Link-Editor 2

Overview

The link-editing process builds an output file from one or more input files. The building of the output file is directed by the options supplied to the link-editor together with the input sections provided by the input files.

All files are represented in the *executable and linking format* (ELF). For a complete description of the ELF format see Chapter 6, "Object Files". For this introduction, however, it is first necessary to introduce two ELF structures, *sections* and *segments*.

Sections are the smallest indivisible units that can be processed within an ELF file. Segments are a collection of sections that represent the smallest individual units that can be mapped to a memory image by `exec(2)` or by the runtime linker.

Although there are many types of ELF sections, they all fall into two categories with respect to the link-editing phase:

- Sections that contain *program data*, whose interpretation is meaningful only to the application itself (such as the program instructions *.text* and the associated data *.data* and *.bss*).
- Sections that contain *link-editing information* (such as the symbol table information found from *.symtab* and *.strtab*, and relocation information such as *.rela.text*).

2

Basically, the link-editor concatenates the *program data* sections into the output file. The *link-editing information* sections are interpreted by the link-editor to modify other sections or to generate new output information sections used in later processing of the output file.

The following simple breakdown of link-editor functionality introduces the topics covered in this chapter:

- It verifies and checks for consistency all the options passed to it.
- It concatenates sections of the same characteristics (for example, type, attributes and name) from the input relocatable objects to form new sections within the output file. These concatenated sections can in turn be associated to output segments.
- It reads symbol table information from both relocatable objects and shared objects to verify and unite references with definitions, and usually generates a new symbol table, or tables, within the output file.
- It reads relocation information from the input relocatable objects and applies this information to the output file by updating other input sections. In addition, output relocation sections might be generated for use by the runtime linker.
- It generates *program headers* that describe any segments created.
- It generates a dynamic linking information section if necessary, which provides information such as shared object dependencies to the runtime linker.

The process of concatenating like *sections* and associating *sections* to *segments* is carried out using default information within the link-editor. The default *section* and *segment* handling provided by the link-editor is usually sufficient for most link-edits. However, these defaults can be manipulated using the `-M` option with an associated `mapfile` (see Chapter 7, "Mapfile Option" for more details).

Invoking the Link-Editor

You can either run the link-editor directly from the command-line or have a compiler driver invoke it for you. In the following two sections both methods are expanded upon. However, the latter is the preferred choice, as the compilation environment is often the consequence of a complex and occasionally changing series of operations known only to compiler drivers.

Direct Invocation

When you invoke the link-editor directly, you have to supply every object file and library required to build the intended output. The link-editor makes no assumptions about the object modules or libraries you *meant* to use in building the output. For example, when you issue the command:

```
$ ld test.o
```

the link-editor builds a dynamic executable named `a.out` using *only* the input file `test.o`. For the `a.out` to be a useful executable, it should include start-up and exit processing code. This code can be language or operating system specific, and is usually provided through files supplied by the compiler drivers.

Additionally, you can also supply your own initialization and termination code. This code must be encapsulated and labeled correctly for it to be correctly recognized and made available to the runtime linker. This encapsulation and labeling is also provided through files supplied by the compiler drivers.

In practice, there is little reason to invoke the link-editor directly.

Using a Compiler Driver

The conventional way to use the link-editor is through a language-specific compiler driver. You supply the compiler driver, `cc(1)`, `f77(1)`, etc., with the input files that make up your application, and the compiler driver adds additional files and default libraries to complete the link-edit. These additional files can be seen by expanding the compilation invocation, for example:

```
$ cc -# -o prog main.o
/usr/ccs/bin/ld -dy /opt/COMPILER/crti.o /opt/COMPILER/crt1.o \
/usr/ccs/lib/values-Xt.o -o prog main.o \
-YP,/opt/COMPILER/lib:/usr/ccs/lib:/usr/lib -Qy -lc \
/opt/COMPILER/crtn.o
```

Note – This is an example; the actual files included by your compiler driver and the mechanism used to display the link-editor invocation might differ.

Specifying the Link-Editor Options

Most options to the link-editor can be passed through the compiler driver command-line. For the most part the compiler and the link-editor options do not conflict. Where a conflict arises, the compiler drivers usually provide a command-line syntax that allows you to pass specific options to the link-editor. However, you can instead provide options to the link-editor by setting the `LD_OPTIONS` environment variable. For example:

```
$ LD_OPTIONS="-R /home/me/libs -L /home/me/libs" cc -o prog \
main.c -lfoo
```

Here the `-R` and `-L` options will be interpreted by the link-editor and *prepended* to any command-line options received from the compiler driver.

The link-editor parses the entire option list looking for any invalid options or any options with invalid associated arguments. When either of these cases is found, a suitable error message is generated, and when the error is deemed fatal the link-edit terminates. For example:

```
$ ld -X -z sillydefs main.o
ld: illegal option -- X
ld: fatal: option -z has illegal argument 'sillydefs'
```

Here the illegal option `-X` is identified, and the illegal argument to the `-z` option is caught by the link-editor's checking. If an option requiring an associated argument is mistakenly specified twice the link-editor will provide a suitable warning but will continue with the link-edit. For example:

```
$ ld -e foo ..... -e bar main.o
ld: warning: option -e appears more than once, first setting taken
```

The link-editor also checks the option list for any fatal inconsistences. For example:

```
$ ld -dy -r main.o
ld: fatal: option -dy and -r are incompatible
```

After processing all options, and if no fatal error conditions have been detected, the link-editor proceeds to process the input files.

See Appendix A, "Link-Editor Quick Reference" for the most commonly used link-editor options, and the `ld(1)` manual page for a complete description of all link-editor options.

Input File Processing

The link-editor reads input files in the order in which they appear on the command-line. Each file is opened and inspected to determine its ELF file type and so determine how it must be processed. The file types applicable as input for the link-edit are determined by the binding mode of the link-edit, either *static* or *dynamic*.

Under *static* linking the link-editor will accept only relocatable objects or archive libraries as input files. Under *dynamic* linking the link-editor will also accept shared objects.

Relocatable objects represent the most basic input file type to the link-editing process. The *program data* sections within these files are concatenated into the output file image being generated. The *link-edit information* sections are organized for later use, but will not become part of the output file image, as new sections will be generated to take their places. Symbols are gathered into a special internal symbol table that allows for their verification and resolution, and eventually for the creation of one or more symbol tables in the output image.

Although any input file can be specified directly on the link-edit command-line, archive libraries and shared objects are commonly specified using the `-l` option (see "Linking with Additional Libraries" on page 14 for coverage of this mechanism and how it relates to the two different linking modes). However, even though shared objects are often referred to as shared *libraries*, and both of

these objects can be specified using the same option, the interpretation of shared objects and archive libraries is quite different. The next two sections expand upon these differences.

