12. THE LINK EDITOR

The Link Editor

In Chapter 2 there was a discussion of link editor command line options (some of which may also be provided on the **cc**(1) command line). This chapter contains information on the Link Editor Command Language.

The command language enables you to:

- specify the memory configuration of the target machine
- combine the sections of an object file in arrangements other than the default
- bind sections to specific addresses or within specific portions of memory
- define or redefine global symbols

Under most normal circumstances there is no compelling need to have such tight control over object files and their location in memory. When you do need to be precise in controlling the link editor output, you do it via the command language.

Link editor command language directives are passed in a file named on the **ld**(1) command line. Any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives. The following paragraphs define terms and describe conditions with which you need to be familiar before you begin to use the command language.

Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K–1 are then not configured. Unconfigured memory is treated as reserved or unusable by **ld**(1). Nothing can ever be linked into unconfigured memory. Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) illegal or nonexistent with respect to the linking process. Memory configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc. are with respect to the configured sections of the address space.

Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the **ld**(1) command language.

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by **ld**(1). **ld**(1) accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to **ld**(1) can also be absolute files.

Files produced from the compilation system may contain sections called **.text** and **.data**. The **.text** section contains the instruction text (executable instructions), **.data** contains initialized data variables. For example, if a C program contains the global (i.e., not inside a function) declaration:

```
int i = 100;
```

and the assignment:

```
i = 0;
```

then compiled code from the C assignment is stored in **.text**, and the variable **i** is located in **.data**.

Link Editor Command Language

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 12-2, "Syntax Diagram for Input Directives.") Constants are, as in C, recognized as decimal numbers unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers. Symbol names may contain uppercase or lowercase letters, digits, and the underscore, _. Symbols within an expression have the value of the address of the symbol only. **ld**(1) does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

ld(1) uses a **lex**-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names reserved and unavailable as symbol names or section names:

ADDR	**BLOCK**	**GROUP**	**NEXT**	**RANGE**	**SPARE**
ALIGN	**COMMON**	**INFO**	**NOLOAD**	**REGIONS**	**PHY**
ASSIGN	**COPY**	**LENGTH**	**ORIGIN**	**SECTIONS**	**TV**
BIND	**DSECT**	**MEMORY**	**OVERLAY**	**SIZEOF**	

addr	**block**	**length**	**origin**	**sizeof**
align	**group**	**next**	**phy**	**spare**
assign	**l**	**o**	**range**	
bind	**len**	**org**	**s**	

The operators that are supported, in order of precedence from high to low, are shown in Figure 12-1.

symbol
! ~ – (UNARY Minus)
* / %
+ – (BINARY Minus)
>> <<
== != > < <= >=
&
|
&&
||
= += -= *= /=

Figure 12-1. Operator Symbols

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

Assignment Statements

External symbols may be defined and assigned addresses via the assignment statement. The syntax of the assignment statement is:

```
symbol = expression;
```

or:

```
symbol op= expression;
```

where *op* is one of the operators +, –, *, or / . Assignment statements must be terminated by a semicolon.

12

All assignment statements (except the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to **ld**(1).

Assignment statements are normally placed outside the scope of section-definition directives (see "Section Definition Directives" under "Link Editor Command Language"). However, there exists a special symbol, called **dot**, ., that can occur only within a section-definition directive. This symbol refers to the current address of **ld**(1)'s location counter. Thus, assignment expressions involving . are evaluated during the allocation phase of **ld**(1). Assigning a value to the . symbol within a section-definition directive can increment (but not decrement) **ld**(1)'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of the . symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an *n*-byte boundary within an output section, where *n* is a power of 2. For example, the expression:

```
align(n)
```

is equivalent to:

```
(. + n - 1) &~(n - 1)
```

SIZEOF and ADDR are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside section directives.

Link editor expressions may have either an absolute or a relocatable value. When **ld**(1) creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.
- The difference of two relocatable symbols from the same section is absolute.
- All other expressions are combinations of the above.

Specifying a Memory Configuration

MEMORY directives are used to specify:

1. The total size of the virtual space of the target machine.
2. The configured and unconfigured areas of the virtual space.

If no directives are supplied, **ld**(1) assumes that all memory is configured. The size of the default memory is dependent on the target machine.

Using MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters **$**, **.**, or **_**. Names of memory ranges are used by **ld**(1) only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld**(1)'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory.

As an option on the MEMORY directive, attributes may be associated with a named memory area. In future releases this may be used to provide error checking. Currently, error checking of this type is not implemented.

