAIN directives

Format: MAIN=<modulename>

description:

| | |
|---|---|
| MAIN | is the directive keyword |
| modulename | is the name of a module contained in an object file specified by the FILE_LIST or LIBRARY directive. |

The MAIN directive specifies which module gains control immediately after loading of the executable file on the DSI-32 coprocessor.

If the MAIN directive is not specified, LN determines which module will gain control using the following set of rules:

1. If one and only one of the modules in the FILE_LIST or LIBRARY directives contain the START pseudo-op, pass control to that module.
2. If there is only one module to link, pass control to it.
3. If 1 or 2 is not the case, send an error message to the console and stop linking. Do not generate an executable file.

If the START pseudo-op is in more than one object module, the "one and only one" criteria of rule 1 is not met and LN sends an error message to the console.

Examples:

```
LN> MAIN=FILE1
```

In this example the module FILE1 will gain control immediately prior to any other module. Note that any other module that is linked with FILE1 must not have a START directive, if this is not the case then LN will issue an error regarding multiple main modules.

### 8.6.2 INDIRECT DIRECTIVES

The INDIRECT FILE directives specify which indirect (command) files are to be used in producing an executable file. These directives are:

- INDIRECT
- NODEFAULT

For a full discussion on how to use the linker with indirect files, see section 8.7

#### 8.6.2.1 The INDIRECT directive

Format: INDIRECT=<filename>

description:

| | |
|---|---|
| INDIRECT | is the directive keyword |
| filename | is the name of an ASCII file containing LN directives. The default file extension is .IND. |

The INDIRECT directive specifies the name of a file to be user to direct the linker.

When the INDIRECT directive is used, the linker opens the specified file and begins interpreting the text in the file in a similar manner to keyboard input. The syntax used for directives in indirect files are identical to that for interactive input.

As the linker processes an INDIRECT directive, it echoes the directive to the console prefaced with an at (@) sign. If during the processing of the directive an error is detected, the linker sends an error message to the console immediately after the command. In interactive mode, the linker returns to the LN> prompt after the indirect file has been fully processed. If the linker is in non-interactive mode, after it finishes processing the indirect file, it processes the default file and proceeds to link the modules.

INDIRECT directives cannot be used in indirect files. Link directives (<CR>) are not processed in indirect files.

Examples:

1. LN> IND=TEST
2. C>LN IND=TEST

In example 1, the INDIRECT directive is used to specify the file TEST.IND as the indirect file. This example is in interactive mode.

In example 2, the INDIRECT directive is specified in the MS-DOS command line and causes the linker to process in non-interactive mode.

### 8.6.2.2 The NODEFAULT directive

Format: NODEFAULT

description:
NODEFAULT is the directive keyword

The NODEFAULT directive tells the linker that the file LNx.D32 is not to be searched for on read when the link directive (<CR>) is issued.

The default file (LNx.D32) usually contains directives which remain invariant for each invocation of the linker.

Example:

```
LN> NODEF
```

This example tells the linker that LNx.D32 is not to be searched for or read.

### 8.6.3 MEMORY ALLOCATION DIRECTIVES

The MEMORY ALLOCATION directives control the allocation of certain linker data structures in RAM. These directives are:

- CODE
- LINK
- MOD_START
- RAM
- ROM
- SB_START
- STACK
- MODULE

The order in which these directives are not important and have no impact on how LN allocates memory. For each of the above directives, LN will abort under the following circumstances:

1. an attempt to allocate memory in a non-existent area
2. memory allocations intersect previous allocations
3. an attempt to allocate memory in a ROM area

#### 8.6.3.1 The CODE directive

Format: CODE=<address>

description:

| | |
|---|---|
| CODE | is the directive keyword |
| address | is a valid RAM address |

The CODE directive specifies the beginning location that the linker can load code. This directive allocates a contiguous block of memory for the code of each module starting at <address>.

If the code block does not fit into the allocated memory, LN aborts processing and returns to the MS-DOS system prompt.

Example:

```
LN> CODE=C000
```

This example locates the code block beginning at address C000.

### 8.6.3.2 The LINK directive

Format: LINK=<address>

description:

| | |
|---|---|
| LINK | is the directive keyword |
| address | is a valid RAM address |

The LINK directive specifies the beginning location that the linker can load the link tables for each module. This directive allocates a contiguous block of memory for the links of each module starting at <address>.