Archive Processing

Archives are built using `ar(1)`, and usually consist of a collection of relocatable objects with an archive symbol table. This symbol table provides an association of symbol definitions with the objects that supply these definitions. When the link-editor reads an archive, it uses information within the internal symbol table it is creating to select *only* the objects from the archive it requires to complete the binding process. To be more precise, the link-editor will extract a relocatable object from an archive if:

- The archive contains a symbol definition that satisfies a symbol reference (sometimes referred to as an *undefined* symbol) presently held in the link-editor's internal symbol table, or
- The archive contains a *data* symbol definition that satisfies a *tentative* symbol definition presently held in the link-editor's internal symbol table. An example of this is a FORTRAN COMMON block definition which will cause the extraction of a relocatable object that defines the same DATA symbol.

Note – A *weak* symbol reference will not cause the extraction of an object from an archive. Weak symbols are expanded upon in section "Simple Resolutions" on page 22.

The link-editor will make multiple passes through an archive extracting relocatable objects as needed to satisfy the symbol information being accumulated in the link-editor internal symbol table. Once the link-editor has made a complete pass through the archive without extracting any relocatable objects, it will move on to process the next input file. This mechanism of extracting from the archive only the relocatable objects needed *at the time* the archive was encountered means that the position of the archive within the input file list can be significant (see "Position of an Archive on the Command-Line" on page 16 for more details).

Note – Although the link-editor will make multiple passes through an archive to resolve symbols, this mechanism can be quite costly for large archives containing random organizations of relocatable objects. In these cases it is

recommended that tools like `lorder(1)` and `tsort(1)` be used to order the relocatable objects within the archive and so reduce the number of passes the link-editor must carry out.

Shared Object Processing

Shared objects are indivisible, whole units that have been generated by a previous link-edit of one or more input files. When the link-editor processes a shared object the entire contents of the shared object become a *logical* part of the resulting output file image. The shared object is not copied physically during the link-edit as its actual inclusion is deferred until process execution. This logical inclusion means that *all* symbol entries defined in the shared object are made available to the link-editing process.

The shared object's *program data* sections and most of the *link-editing information* sections are *unused* by the link-editor, as these will be interpreted by the runtime linker when the shared object is bound to generate a runable process. However, the occurrence of a shared object will be remembered, and information will be stored in the output file image to indicate that this object is a dependency and must be made available at runtime.

If a shared object has dependencies on other shared objects, these too will be processed. This processing will occur *after* all command-line input files have been processed. These shared objects will be used to complete the symbol resolution process, however their names *will not* be recorded as dependencies in the output file image being generated.

Although the position of a shared object on the link-edit command-line has less significance than it does for archive processing, it can have a global effect. Multiple symbols of the same name are allowed to occur between relocatable objects and shared objects, and between multiple shared objects (see "Symbol Resolution" on page 21 for more details).

The *order* of shared objects processed by the link-editor is maintained in the dependency information stored in the output file image. As the runtime linker reads this information it will load the specified shared objects in the same order. Therefore, the link-editor and the runtime linker will select the first occurrence of a symbol of a multiply defined series of symbols.

Note – Multiple symbol definitions, and thus the information to describe the interposing of one definition of a symbol for another, are reported in the load map output generated using the `-m` option.

Linking with Additional Libraries

Although the compiler drivers will often ensure that appropriate libraries are specified to the link-editor, it is frequently necessary for you to supply your own. Shared objects and archives can be specified by explicitly naming the input files required to the link-editor, but a more common and more flexible method involves using the link-editor's `-l` option.

Library Naming Conventions

By convention, shared objects are usually designated by the prefix `lib` and the suffix `.so`, and archives are designated by the prefix `lib` and the suffix `.a`. For example, `libc.so` is the shared object version of the standard C library made available to the compilation environment, and `libc.a` is its archive version.

These conventions are recognized by the `-l` option of the link-editor. This option is commonly used to supply additional libraries to a link-edit. The following example:

```
$ cc -o prog file1.c file2.c -lfoo
```

directs the link-editor to search for `libfoo.so`, and if it does not find it, to search for `libfoo.a`, before moving on to the next directory to be searched.

Note – There is a naming convention regarding the *compilation* environment and the *runtime* environment use of shared objects. The compilation environment uses the simple `.so` suffix, whereas the runtime environment commonly uses the suffix with an additional version number. See "Naming Conventions" on page 84, and "Coordination of Versioned Filenames" on page 136 for more details.

When link-editing in dynamic mode, you can choose to link with a mix of shared objects and archives. When link-editing in static mode, only archive libraries are acceptable for input.

When in dynamic mode and using the `-l` option to enable a library search, the link-editor will first search in a given directory for a shared object that matches the specified name. If no match is found the link-editor will then look for an archive library in the same directory. When in static mode and using the `-l` option, only archive libraries will be sought.

Linking with a Mix of Shared Objects and Archives

Although the library search mechanism, in dynamic mode, searches a given directory for a shared object, and then an archive library, finer control of the type of search required can be achieved using the `-B` option.

By specifying the `-Bdynamic` and `-Bstatic` options on the command-line, as many times as required, the library search can be toggled between shared objects or archives respectively. For example, to link an application with the archive `libfoo.a` and the shared object `libbar.so`, issue the following command:

```
$ cc -o prog main.o file1.c -Bstatic -lfoo -Bdynamic -lbar
```

The `-Bstatic` and `-Bdynamic` keywords are not exactly symmetrical. When you specify `-Bstatic`, the link-editor does *not* accept shared objects as input until the next occurrence of `-Bdynamic`. However, when you specify `-Bdynamic`, the link-editor will first look for shared objects and then archives in any given directory.

In the previous example it is more precise to say that the link-editor first searches for `libfoo.a`, and then for `libbar.so`, and if that fails, for `libbar.a`. Finally, it will search for `libc.so`, and if that fails, `libc.a`.

Another example of using these options is in the creation of an ABI-conforming application. For example:

```
$ cc -o prog main.c file1.c -lsys -Bstatic
```

Here all the basic system routines defined in `libsys.so` will be bound to this shared object. Because the compiler driver appends a `-lc` to the options supplied to the link-editor, and because the `-Bstatic` has instructed the link-editor to search for archive libraries only, any remaining undefined symbols will be resolved by extracting the appropriate relocatable objects from `libc.a`.