The attributes currently accepted are

1. R : readable memory
2. W : writable memory
3. X : executable, i.e., instructions may reside in this memory
4. I : initializable, i.e., stack areas are typically not initialized

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

The syntax of the MEMORY directive is:

```
MEMORY
{
        name1 (attr) :origin = n1, length = n2
        name2 (attr) :origin = n3, length = n4
        etc.
}
```

12

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype. The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, **ld**(1) can be told that memory is configured in some way other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this:

```
MEMORY
{
        valid : org = 0x10000, len = 0xFE0000
}
```

Section Definition Directives

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections s1 and s2 and the other containing sections s3 and s4, the output object file contains the four sections s1, s2, s3, and s4. The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is:

```
SECTIONS
{
        secname1 :
        {
                file_specifications,
                assignment_statements
        }
        secname2 :
        {
                file_specifications,
                assignment_statements
        }
etc.
}
```

The various types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by:

```
filename ( secname )
```

or:

```
filename ( secnam1 secnam2 . . . )
```

Sections of an input file are separated either by white space or commas as are the file specifications themselves.

```
filename [COMMON]
```

may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

If a file name appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section. For example:

```
SECTIONS
{
        outsec1:
        {
                file1.o (sec1)
                file2.o
                file3.o (sec1, sec2)
        }
}
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be

1. section **sec1** from file **file1.o**
2. all sections from **file2.o**, in the order they appear in the file
3. section **sec1** from file **file3.o**, and then section **sec2** from file **file3.o**

If there are any additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in **file1.o** or **file3.o**, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The code:

```
*(secname)
```

may be used to indicate all previously unallocated input sections of the given name, regardless of what input file they are contained in.

Load a Section at a Specified Address

An output section is bonded to a specific virtual address by an **ld**(1) option as shown in the following SECTIONS directive example:

```
SECTIONS
{
        outsec addr:
        {
                . . .
        }
        etc.
}
```

The *addr* is the bonding address expressed as a C constant. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld**(1) issues an appropriate error message. *addr* may also be the word BIND, followed by a parenthesized expression. The expression may use the pseudo-functions SIZEOF, ADDR or NEXT. NEXT accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; SIZEOF and ADDR accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to **ld**(1) in any particular order, unless SIZEOF or ADDR is used.

ld(1) does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. **ld**(1) directives can be used to force a section to start on an odd byte boundary although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld**(1) issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option:

```
ALIGN(n)
```

is equivalent to specifying a bonding address of:

```
( . + n - 1) &~(n - 1)
```

For example:

```
SECTIONS
{
        outsec  ALIGN(0x20000) :
        {
                . . .
        }
        etc.
}
```

The output section **outsec** is not bound to any given address but is placed at some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

Grouping Sections Together

The default allocation algorithm for **ld**(1) does the following:

1. Links all input **.init** sections together, followed by **.text** sections, into one output section. This output section is called **.text** and is bound to an address of 0x2000 plus the size of all headers in the output file.
2. Links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine dependent constant plus a number dependent on the size of headers and text.
3. Links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated to immediately follow the output section **.data**. Note that the output section **.bss** is not given any particular address alignment.

Specifying any SECTIONS directives results in this default allocation not being performed. Rather than relying on the **ld**(1) default algorithm, if you are

manipulating COFF files, the one certain way of determining address and order information is to take it from the file and section headers. The default allocation of **ld**(1) is equivalent to supplying the following directive:

```
SECTIONS
{
        .text sizeof_headers : { *(.init) *(.text) }
        GROUP BIND( NEXT(align_value) +
                        ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
{
        .data     : { }
                .bss      : { }
        }
}
```

where *align_value* is a machine dependent constant. The GROUP command ensures that the two output sections, **.data** and **.bss**, are allocated (e.g., grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following SECTIONS directive is used:

```
SECTIONS
{
        GROUP                       :
        {
                .text           : { }
                .data           : { }
                .bss            : { }
        }
}
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive. To bind to 0xC0000, use:

```
GROUP 0xC0000 : {
```