If the link block does not fit into the allocated memory, LN aborts processing and returns to the MS-DOS system prompt.

Example:

```
LN> LINK=D000
```

This example locates the link table block beginning at address D000.

### 8.6.3.3 The MOD directive

Format: MOD=<address>

description:

| | | |
|---|---|---|
| MOD | is | the directive keyword |
| address | is | a valid RAM address |

The MOD directive specifies the beginning location that the linker can load the global module tables for each module. This directive allocates a contiguous block of memory for the global module table starting at <address>.

If the global module table block does not fit into the allocated memory, LN aborts processing and returns to the MS-DOS system prompt.

Example:

```
LN> MOD=D000
```

This example locates the global module table block beginning at address D000.

### 8.6.3.4 The RAM directive

Format: RAM={<low_addr>..<high_addr>|<low_addr>+<length>}
[,{<low_addr>..<high_addr>|<low_addr>+<length>}]...

description:

RAM is the directive keyword

low_addr, high_addr
are valid RAM addresses

length is a positive hex integer.

The RAM directive specifies all of the valid RAM address space within which the linker may load code and data. This is a required directive.

The RAM directive can specify valid RAM address space in two ways. First, a range of valid address space can be specified by using the <low_addr>..<high_addr> format. In this format, low_addr is a valid hex address representing the beginning of the address space and high_addr is a valid hex address representing the top of the address space. low_addr must be less than high_addr. In the second format, <low_addr>+<length>, low_addr is a valid hex address representing the beginning of the address space and length is a positive hex integer representing the number of bytes in the space.

More than one contiguous RAM address space can be specified in the RAM directive. However, if the address spaces overlap, LN sends an error message to the console and returns to the MS-DOS system.

NOTE: RAM address spaces must not overlap with memory used by the MON monitor program if it is required to run the monitor and the user program simultaneously. The monitor normally changes the top of the heap to just below itself. Refer to the Monitor Reference Manual for more information.

Examples:

1. LN> RAM=0..FFFFF
2. LN> RAM=9000..DFFF,E000+500

In example 1, the RAM directive specified one megabyte of contiguous memory is available for code and data.

In example 2, the RAM directive specifies two disjoint, contiguous address spaces available for code and data. The first format specifies a range, whereas the second format specifies a number of bytes.

### 8.6.3.5 The ROM directive

Format: ROM={<low_addr>..<high_addr>|<low_addr>+<length>}
[,{<low_addr>..<high_addr>|<low_addr>+<length>}]...

description:

| | |
|---|---|
| ROM | is the directive keyword |
| low_addr, high_addr | are valid ROM addresses |
| length | is a positive hex integer. |

This directive is not intended for use on the DSI-32 coprocessor board. It is solely available when LN is being used to link a program that is targeted to a different 32000 environment.

The ROM directive specifies all of the valid ROM address space within which the linker may not load code and data. This directive is intended for systems that have ROM taking up part of the real or virtual address space.

The ROM directive can specify valid ROM address space in two ways. First, a range of valid address space can be specified by using the <low_addr>..<high_addr> format. In this format, low_addr is a valid hex address representing the beginning of the address space and high_addr is a valid hex address representing the top of the address space. low_addr must be less than high_addr. In the second format, <low_addr>+<length>, low_addr is a valid hex address representing the beginning of the address space and length is a positive hex integer representing the number of bytes in the space.

More than one contiguous ROM address space can be specified in the ROM directive. However, if the address spaces overlap, LN sends an error message to the console and returns to the MS-DOS system.

Examples:

1. LN> ROM=0..200
2. LN> ROM=9000..9020,E000+200

In example 1, the ROM directive specified 201 bytes of contiguous memory is not available for code and data.

In example 2, the ROM directive specifies two disjoint, contiguous address spaces unavailable for code and data. The first format specifies a range, whereas the second format specifies a number of bytes.

### 8.6.3.6 The SB_START directive

Format: SB_START=<address>

description:

| | | |
|---|---|---|
| SB_START | is the directive keyword | |
| address | is a valid RAM address | |

The SB_START directive specifies the address where allocation of the SB (static base) data is to begin.

Example:

```
LN> SB_START=D000
```

In this example, the static base address is set to D000.