Position of an Archive on the Command-Line

The position of an archive on the command-line can affect the output file being produced. The link-editor searches an archive only to resolve undefined or tentative external references it has previously seen. Once this search is completed and the required relocatable objects have been extracted, the archive will not be available to resolve any new symbols obtained from the input files that follow the archive on the command-line. For example, the command

```
$ cc -o prog file1.c -Bstatic -lfoo file2.c file3.c -Bdynamic
```

directs the link-editor to search `libfoo.a` only to resolved symbol references that have been obtained from `file1.c`; `libfoo.a` will not be available to resolve symbol references from `file2.c` or `file3.c`.

Note – As a rule, it is best to specify any archives at the end of the command-line unless multiple-definition conflicts require you to do otherwise.

Directories Searched by the Link-Editor

All previous examples assumed that the link-editor knows where to search for the libraries listed on the command-line. By default the link-editor knows of only two standard places to look for libraries, `/usr/ccs/lib` and `/usr/lib`. All other directories to be searched must be added to the link-editor's search path explicitly.

There are two ways to change the link-editor search path: using a command-line option, or using an environment variable.

Using a Command-Line Option

The `-L` option can be used to *add* a new pathname to the library search path. This option affects the search path *at the point* it is encountered on the command-line. For example, the command

```
$ cc -o prog main.o -Lpath1 file1.c -lfoo file2.c -Lpath2 -lbar
```

searches `path1` (then `/usr/ccs/lib` and `/usr/lib`) to find `libfoo`, but searches `path1` and then `path2` (and then `/usr/ccs/lib` and `/usr/lib`) to find `libbar`.

Pathnames defined using the `-L` option are used only by the link-editor. They are not recorded in the output file image created for use by the runtime linker.

Note – You must specify `-L` if you want the link-editor to search for libraries in your current directory. You can use a period (.) to represent the current directory.

The `-Y` option can be used to *change* the default directories searched by the link-editor. The argument supplied with this option takes the form of a colon separated list of directories. For example, the command

```
$ cc -o prog main.c -YP,/opt/COMPILER/lib:/home/me/lib -lfoo
```

searches for `libfoo` only in the directories `/opt/COMPILER/lib` and `/home/me/lib`. The directories specified using the `-Y` option can be supplemented by using the `-L` option.

Using an Environment Variable

You can also use the environment variable `LD_LIBRARY_PATH`, which takes a colon-separated list of directories, to add to the link-editor's library search path. In its most general form, `LD_LIBRARY_PATH` takes two directory lists separated by a semicolon. The first list is searched before the list(s) supplied on the command-line, and the second list is searched after.

Here is the combined effect of setting `LD_LIBRARY_PATH` and calling the link-editor with several `-L` occurrences:

```
$ LD_LIBRARY_PATH=dir1:dir2;dir3
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

The effective search path will be `dir1:dir2:path1:path2... pathn:dir3:/usr/ccs/lib:/usr/lib`.

If no semicolon is specified as part of the `LD_LIBRARY_PATH` definition the specified directory list is interpreted *after* any `-L` options. For example:

```
$ LD_LIBRARY_PATH=dir1:dir2
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

Here the effective search path will be `path1:path2... pathn:dir1:dir2:/usr/ccs/lib:/usr/lib`.

Note – This environment variable can also be used to augment the search path of the runtime linker (see "Directories Searched by the Runtime Linker" on page 55 for more details). To prevent this environment variable from influencing the link-editor the `-i` option can be used.

Directories Searched by the Runtime Linker

The runtime linker knows of only one standard place to look for libraries, `/usr/lib`. All other directories to be searched must be added to the runtime linker's search path explicitly.

When a dynamic executable or shared object is linked with additional shared objects, these shared objects are recorded as dependencies that must be located again during process execution by the runtime linker. During the link-edit, one or more pathnames can be recorded in the output file. These pathnames will be used by the runtime linker to search for any shared object dependencies. These recorded pathnames are referred to as a *runpath*.

Note – No matter how you modify the runtime linker's library search path, its last element is always `/usr/lib`.

The `-R` option, which takes a colon-separated list of directories, can be used to record a runpath in a dynamic executable or shared library. For example:

```
$ cc -o prog main.c -R/home/me/lib:/home/you/lib -Lpath1 \
-Lpath2 file1.c file2.c -lfoo -lbar
```

will record the runpath `/home/me/lib:/home/you/lib` in the dynamic executable `prog`. The runtime linker will use these paths, and then the default location `/usr/lib`, to locate any shared object dependencies. In this case, this runpath will be used to locate `libfoo.so.1` and `libbar.so.1`.

The link-editor accepts multiple `-R` options and will concatenate each of these specifications, separated by a colon. Thus, the above example can also be expressed as:

```
$ cc -o prog main.c -R/home/me/lib -Lpath1 \
-R/home/you/lib -Lpath2 file1.c file2.c -lfoo -lbar
```

Note – A historic alternative to specifying the `-R` option is to set the environment variable `LD_RUN_PATH`, and make this available to the link-editor. The scope and function of `LD_RUN_PATH` and `-R` are identical, but when both are specified, `-R` supersedes `LD_RUN_PATH`.

Initialization and Termination Sections

The *.init* and *.fini* section types provide for runtime initialization and termination processing. These section types are concatenated from the input relocatable objects like any other sections. However, the compiler drivers can also supply *.init* and *.fini* sections as part of the additional files they add to the beginning and end of the your input-file list.

These files have the effect of encapsulating the *.init* and *.fini* code into individual functions that are identified by the reserved symbol names `_init` and `_fini` respectively.

When building a dynamic executable or shared object, the link-editor records these symbol addresses in the output file's image so they can be called by the runtime linker during initialization and termination processing. See "Initialization and Termination Routines" on page 64 for more details on the runtime processing of these sections.

The creation of *.init* and *.fini* sections can be carried out directly using an assembler, or some compilers can offer special primitives to simplify their declaration. For example, the following code segments result in a *call* to the function `foo` being placed in an *.init* section, and a *call* to the function `bar` being placed in a *.fini* section:

```
#pragma init (foo)
#pragma fini (bar)

foo()
{
    /* Perform some initialization processing. */
    ......
}

bar()
{
    /* Perform some termination processing. */
    .......
}
```

Care should be taken when designing initialization and termination code that can be included in both a shared object and archive library. If this code is spread throughout several relocatable objects within an archive library, then the link-edit of an application using this archive can extract only a portion of the modules, and therefore only a portion of the initialization and termination code. At runtime, only this portion of code will be executed.

The same application built against the shared object will have *all* the accumulated initialization and termination code executed at runtime when the shared object is mapped in as one of the application's dependencies.