To align to 0x10000, use:

```
GROUP ALIGN(0x10000) : {
```

With this addition, first the output section **.text** is bound at 0xC0000 (or is aligned to 0x10000); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the GROUP directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
        .text    : { }
        .data ALIGN(0x20000)   : { }
        .bss     : { }
}
```

The **.text** section starts at virtual address 0x0 (if it is in configured memory) and the **.data** section at a virtual address aligned to 0x20000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section. The order in which output sections are defined to **ld**(1) cannot be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections

The special symbol dot, ., appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, . causes **ld**(1)'s location counter to be incremented or reset and a hole left in the output section. Holes built into output sections this way take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the **–f** option in "Using the Link Editor" and the discussion of filling holes in "Initialized Section Holes" or "**.bss** Sections" in this chapter.

Consider the following section definition:

```
outsec:
{
        . += 0x1000;
        f1.o (.text)
        . += 0x100;
        f2.o (.text)
        . = align (4);
        f3.o (.text)
}
```

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section **f1.o (.text)** is linked after this hole.
2. The **.text** section of input file **f2.o** begins at 0x100 bytes following the end of **f1.o (.text)**.
3. The **.text** section of **f3.o** is linked to start at the next full word boundary following the **.text** section of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld**(1) treats the output section as if it began at address zero. As a result, if, in the above example, **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address—even though **f3.o (.text)** is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

Note that the assembler, **as**, always pads the sections it generates to a full word length making explicit alignment specifications unnecessary. This also holds true for the compiler.

Expressions that decrement . are illegal. For example, subtracting a value from the location counter is not allowed since overwrites are not allowed. The most common operators in expressions that assign a value to . are += and **align**.

Creating and Defining Symbols at Link-Edit Time

The assignment instruction of **ld**(1) can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. Use of **.** to adjust **ld**(1)'s location counter during allocation.
2. Use of **.** to assign an allocation-dependent value to a symbol.
3. Assigning an allocation-independent value to a symbol.

Case 1) has already been discussed in the previous section.

Case 2) provides a means to assign addresses (known only after allocation) to symbols. For example:

```
SECTIONS
{
        outsc1: {...}
        outsc2:
        {
                file1.o (s1)
                s2_start = . ;
                file2.o (s2)
                s2_end = . - 1;
        }
}
```

The symbol **s2_start** is defined to be the address of **file2.o(s2)**, and **s2_end** is the address of the last byte of **file2.o(s2)**.

Consider the following example:

```
SECTIONS
{
        outsc1:
        {
                file1.o (.data)
                mark = .;
                . += 4;
                file2.o (.data)
        }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file1.o's .data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving . must appear within SECTIONS definitions since they are evaluated during allocation. Assignment instructions that do not involve . can appear within SECTIONS definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. The **ld**(1) issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a MEMORY directive). (The > notation is borrowed from the operating system concept of redirected output.) For example:

```
MEMORY
{
        mem1:           o=0x000000    l=0x10000
        mem2 (RW):      o=0x020000    l=0x40000
        mem3 (RW):      o=0x070000    l=0x40000
        mem1:           o=0x120000    l=0x04000
}

SECTIONS
{
        outsec1: { f1.o(.data) } > mem1
        outsec2: { f2.o(.data) } > mem3
}
```

12

This directs **ld**(1) to place **outsec1** anywhere within the memory area named **mem1** (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Creating Holes within Output Sections"), **ld**(1) normally puts out bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the **.o** file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized **.bss** section, if part of such a section is initialized, then the entire section is initialized. In other words, if a **.bss** section is to be combined with a **.text** or **.data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will hold:

a. Explicit initialization options must be used to initialize all **.bss** sections in the output section.

b. **ld**(1) will use the default fill value to initialize all **.bss** sections in the output section.

Consider the following **ld**(1) ifile:

```
SECTIONS
{
        sec1:
        {
                f1.o
                . =+ 0x200;
                f2.o (.text)
        } = 0xDFFF
        sec2:
        {
                f1.o (.bss)
                f2.o (.bss) = 0x1234
        }
        sec3:
        {
                f3.o (.bss)
                . . .
        } = 0xFFFF
         sec4: { f4.o (.bss) }
}
```

In the example above, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2**, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All **.bss** sections within sec3 as well as all holes are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object file for this section.

Notes and Special Considerations

Changing the Entry Point

The **a.out** optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

a. The value of the symbol specified with the **-e** option, if present, is used.

b. The value of the symbol **_start**, if present, is used.

c. The value of the symbol **main**, if present, is used.

d. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the **–e** option or by using an assignment instruction in an ifile of the form

```
_start  =  expression;
```

If **ld**(1) is called through **cc**(1), a startup routine is automatically linked in. Then, when the program is executed, the routine **exit**(1) is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld**(1) directly or when changing the entry point. The user must supply the startup routine or make sure that the program always calls exit rather than falling through the end. Otherwise, the program will dump core.