### 8.6.3.7 The STACK directive

Format: STACK={<low_addr>..<high_addr>|<low_addr+<length>
|+<length>}

description:

STACK is the directive keyword

low_addr, high_addr
are valid RAM addresses

length is a hex integer

The STACK directive is specifies the address ranges for the global stack.

The STACK directive can specify valid RAM address space in three ways. First, a range of valid address space can be specified by using the <low_addr>..<high_addr> format. In this format, low_addr is a valid hex address representing the beginning of the address space and high_addr is a valid hex address representing the top of the address space. low_addr must be less than high_addr. In the second format, <low_addr>+<length>, low_addr is the bottom of the stack and the top of the stack will be the sum of <low_addr> and <length>. The third format, +<length>, <low_addr> is chosen by LN as the next available memory location and the length is a positive hex integer representing the number of bytes to allocate to the stack.

Example:

```
LN> STACK=+3000
```

In the example, the bottom of the stack is chosen by LN while the top of the stack is the sum of <low_addr> (chosen by LN) and <length>.

### 8.6.3.8 The MODULE directive

Format: MODULE=<modulename>/<memdir>

description:

| | |
|---|---|
| MODULE | is the directive keyword |
| modulename | is a valid module name |
| memdir | is any memory directive other than MODULE |

The MODULE directive allows control of memory allocation on a module by module basis independent of the global memory allocations.

This directive specifies certain allocation requirements that are not necessarily addressed by a global allocation scheme, for example COMMON memory in a FORTRAN program.

Example:

```
LN> MODULE=MAIN/SB_START=E000
LN> MODULE=SUB/SB_START=E000
```

In this example, the static base beginning address for the modules MAIN and SUB are set to E000. This allows both modules to share the same data without having to pass an address.

### 8.6.4 OUTPUT DIRECTIVES

The OUTPUT directives control the output of executable files and map files. These directives are:

- EXECUTABLE/NOEXECUTABLE
- MAP

#### 8.6.4.1 The EXEC and NOEXEC directives

Format: EXEC=<filename>
NOEXEC

description:

| | |
|---|---|
| EXEC, NOEXEC | is the directive keyword |
| <filename> | is the name of an executable file to be generated by the LN. The default filename extension is .E32. |

The EXEC and NOEXEC directives control the output of the executable file generated by LN.

If both the EXEC and NOEXEC directive are omitted from a link session, the name of the executable file defaults to the first file specified in the first FILE_LIST directive with a .E32 extension. If the EXEC directive is specified, the name of the executable file is the name specified in the directive. If no extension is specified, .E32 is the default. If the NOEXEC directive is specified, no executable file is generated. This is useful when only a map file is required.

Examples:

1. LN> EXEC=TEST
2. LN> NOEXEC
   LN> MAP

In example 1, the linker generates an executable file named TEST.E32.

In example 2, the linker does not generate an executable file, though it will attempt to satisfy all unresolved references if directed to. It will generate a MAP file, however.

### 8.6.4.2 The MAP directive

Format: MAP[=<filename>]

description:

| | |
|---|---|
| MAP | is the directive keyword |
| filename | is the name of a map file to be generated by the LN linker. The default filename extension is .M32. |

The MAP directive specifies that a map file is to be generated and optionally allows the user to specify the name of the map file. If the filename is specified and no extension is given in the filename, the extension defaults to .M32. If no filename is given, the name of the map file will default to the name of the first file specified in the first FILE_LIST directive with an extension of .M32.

Example:

```
LN> MAP=TEST
```

In the example, a map file is generated with the name TEST.M32.

### 8.6.5 LINKER CONTROL DIRECTIVES

The CONTROL directives control the execution of LN. These directives are:

- QUIT/EXIT
- link (<CR>)

#### 8.6.5.1 The QUIT and EXIT directives

Format: QUIT
EXIT

description:

QUIT, EXIT are the directive keywords

The QUIT and EXIT function identically. They halt all link activity and return control to the MS-DOS operating system. No output files are generated when these directives are used.

### 8.6.5.2 The link (<CR>) directive

Format: <CR>

description:
<CR> is a carriage return character

The link directive is specified by entering a <CR> immediately after the LN> prompt. This directive is only available in interactive mode.