Symbol Processing

During input file processing, all *local* symbols from the input relocatable objects are passed through to the output file image. All *global* symbols are accumulated internally within the link-editor. This internal symbol table is searched for each new *global* symbol entry processed to determine if a symbol with the same name has already been encountered from a previous input file. If so, a symbol resolution process is called to determine which of the two entries is to be kept.

On completion of input file processing, and providing no fatal error conditions have been encountered during symbol resolution, the link-editor determines if any unbound symbol references (undefined symbols) remain that will cause the link-edit to fail.

Finally, the link-editor's internal symbol table is added to the symbol table(s) of the image being created.

The following sections expand upon symbol resolution and undefined symbol processing.

Symbol Resolution

Symbol resolution runs the entire spectrum, from simple and intuitive to complex and perplexing. Resolutions can be carried out silently by the link-editor, be accompanied by warning diagnostics, or result in a fatal error condition.

The resolution of two symbols depends on the symbols' attributes, the type of file providing the symbol and the type of file being generated. For a complete description of symbol attributes see "Symbol Table" on page 162. For the following discussions, however, it is worth identifying three basic symbol types:

- *Undefined* symbols. These symbols have been referenced in a file but have not been assigned a storage address.
- *Tentative* symbols. These symbols have been created within a file but have not yet been sized or allocated in storage. They appear as uninitialized C symbols, or FORTRAN COMMON blocks within the file.
- *Defined* symbols. These symbols have been created and assigned storage addresses and space within the file.

In its simplest form, symbol resolution involves the use of a precedence relationship that has *defined* symbols dominating *tentative* symbols, which in turn dominate *undefined* symbols.

The following C code example shows how these symbol types can be generated (undefined symbols are prefixed with `u_`, tentative symbols are prefixed with `t_`, and defined symbols are prefixed with `d_`):

```
$ cat main.c
extern int      u_bar;
extern int      u_foo();

int             t_bar;
int             d_bar = 1;

d_foo()
{
        return (u_foo(u_bar, t_bar, d_bar));
}
$ cc -o main.o -c main.c
$ nm -x main.o

[Index]   Value      Size      Type  Bind  Other Shndx   Name
.............
[8]     |0x00000000|0x00000000|NOTY |GLOB |0x0  |UNDEF  |u_foo
[9]     |0x00000000|0x00000040|FUNC |GLOB |0x0  |2      |d_foo
[10]    |0x00000004|0x00000004|OBJT |GLOB |0x0  |COMMON |t_bar
[11]    |0x00000000|0x00000000|NOTY |GLOB |0x0  |UNDEF  |u_bar
[12]    |0x00000000|0x00000004|OBJT |GLOB |0x0  |3      |d_bar
```

Simple Resolutions

These symbol resolutions are by far the most common, and result when two symbols with similar characteristics are detected, and one symbol takes precedence over the other. This symbol resolution is carried out silently by the link-editor. For example, for symbols with the same binding, a reference to an *undefined* symbol from one file will be bound to, or satisfied by, a *defined* or *tentative* symbol definition from another file. Or, a *tentative* symbol definition from one file will be bound to a *defined* symbol definition from another file.

Symbols that undergo resolution can have either a global or weak *binding*. Weak bindings have less precedence than global binding, and so symbols with different bindings are resolved according to a slight alteration of the simple rules outlined above. But first, it is worth introducing how weak symbols can be produced.

Weak symbols can be defined individually or as aliases to global symbols using a `pragma` definition:

```
$ cat main.c
#pragma weak    bar
#pragma weak    foo = _foo

int             bar = 1;

_foo()
{
        return (bar);
}
$ cc -o main.o -c main.c
$ nm -x main.o

[Index]   Value      Size      Type  Bind  Other Shndx   Name
..............
[7]     |0x00000000|0x00000004|OBJT |WEAK |0x0  |3      |bar
[8]     |0x00000000|0x00000028|FUNC |WEAK |0x0  |2      |foo
[9]     |0x00000000|0x00000028|FUNC |GLOB |0x0  |2      |_foo
```

Notice that the weak alias `foo` is assigned the same attributes as the global symbol `_foo`. This relationship will be maintained by the link-editor and will result in the symbols being assigned the same *value* in the output image.

In symbol resolution, weak defined symbols will be silently overridden by any global definition of the same name.

Another form of simple symbol resolution occurs between relocatable objects and shared objects, or between multiple shared objects, and is termed *interposition*. In these cases, if a symbol is multiply defined, the relocatable object, or the first definition between multiple shared objects, will be silently taken by the link-editor. The relocatable object's definition, or the first shared

object's definition, is said to *interpose* on all other definitions. This interposition can be used to override the functionality provided by one shared object by a dynamic executable or another shared object.

The combination of weak symbols and interposition provides a very useful programming technique. For example, the standard C library provides several services that you are allowed to redefine. However, ANSI C defines a set of standard services that must be present on the system and cannot be replaced in a strictly conforming program.

The function `fread(3S)`, for example, is an ANSI C library function, whereas the system function `read(2)` is not. A conforming ANSI C program must be able to redefine `read(2)`, and still use `fread(3S)` in a predictable way.

The problem here is that `read(2)` underlies the `fread(3S)` implementation in the standard C library, and so it would seem that a program that redefines `read(2)` might confuse the `fread(3S)` implementation. To guard against this, ANSI C states that an implementation cannot use a name that is not reserved to it, and by using the `pragma` directive shown below:

```
# pragma weak read = _read
```

you can define just such a reserved name, and from it generate an alias for the function `read(2)`. Thus, you can quite freely define your own `read()` function without compromising the `fread(3S)` implementation, which in turn is implemented to use the `_read()` function.

The link-editor will not complain of your redefinition of `read()`, either when linking against the shared object or archive version of the standard C library. In the former case, interposition will take its course, whereas in the latter case, the fact that the C library's definition of `read(2)` is weak allows it to be quietly overridden.

By using the link-editor's `-m` option, a list of all interposed symbol references, along with section load address information, is written to the standard output.

Complex Resolutions

Complex resolutions occur when two symbols of the same name are found with differing attributes. In these cases the link-editor will select the most appropriate symbol and will generate a warning message indicating the symbol, the attributes that conflict, and the identity of the file from which the symbol definition is taken. For example:

```
$ cat foo.c
int array[1];

$ cat bar.c
int array[2] = { 1, 2 };

$ cc -dn -r -o temp.o foo.c bar.c
ld: warning: symbol `array' has differing sizes:
        (file foo.o value=0x4; file bar.o value=0x8);
        bar.o definition taken
```

Here, two files with a definition of the data item `array` have different size requirements. A similar diagnostic is produced if the symbols' alignment requirements differed. In both of these cases the diagnostic can be suppressed by using the link-editor's `-t` option.