Use of Archive Libraries

Each member of an archive library (e.g., **libc.a**) is a complete object file. Archive libraries are created with the **ar**(1) command from object files generated by **cc** or **as**. An archive library is always processed using selective inclusion: only those members that resolve existing undefined-symbol references are taken from the library for link editing. Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever:

a. There exists a reference to a symbol defined in that member.

b. The reference is found by **ld**(1) before the scanning of the library.

When a library member is included by searching the library inside a SECTIONS directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library outside a SECTIONS directive, all input sections from the library member are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that:

1. Specific members of a library cannot be referenced explicitly in an ifile.

2. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The **–l** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the **–l** option by simply giving the (full or relative) file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld**(1) will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

1. The input files **file1.o** and **file2.o** each contain a reference to the external function FCN.
2. Input **file1.o** contains a reference to symbol ABC.
3. Input **file2.o** contains a reference to symbol XYZ.
4. Library **liba.a**, member 0, contains a definition of XYZ.
5. Library **libc.a**, member 0, contains a definition of ABC.
6. Both libraries have a member 1 that defines FCN.

If the **ld**(1) command were entered as:

```
ld file1.o -la file2.o -lc
```

then the FCN references are satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ remains undefined (because the library **liba.a** is searched before **file2.o** is specified). If the **ld**(1) command were entered as:

```
ld file1.o file2.o -la -lc
```

then the FCN references is satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0. If the **ld**(1) command were entered as:

```
ld file1.o file2.o -lc -la
```

then the FCN references is satisfied by **libc.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0.

The **-u** option is used to force the linking of library members when the link edit run does not contain an external reference to the members. For example, the command:

```
ld -u rout1 -la
```

creates an undefined symbol called **rout1** in **ld**(1)'s global symbol table. If any member of library **liba.a** defines this symbol, it (and perhaps other members as

well) is extracted. Without the **–u** option, there would have been no unresolved references or undefined symbols to cause **ld**(1) to search the archive library.

Dealing With Holes in Physical Memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
        mem1:       o = 0x00000       l = 0x02000
        mem2:       o = 0x40000       l = 0x05000
        mem3:       o = 0x20000       l = 0x10000
}
```

Let the files **f1.o**, **f2.o**, . . . **f***n***.o** each contain three sections **.text**, **.data**, and **.bss**, and suppose the combined **.text** section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the **.text** output section so **ld**(1) may do allocation. For example:

```
SECTIONS
{
        txt1:
        {
                f1.o (.text)
                f2.o (.text)
                f3.o (.text)
        }
        txt2:
        {
                f4.o (.text)
                f5.o (.text)
                f6.o (.text)
        }
        etc.
}
```

Allocation Algorithm

An output section is formed either as a result of a SECTIONS directive, by combining input sections of the same name, or by combining **.text** and **.init** into **.text**. An output section can comprise zero or more input sections. After the composition of an output section is determined, it must then be allocated into configured virtual memory. **ld**(1) uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit bonding addresses were specified are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.
3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order they appear to **ld**(1). Otherwise, output sections are allocated in the order they were defined or made known to **ld**(1) into the first available space they fit.

Incremental Link Editing

As previously mentioned, the output of **ld**(1) can be used as an input file to subsequent **ld**(1) runs providing that the relocation information is retained (**–r** option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application.

For example, Step 1:

ld –r –o outfile1 ifile1 infile1.o

```
/* ifile1  */
SECTIONS
{
        ss1:
        {
                f1.o
                f2.o
                . . .
                fn.o
        }
}
```

Step 2:

ld –r –o outfile2 ifile2 infile2.o

```
/* ifile2  */
SECTIONS
{
        ss2:
        {
                g1.o
                g2.o
                . . .
                gn.o
        }
}
```

Step 3:

ld –a –o final.out outfile1 outfile2

By judiciously forming subsystems, applications may achieve a form of incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules:

1. Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.
2. All allocation and memory directives, as well as any assignment statements, are included only in the final **ld**(1) call.

DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections

Sections may be given a type in a section definition as shown in the following example:

```
SECTIONS
{
        name1 0x200000 (DSECT)      : { file1.o }
        name2 0x400000 (COPY)       : { file2.o }
        name3 0x800000 (NOLOAD)     : { file3.o }
        name4          (INFO)       : { file4.o }
        name5 0x900000 (OVERLAY)    : { file5.o }

}
```

The DSECT option creates what is called a dummy section. A dummy section has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld**(1).
2. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not as a DSECT).
4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from **file1.o** are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the COPY option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An INFO section is the same as a COPY section but its purpose is to carry information about the object file whereas the COPY section may contain valid text and data. INFO sections are usually used to contain file version identification information.

A section with the type of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, etc.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct **ld**(1) to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file:

```
SECTIONS
{
        .text BLOCK(0x200) : { }
        .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this SECTIONS directive, **ld**(1) assures that each section, **.text** and **.data**, is physically written at a file offset, which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400, and so forth, in the file).