After specifying all necessary directives for a successful link, the link directive tells the linker to execute the link based on the specifications of the previously entered directives. If all directives have been specified correctly, the linker will attempt to resolve all external references, allocate all memory structures and if directed to do so, will generate executable and map files. If an error occurs during the link operation, an error message is sent to the console and processing stops. Depending on the error and the mode, control returns either to the LN> prompt or the MS-DOS system prompt.

Example:

```
C>LN
DSI-32 Linker V1.10
Copyright (c)1985 Definicon Systems Inc.
LN> FILE=TEST1.O32,TEST2.O32
LN> FILE=TEST3.O32
LN> EXEC=TEST.O32
LN> MOD=80
LN> CODE=4000
LN> RAM=0..fffff
LN>

C>
```

In the example, the object modules TEST1.O32, TEST2.O32 and TEST3.O32 are linked to produce the executable file TEST.E32. Note that the last line of the linker directives is a carriage return character.

## 8.7 USE OF INDIRECT FILES

Indirect files allow the user the ease of instructing the linker by using a file containing a set of directives for linking, rather than having to re-enter the set of directives each time the linker is used.

Indirect files can be specified in two ways. First, using the command line mode, an indirect file can be specified by typing "LN IND=filename" where filename is a file containing linker directives. This instructs the linker to open the file filename and take instructions from that file. The default extension on indirect files is .IND. The second way to specify indirect files is after invoking interactive mode, use the INDIRECT directive to specify an indirect file. Execution of the linker will then proceed with directives from the indirect file.

When invoked in indirect mode, the linker takes directives from the specified indirect file until it reaches the end of the file. When a directive is read from the file, the linker processes the directive, echoes the directive to the console and reports any errors to the console before going on to the next directive. If an error occurs, processing will always continue until the complete file has been read. Depending on the severity of the error, however, the linker will return control to the LN> prompt or to the MS-DOS system prompt.

If, for example the file a.ind has the following directives,

```
FILE=a.o32
EXEC=d.e32
RAM=0..fffff
```

then the following command sequences can be used to take advantage of them:

```
C>LN IND=a.ind
```

or

```
C>LN
DSI-32 Linker V1.10
Copyright (c)1985 Definicon Systems Inc.
LN> IND=a.ind
```

## Object File Structure

### 8.8.1 Introductions

Object files are produced by assembler, AS, and may be manipulated by the librarian, LIB. Object files are formed of five distinct blocks:

- Directory
- Code
- Static Data
- Import Table
- Export Table

Each block starts at a 512 byte boundary within the object file and may occupy zero or more pages. The first block is always the directory block and the rest of the blocks are arbitrarily allocated by either that assembler or librarian.

#### 8.8.1.1 Directory

Each directory block contains information for up to ten modules. If more than ten modules are contained in the ojbect file then further contiguous blocks are allocated. The directory block is formed of three distinct sections: a header section, an array of module information and a trailer (padding to 512 bytes).

| NAME | BITS | CONTENTS |
|---|---|---|
| Header | | |
| NEXTDIR | 16 | Pointer to next directory block, -1 if last block |
| MODCOUNT | 16 | Number of modules in this block |
| Array of module information (maximum of 10 entries per block) | | |
| MODNM | 64 | Module name, left justified, space padded |
| MODTYPE | 3 | Language (0 = pascal, 1 = assembler, 2 =C, 3=fortran) |
| STATUS | 1 | Module status (0=main, 1=not main) |
| UNUSED | 12 | Reserved |
| STARTADDR | 32 | Execution start address (offset from code start) |
| CODELEN | 32 | Code size in bytes |
| SBSIZE | 32 | Static Base data size in bytes |
| SBBLK | 16 | Static base block number |
| SYMLEN | 16 | Symbol table length (currently reserved) |
| SBADDR | 32 | Static Base address (-1 if no SB data) |
| CODEADDR | 32 | Code address (-1 if no code) |
| EXPOBLK | 16 | Start block of export symbols |
| IMPOBLK | 16 | Start block of import symbols |
| CODEBLK | 16 | Start block of code |
| SYMBLK | 16 | Start block of symbol table (reserved) |
| EXPOLEN | 16 | Number of export symbols |
| IMPOLEN | 16 | Number of import symbols |

| NAME | BITS | CONTENTS |
|---|---|---|
| FILLER | 304 | Reserved |
| VERSION | 16 | Version number |
| CHECK | 64 | Character string that contains 'CODEFIL ' |

8.8.1.2 Code

The code block contains the actual program code to be executed. The code blocks are contiguous.