Another form of attribute difference is the symbol's *type*. For example:

```
$ cat foo.c
bar()
{
        return (0);
}
$ cc -o libfoo.so -G -K pic foo.c
$ cat main.c
int     bar = 1;

main()
{
        return (bar);
}
$ cc -o main main.c -L. -lfoo
ld: warning: symbol `bar' has differing types:
        (file main.o type=OBJT; file ./libfoo.so type=FUNC);
        main.o definition taken
```

Here the symbol `bar` has been defined as both a data item and a function.

Note – *types* in this context are the symbol types that can be expressed in ELF. They are *not* related to the data types as employed by the programming language except in the crudest fashion.

In cases like this, the relocatable object definition will be taken when the resolution occurs between a relocatable object and a shared object, or, the first definition will be taken when the resolution occurs between two shared objects. When such resolutions occur between symbols of different bindings (*weak* or *global*), a warning will also be produced.

Inconsistences between symbol types are not suppressed by the link-editor's `-t` option.

Fatal Resolutions

Symbol conflicts that cannot be resolved result in a fatal error condition. In this case an appropriate error message is provided indicating the symbol name together with the names of the files that provided the symbols, and no output

file will be generated. Although the fatal condition is sufficient to terminate the link-edit, *all* input file processing will first be completed. In this manner all fatal resolution errors can be identified.

The most common fatal error condition exists when two relocatable objects both define symbols of the same name, and neither symbol is a weak definition:

```
$ cat foo.c
int bar = 1;

$ cat bar.c
bar()
{
        return (0);
}

$ cc -dn -r -o temp.o foo.c bar.c
ld: fatal: symbol `bar' is multiply defined:
        (file foo.o and file bar.o);
ld: fatal: File processing errors.  No output written to int.o
```

Here `foo.c` and `bar.c` have conflicting definitions for the symbol `bar`. Since the link-editor cannot determine which should dominate, it will usually give up. However, the link-editor's `-z muldefs` option can be used to suppress this error condition, and allows the first symbol definition to be taken.

Undefined Symbols

After all input files have been read and all symbol resolution is complete, the link-editor will search the internal symbol table for any symbol references that have not been bound to symbol definitions. These symbol references are referred to as *undefined* symbols. The effect of these undefined symbols on the link-edit process can vary according to the type of output file being generated, and possibly the type of symbol.

Generating an Executable

When the link-editor is generating an executable output file, the link-editor's default behavior is to terminate the link-edit with an appropriate error message should any symbols remain undefined. A symbol remains undefined when a symbol *reference* in a relocatable object is never matched to a symbol *definition*:

```
$ cat main.c
extern int foo();
main()
{
        return (foo());
}

$ cc -o prog main.c
Undefined                       first referenced
 symbol                             in file
foo                                 main.o
ld: fatal: Symbol referencing errors. No output written to prog
```

In a similar manner, a symbol reference within a shared object that is never matched to a symbol definition when the shared object is being used to build a dynamic executable, will also result in an undefined symbol:

```
$ cat foo.c
extern int bar;
foo()
{
        return (bar);
}

$ cc -o libfoo.so -G -K pic foo.c
$ cc -o prog main.c -L. -lfoo
Undefined                       first referenced
 symbol                             in file
bar                                 ./libfoo.so
ld: fatal: Symbol referencing errors. No output written to prog
```

If you wish to allow undefined symbols, as in cases like the previous example, then the default fatal error condition can be suppressed by using the link-editor's `-z nodefs` option.

Note – Care should be taken when using the `-z nodefs` option. If an unavailable symbol reference is required during the execution of a process, a fatal runtime relocation error will occur. Although this error can be detected during the initial execution and testing of an application, more complex execution paths can result in this error condition taking much longer to detect, which can be time consuming and costly.

Symbols can also remain undefined when a symbol reference in a relocatable object is bound to a symbol definition in an *implicitly* defined shared object. For example, continuing with the files `main.c` and `foo.c` used in the previous example:

```
$ cat bar.c
int bar = 1;

$ cc -o libbar.so -R. -G -K pic bar.c -L. -lfoo
$ ldd libbar.so
        libfoo.so =>     ./libfoo.so

$ cc -o prog main.c -L. -lbar
Undefined                       first referenced
 symbol                             in file
foo                                 main.o  (symbol belongs to
implicit dependency ./libfoo.so)
ld: fatal: Symbol referencing errors. No output written to prog
```

Here `prog` is being built with an *explicit* reference to `libbar.so`, and because `libbar.so` has a dependency on `libfoo.so`, an *implicit* reference to `libfoo.so` from `prog` is established.

Because `main.c` made a specific reference to the interface provided by `libfoo.so`, then `prog` really has a dependency on `libfoo.so`. However, *only* explicit shared object dependencies are recorded in the output file being generated. Thus, `prog` will fail to run if a new version of `libbar.so` is developed that no longer has a dependency on `libfoo.so`.

For this reason, bindings of this type are deemed fatal, and the implicit reference must be made explicit by referencing the library directly during the link-edit of `prog` (the required reference is hinted at in the fatal error message shown in this example).

Generating a Shared Object

When the link-editor is generating a shared object, it will by default allow undefined symbols to remain at the end of the link-edit. This allows the shared object to import symbols from either relocatable objects or other shared objects when it is used to build a dynamic executable. The link-editor's `-z defs` option can be used to force a fatal error if any undefined symbols remain.

Weak Symbols

Weak symbol references that are not bound during a link-edit will not result in a fatal error condition, no matter what output file type is being generated.

If a static executable is being generated, the symbol will be converted to an absolute symbol and assigned a value of zero.

If a dynamic executable or shared object is being produced, the symbol will be left as an undefined weak reference. During process execution, the runtime linker will search for this symbol, and if it does not find a match, will bind the reference to an address of zero instead of generating a fatal runtime relocation error.

Within the confines of position-independent code (see section "Position-Independent Code" on page 100 for more information), these undefined weak referenced symbols can provide a useful mechanism for testing for the existence of functionality. For example, let's take the following C code fragment which exists in the shared object `libfoo.so.1`:

```
#pragma weak    foo

extern  void    foo(char *);

void
bar(char * path)
{
        void (* fptr)();

        if ((fptr = foo) != 0)
                (* fptr)(path);
}
```

When an application is built that references `libfoo.so.1`, the link-edit will successfully complete regardless of whether a definition for the symbol `foo` is found. If, during execution of the application the function address tests nonzero, the function will be called. However, if the symbol definition is not found, the function address will test zero, and so will not be called.