Nonrelocatable Input Files

If a file produced by **ld**(1) is intended to be used in a subsequent **ld**(1) run, the first **ld**(1) run should have the **–r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld**(1) does not have relocation or symbol table information (perhaps from the action of a **strip**(1) command, or from being link edited without a **–r** option or with a **–s** option), the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met:

1. Each input file must have no unresolved external references.
2. Each input file must be bound to the exact same virtual address as it was bound to in the **ld**(1) run that created it.

NOTE

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld**(1).

Syntax Diagram for Input Directives

Figure 12-2 summarizes the system requirements for input directives. Note that two punctuation symbols, square brackets and curly braces, do double duty in this diagram.

Where the symbols [] and {} are used, they are part of the syntax and must be present when the directive is specified.

Where you see the symbols and (larger and in bold), it means the material enclosed is optional.

Where you see the symbols { and } (larger and in bold), it means multiple occurrences of the material enclosed are permitted.

Directives	Expanded Directives
<ifile>	{<cmd>}
<cmd>	<memory>
	<sections>
	<assignment>
	<filename>
	<flags>
<memory>	MEMORY { <memory_spec>
<memory_spec>	<name> [<attributes>] :
<attributes>	({ R | W | X | I })
<origin_spec>	<origin> = <long>
<lenth_spec>	<length> = <long>
<origin>	ORIGIN | o | org | origin
<length>	LENGTH | l | len | length

Figure 12-2. Syntax Diagram for Input Directives (Sheet 1 of 4)

Directives	Expanded Directives
<sections>	SECTIONS { {<sec_or_group>} }
<sec_or_group>	<section> | <group> | <library>
<group>	GROUP <group_options> : {
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> :
<group_options>	[<addr>] | [<align_option>] [<block_option>]
<sec_options>	[<addr>] | [<align_option>]
<addr>	<long> | <bind>(<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN | align
<block_option>	<block> (<long>)
<block>	BLOCK | block
<type_option>	(DSECT) | (NOLOAD) | (COPY)
<fill>	= <long>
<mem_spec>	> <name> > <attributes>
<statement>	<filename> <filename> (<name_list>) | [COMMON] * (<name_list>) | [COMMON] <assignment> <library> *null*

Figure 12-2. Syntax Diagram for Input Directives (Sheet 2 of 4)

Directives	Expanded Directives
<name_list>	<section_name> [,] { <section_name> }
<library>	–l<name>
<bind>	BIND | bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> | .
<assign_op>	= | += | –= | *= |/ =
<end>	; | ,
<expr>	<expr> <binary_op> <expr>
	<term>
<binary_op>	* | / | %
	+ | –
	>> | <<
	== | != | > | < | <= | >=
	&
	|
	&&
	||
<term>	<long>
	<name>
	<align> (<term>)
	(<expr>)
	<unary_op> <term>
	<phy> (<lside>)
	<sizeof>(<sectionname>)
	<next>(<long>)
	<addr>(<sectionname>)
<unary_op>	! | –
<phy>	PHY | phy
<sizeof>	SIZEOF | sizeof

12

Figure 12-2. Syntax Diagram for Input Directives (Sheet 3 of 4)

Directives	Expanded Directives
<next>	NEXT | next
<addr>	ADDR | addr
<flags>	–e<wht_space><name>
	–f<wht_space><long>
	–h<wht_space><long>
	–l<name>
	–m
	–o<wht_space><filename>
	–r
	–s
	–t
	–u<wht_space><name>
	–z
	–H
	–L<path_name>
	–M
	–N
	–S
	–V
	–VS<wht_space><long>
	–a
	–x
<name>	Any valid symbol name
<long>	Any valid long integer constant
<wht_space>	Blanks, tabs, and newlines
<filename>	Any valid operating system
<sectionname>	Any valid section name,
<path_name>	Any valid operating system

Figure 12-2. Syntax Diagram for Input Directives (Sheet 4 of 4)