8.8.1.3 Static Base data

The static base data contains the data that is referenced by the SB register of the 32000 series family. The static base blocks are contiquous.

8.8.1.4 Import Table

| NAME | BITS | CONTENTS |
|---|---|---|
| INAME | 72 | The import symbol name |
| ISIZE | 32 | Size of data (used when symbol is common) |
| ITYPE | 1 | Symbol type (0=data, 1=entry point) |
| ICOMM | 1 | Common flag (0=not common, 1=common) |
| ISTAT | 1 | Static flag (0 = global symbol, 1 = local symbol) |
| UNUSED | 13 | Reserved |

8.8.1.5 Export Table

| NAME | BITS | CONTENTS |
|---|---|---|
| ENAME | 72 | The export symbol name |
| ESIZE | 32 | Byte offset within segment |
| ESTAT | 1 | Static flag (0 = global symbol, 1 = local symbol) |
| UNUSED | 2 | Reserved |
| ETYPE | 1 | Type (0 = SB data, 1 = ABS data, 2 = PC, 3 = PC data 4 = PC) entry |
| UNUSED | 10 | Reserved |

**Executable File Structure**

8.8.2 Introduction

Executable files are produced by linker, An executable file is typically made up of serveral object files. They have and extension .E32 and are loaded into the DSI-32 board by the Definicon Loader. An executable file consists of five distinct blocks.

- o General Information
- o Module directory
- o Code
- o Static data
- o Link tables

Each block starts at a 512 byte boundary within the executable file and may occupy zero or more pages. The first block is always the general information block and the rest of the blocks are arbitrarily allocated by the linker.

8.8.2.1 General Information

The general information block contains information that is common to all modules. Although also contained in this section, the Global Module Table, is not physically part of the general information section. It contains a copy of the module data that enables the separate modules in the executable file to communicate and pass data between themselves.

| NAME | BITS | CONTENTS |
|---|---|---|
| EXECID | 16 | 7699 plus Version number |
| DIRBLK | 16 | Pointer to first directory block |
| HEAP_LOW | 32 | Heap low address |
| HEAP_HIGH | 32 | Heap high address |
| STACK_LOW | 32 | Stack low address |
| STACH_HIGH | 32 | Stack high address |
| MAIN | 32 | Main module number |
| MODCOUNT | 32 | Number of modules in the module table |
| MODADDR | 32 | Module table load address |

Global Module Table (MODCOUNT of these items)

| | | |
|---|---|---|
| SBAD | 32 | Static Base data |
| LINKAD | 32 | Link table address |
| CODEAD | 32 | Code address |
| UNUSED | 32 | Reserved |

8.8.2.2 Module Directory

The Module Directories are contiquous arrays of data, one for each General Module Table entry (ie MODCOUNT of them). Eight Module Directories are held in each 512 byte block.

| NAME | BITS | CONTENTS |
|---|---|---|
| MODNM | 64 | Module name (left justified, space padded) |
| MODTYPE | 3 | Reserved |
| UNUSED | 13 | Reserved |
| STRTADDR | 32 | Execution start address (offset from code start) |
| SLEN | 32 | Length of static base data in bytes |
| LLEN | 32 | Length of link table in bytes |
| CLEN | 32 | Length of code in bytes |
| SYMLEN | 32 | Reserved |
| SADDR | 32 | Static base data start address |
| LADDR | 32 | Link table start address |
| CADDR | 32 | Code start address |
| LBLK | 16 | Link table start block number |
| CBLK | 16 | Code start block number |
| SYMBLK | 16 | Reserved |
| SBLK | 16 | Static base start block number |
| UNUSED | 112 | Reserved |

8.8.2.3 Code

The code blocks contain the actual code that belongs to the module.

8.8.2.4 Static Base

The static base blocks contain the data that is referenced by the series 32000 microprocessor's SB register.

8.8.2.5 Link Table

The link table blocks store the linkage information for each module. The link table information allows the external addressing mode of the series 32000 microprocessors to determine either the absolute address of a data item or the module and offset of a procedure in another module.