Tentative Symbol Order Within the Output File

Contributions from input files usually appear in the output file in the order of their contribution. An exception occurs when processing tentative symbols and their associated storage. These symbols are not fully defined until their resolution is complete. If the resolution occurs as a result of encountering a *defined* symbol from a relocatable object, then the order of appearance will be that which would have occurred for the definition.

If it is desirable to control the ordering of a group of symbols, then any tentative definition should be redefined to a zero-initialized data item. For example, the following tentative definitions result in a reordering of the data items within the output file compared to the original order described in the source file `foo.c`:

```
$ cat foo.c
char A_array[0x10];
char B_array[0x20];
char C_array[0x30];
$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]    |0x00020754|0x00000010|OBJT |GLOB |0x0  |15  |A_array
[34]    |0x00020764|0x00000030|OBJT |GLOB |0x0  |15  |C_array
[42]    |0x00020794|0x00000020|OBJT |GLOB |0x0  |15  |B_array
```

By defining these symbols as initialized data items, the relative ordering of these symbols within the input file is carried over to the output file:

```
$ cat foo.c
char A_array[0x10] = { 0 };
char B_array[0x20] = { 0 };
char C_array[0x30] = { 0 };
$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]    |0x000206bc|0x00000010|OBJT |GLOB |0x0  |12  |A_array
[42]    |0x000206cc|0x00000020|OBJT |GLOB |0x0  |12  |B_array
[34]    |0x000206ec|0x00000030|OBJT |GLOB |0x0  |12  |C_array
```

Defining Additional Symbols

Besides the symbols provided from any input files, you can also supply additional symbol references or definitions to a link-edit. In the simplest form, symbol references can be generated using the link-editor's `-u` option. Greater flexibility is provided with the link-editor's `-M` option and an associated `mapfile` which allows you to define symbol references and a variety of symbol definitions.

The -u option provides a mechanism for generating a symbol reference from the link-edit command line. This option can be used to perform a link-edit entirely from archives, or to provide additional flexibility in selecting the objects to extract from multiple archives (see section "Archive Processing" on page 12 for an overview of archive extraction).

For example, let's take the generation of a dynamic executable from the relocatable object `main.o` which makes reference to the symbols `foo` and `bar`. You wish to obtain the symbol definition `foo` from the relocatable object `foo.o` contained in `lib1.a`, and the symbol definition `bar` from the relocatable object `bar.o` contained in `lib2.a`.

However, the archive `lib1.a` also contains a relocatable object defining the symbol `bar` (presumably of differing functionality to that provided in `lib2.a`). To specify the required archive extraction, the following link-edit can be used:

```
$ cc -o prog -L. -u foo -l1 main.o -l2
```

Here, the -u option generates a reference to the symbol `foo`. This reference will cause extraction of the relocatable object `foo.o` from the archive `lib1.a`. As the first reference to the symbol `bar` occurs in `main.o`, which is encountered after `lib1.a` has been processed, the relocatable object `bar.o` will be obtained from the archive `lib2.a`.

Note – This simple example assumes that the relocatable object `foo.o` from `lib1.a` does not directly, or indirectly, reference the symbol `bar`. If it did then the relocatable object `bar.o` will also be extracted from `lib1.a` during its processing (see section "Archive Processing" on page 12 for a discussion of the link-editor's multi-pass processing of an archive).

A more extensive set of symbol definitions can be provided using the link-editor's -M option and an associated `mapfile`. The syntax for these `mapfile` entries is:

```
[ name ] {
     scope:
             symbol [ = [ type ] [ value ] [ size ] ];
};
```

- *name* represents a label for this set of symbol definitions, and if present, identifies a *version definition* within the image. See Chapter 5, "Versioning" for more details.
- *scope* indicates the visibility of the symbols' binding within the output file being generated. This can have either the value **local** or **global**. All symbols defined with a `mapfile` are treated as global in scope during the link-edit process. That is, they will be resolved against any other symbols of the same name obtained from any of the input files. However, symbols defined as **local** scope will be reduced to symbols with a local binding within any executable or shared object file being generated.
- *symbol* is the name of the symbol required. If the name is not followed by any symbol attributes then the result will be the creation of a symbol reference. This reference is exactly the same as would be generated using the `-u` option discussed earlier in this section. If the symbol name is followed by an optional "=" character then a symbol definition will be generated using the associated attributes.

  When in **local** scope, this symbol name can be defined as the special *auto-reduction* directive "*". This directive results in all global symbols, not explicitly defined to be **global** in the `mapfile`, being given a local binding within any executable or shared object file being generated.
- *type* indicates the symbols' type attribute and can be either **data**, **function** or **common**. The former two type attributes result in an *absolute* symbol definition (see "Symbol Table" on page 162). The latter type attribute results in a *tentative* symbol definition.
- *value* indicates the symbols' value attribute and takes the form of **V***number*.
- *size* indicates the symbols' size attribute and takes the form of **S***number*.

If either a *version definition* or the *auto-reduction* directive is specified, then versioning information is recorded in the image created. If this image is an executable or shared object, then any symbol reduction is also applied.

If the image being created is a relocatable object, then by default no symbol reduction is applied. In this case, any symbol reductions are recorded as part of the versioning information, and these reductions will be applied when the relocatable object is finally used to generate an executable or shared object.

A more detailed description of the versioning information is provided in Chapter 5, "Versioning".

The remainder of this section presents several examples of using this `mapfile` syntax.

The following example shows how three symbol references can be defined and used to extract members of an archive. Although this archive extraction can be achieved by specifying multiple `-u` options to the link-edit, this example also shows how the eventual scope of a symbol can be reduced to *local*:

```
$ cat foo.c
foo()
{
        (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
bar()
{
        (void) printf("bar: called from lib.a\n");
}
$ cat main.c
extern  void    foo(), bar();

main()
{
        foo();
        bar();
}
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
{
        local:
                foo;
                bar;
        global:
                main;
};
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ nm -x prog | egrep "main$|foo$|bar$"
[28]    |0x00010604|0x00000024|FUNC |LOCL |0x0  |7      |foo
[30]    |0x00010628|0x00000024|FUNC |LOCL |0x0  |7      |bar
[49]    |0x0001064c|0x00000024|FUNC |GLOB |0x0  |7      |main
```

The significance of reducing a symbol's scope from *global* to *local* is covered in more detail in the section "Reducing Symbol Scope" on page 38.

The following example shows how two absolute symbol definitions can be defined and used to resolve the references from the input file `main.c`:

```
$ cat main.c
extern  int     foo();
extern  int     bar;

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = FUNCTION V0x400;
                bar = DATA V0x800;
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 400
&bar = 800
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00000800|0x00000000|OBJT |GLOB |0x0  |ABS    |bar
[42]    |0x00000400|0x00000000|FUNC |GLOB |0x0  |ABS    |foo
```

When obtained from an input file, symbol definitions for functions or data items are usually associated with elements of data storage. As a `mapfile` definition is insufficient to be able to construct this data storage, these symbols must remain as absolute values.

However, a `mapfile` can also be used to define a **common**, or *tentative*, symbol. Unlike other types of symbol definition, tentative symbols do not occupy storage within a file, but define storage that must be allocated at run-time. Therefore, symbol definitions of this kind can contribute to the storage allocation of the output file being generated.

A feature of tentative symbols, that differs from other symbol types, is that their *value* attribute indicates their alignment requirement. A `mapfile` definition can therefore be used to realign tentative definitions obtained from the input files of a link-edit.

The following example shows the definition of two tentative symbols. The symbol `foo` defines a new storage region whereas the symbol `bar` is actually used to change the alignment of the same tentative definition within the file `main.c`:

```
$ cat main.c
extern  int     foo;
int             bar[0x10];

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = COMMON V0x4 S0x200;
                bar = COMMON V0x100 S0x40;
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol `bar' has differing alignments:
        (file mapfile value=0x100; file main.o value=0x4);
        largest value applied
$ prog
&foo = 20940
&bar = 20900
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00020900|0x00000040|OBJT |GLOB |0x0  |16     |bar
[42]    |0x00020940|0x00000200|OBJT |GLOB |0x0  |16     |foo
```

Note – The above symbol resolution diagnostic can be suppressed by using the link-editor's `-t` option.

Reducing Symbol Scope

In the previous section it was shown how symbol definitions defined to have local scope within a `mapfile` can be used to reduce the symbol's eventual binding. This mechanism can play an important role in reducing the symbol's visibility to future link-edits that use the generated file as part of their input. In fact, this mechanism can provide for the precise definition of a file's interface, and so restrict the functionality made available to others.

For example, let's take the generation of a simple shared object from the files `foo.c` and `bar.c`. The file `foo.c` contains the global symbol `foo` which provides the service that you wish to make available to others. The file `bar.c` contains the symbols `bar` and `str` which provide the underlying implementation of the shared object. A simple build of the shared object will usually result in all three of these symbols having global scope:

```
$ cat foo.c
extern  const char *    bar();

const char * foo()
{
        return (bar());
}
$ cat bar.c
const char * str = "returned from bar.c";

const char * bar()
{
        return (str);
}
$ cc -o lib.so.1 -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[29]    |0x000104d0|0x00000004|OBJT |GLOB |0x0  |12      |str
[32]    |0x00000418|0x00000028|FUNC |GLOB |0x0  |6       |bar
[33]    |0x000003f0|0x00000028|FUNC |GLOB |0x0  |6       |foo
```

You can now use the functionality offered by this shared object as part of the link-edit of another application. References to the symbol `foo` will be bound to the implementation provided by the shared object.

However, because of their global binding, direct reference to the symbols `bar` and `str` is also possible. This can have dangerous consequences, as the you might later change the implementation underlying the function `foo`, and in so doing unintentionally cause an existing application that had bound to `bar` or `str` fail or misbehave.

Another consequence of the global binding of the symbols `bar` and `str` is that they can be *interposed* upon by symbols of the same name (the interposition of symbols within shared objects is covered in section "Simple Resolutions" on page 22). This interposition can be intentional and be used as a means of circumventing the intended functionality offered by the shared object. On the other hand, this interposition can be unintentional, and simply be the result of the application and the shared object using the same common symbol name.

When developing the shared object you can protect against this type of scenario by reducing the scope of the symbols `bar` and `str` to a local binding, for example:

```
$ cat mapfile
{
        local:
                bar;
                str;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[27]    |0x000003dc|0x00000028|FUNC |LOCL |0x0  |6       |bar
[28]    |0x00010494|0x00000004|OBJT |LOCL |0x0  |12      |str
[33]    |0x000003b4|0x00000028|FUNC |GLOB |0x0  |6       |foo
```

Here the symbols `bar` and `str` are no longer available as part of the shared objects interface. Thus these symbols cannot be referenced, or interposed upon, by an external object. You have effectively defined an interface for the shared object. This interface can be managed while hiding the details of the underlying implementation.

This symbol scope reduction has an additional performance advantage. The symbolic relocations against the symbols `bar` and `str` that would have been necessary at runtime are now reduced to relative relocations. This reduces the runtime overhead of initializing and processing the shared object (see section "When Relocations are Performed" on page 106 for details of symbolic relocation overhead).

As the number of symbols processed during a link-edit gets large, the ability to define each local scope reduction within a `mapfile` becomes harder to maintain. An alternative, and more flexible mechanism, allows you to define the shared objects interface in terms of the global symbols that should

be maintained, and instructs the link-editor to reduce all other symbols to local binding. This mechanism is achieved using the special *auto-reduction* directive "*". For example, the previous `mapfile` definition can be rewritten to define `foo` as the only global symbol required in the output file generated:

```
$ cat mapfile
lib.so.1.1 {
        global:
                foo;
        local:
                *;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[30]    |0x00000370|0x00000028|FUNC |LOCL |0x0  |6       |bar
[31]    |0x00010428|0x00000004|OBJT |LOCL |0x0  |12      |str
[35]    |0x00000348|0x00000028|FUNC |GLOB |0x0  |6       |foo
```

This example also defines a version name, `lib.so.1.1`, as part of the `mapfile` directive. This version name establishes an internal *version definition* that defines the files symbolic interface. The creation of a version definition is recommended, and forms the foundation of an internal versioning mechanism that can be used throughout the evolution of the file. See Chapter 5, "Versioning" for more details on this topic.

Whenever a version name is specified, *all* global symbols must be assigned to a version definition. If any global symbols remain unassigned to a version definition the link-editor will generate a fatal error condition:

```
 $ cat mapfile
lib.so.1.1 {
        global:
                foo;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
Undefined                   first referenced
 symbol                        in file
str                         bar.o  (symbol has no version assigned)
bar                         bar.o  (symbol has no version assigned)
ld: fatal: Symbol referencing errors. No output written to
lib.so.1
```

When generating an executable or shared object, any symbol reduction results in the recording of version definitions within the output image, together with the reduction of the appropriate symbols. By default, when generating a relocatable object, the version definitions are created, but the symbol reductions are not processed. The result is that the symbol entries for any symbol reductions will still remain global. For example, using the previous `mapfile` and associated relocatable objects, an intermediate relocatable object is created which shows no symbol reduction:

```
$ ld -o lib.o -M mapfile -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[17]    |0x00000000|0x00000004|OBJT |GLOB |0x0  |3      |str
[19]    |0x00000028|0x00000028|FUNC |GLOB |0x0  |1      |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1      |foo
```

However, the *version definitions* created within this image record the fact that symbol reductions are required. When the relocatable object is eventually used to generate an executable or shared object, the symbol reductions will occur. In other words, the link-editor reads and interprets symbol reduction information contained in relocatable objects in the same manner as it can process the data from a `mapfile`.

Thus, the intermediate relocatable object produced in the previous example can now be used to generate a shared object:

```
$ cc -o lib.so.1 -G lib.o
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[22]    |0x000104a4|0x00000004|OBJT |LOCL |0x0  |14     |str
[24]    |0x000003dc|0x00000028|FUNC |LOCL |0x0  |8      |bar
[36]    |0x000003b4|0x00000028|FUNC |GLOB |0x0  |8      |foo
```

Symbol reductions can be forced to occur when building a relocatable object by using the link-editor's `-B reduce` option:

```
$ ld -o lib.o -M mapfile -B reduce -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[15]    |0x00000000|0x00000004|OBJT |LOCL |0x0  |3      |str
[16]    |0x00000028|0x00000028|FUNC |LOCL |0x0  |1      |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1      |foo
```

Generating the Output Image

Once all input file processing and symbol resolution is completed with no fatal errors, the link-editor will start generating the output file image.

The link-editor establishes what additional *sections* must be generated to complete the output file image. These include the symbol tables that contain local symbol definitions from the input files, together with the global and weak symbol information that has been collected in its internal symbol table.

Also included are any output relocation and dynamic information sections required by the runtime linker. Once all the output section information has been established, the total output file size is calculated and the output file image is created accordingly.

When building a dynamic executable or shared object, two symbol tables are usually generated. The *.dynsym*, and its associated string table *.dynstr*, contain only global, weak and section symbols. These sections become part of the *text* segment which is mapped as part of the process image at runtime. This allows the runtime linker to read these sections and perform any necessary relocations.

The *.symtab*, and its associated string table *.strtab*, contain *all* the symbols collected from the input file processing. These sections are not mapped as part of the process image, and can even be stripped from the image using the link-editor's `-s` option, or after the link-edit using `strip(1)`.

During the generation of the symbol tables *reserved* symbols are created. These have special meaning to the linking process and should not be defined in your code:

- *_etext*, the first location after the text segment.
- *_edata*, the first location after initialized data.
- *_end*, the first location after all data.
- *_DYNAMIC*, the address of the dynamic information section (the *.dynamic* section).
- *_GLOBAL_OFFSET_TABLE_*, the position-independent reference to a link-editor supplied table of addresses (the *.got* section). This table is constructed from position-independent *data* references occurring in objects that have been compiled with the `-K pic` option (see "Position-Independent Code" on page 100 for more information).
- *_PROCEDURE_LINKAGE_TABLE_*, the position-independent reference to a link-editor supplied table of addresses (the *.plt* section). This table is constructed from position-independent *function* references occurring in objects that have been compiled with the `-K pic` option (see "Position-Independent Code" on page 100 for more information).

If the link-editor is generating an executable, it will look for additional symbols to define the executable's entry point. If a symbol was specified using the link-editor's `-e` option it will be used. Otherwise the link-editor will look for the reserved symbol names `_start`, and then `main`. If none of these symbols exists, the first address of the *text* segment will be used.

Having created the output file, all data sections from the input files are copied to the new image. Any relocations specified in the input files are applied to the output image. Any new relocation information that must be generated, together with all the other link-editor generated information, is also written to the new image.

2

Link-Editor Support Interface

The link-editor provides a support library interface that allows information regarding the link-edit to be obtained. This facility provides for input file inspection, and to some degree, input file data modification of those files that comprise a link-edit. You should be intimately familiar with the `elf(3E)` structures and file format when using this interface.

Invoking the Support Interface

The link-editor will accept one or more support libraries provided by the `SGS_SUPPORT` environment variable or with the link-editor's `-S` option. The environment variable consists of a colon separated list of support libraries:

```
$ SGS_SUPPORT=./support.so.1 cc ...
```

The `-S` option specifies a single support library. Multiple `-S` options can be specified:

```
$ ld -S ./support.so.1 -S libldstab.so.1 ...
```

Each support library represents a shared object. The link-editor performs a `dlopen(3X)` on each shared object, in the order they are specified. If both the environment variable and `-S` options are encountered then the shared objects specified with the environment variable are processed first. Each support library is then searched, using `dlsym(3X)`, for any support interface routines. These support routines are then called at various stages of the link-editing process.

By default, the Solaris support library `libldstab.so.1` is used by the link-editor to process compiler generated debugging information contained within input relocatable objects. This default processing is suppressed if you invoke the link-editor with any support libraries specified using the `-S` option. If this default processing is required in addition to your support library services, then `libldstab.so.1` should be explicitly added to the list of support libraries supplied to the link-editor.

Support Interface Functions

The following interface functions can be provided by a support library. These interfaces are defined in the header file `link.h`. All interface arguments are basic C types or ELF types (see `elf(3E)`). The ELF data types can be examined with the elf access library `libelf` (see *man Pages(3): Library Routines*" for a description of `libelf` contents).

The function `ld_start()` is called after the initial pass of the link-editor command line. It indicates the output file that will be generated, and flags the start of input file processing:

```
void ld_start(const char * name, const Elf32_Half etype, const
     char * caller);
```

name is the output filename being created. *etype* is the output file type, which is either `ET_DYN`, `ET_REL`, or `ET_EXEC` (as defined in `sys/elf.h`). *caller* is the application calling the interface which in this case is `ld`.

The function `ld_atexit()` is called on completion of the link-edit:

```
void ld_atexit(int status);
```

status is the `exit(2)` code that will be returned by the link-editor and is either `EXIT_FAILURE`, or `EXIT_SUCCESS` (as defined in `stdlib.h`).

The function `ld_file()` is called for each input file processed. The call is made before any processing of the files data is carried out:

```
void ld_file(const char * name, const Elf_Kind kind, int flags,
     Elf * elf);
```

name is the input file about to be processed. *kind* indicates the input files' type which is either `ELF_K_AR`, or `ELF_K_ELF` (as defined in `libelf.h`). *flags* provides more detailed information on how the link-editor came about obtaining the file and can be either `LD_SUP_DERIVED` (the file name was derived from a `-l` expansion), `LD_SUP_INHERITED` (the file was obtained as a dependency of a command-line shared object), or `LD_SUP_EXTRACTED` (the file

was extracted from an archive). If no *flags* values are specified then the input file has been explicitly named on the command-line. *elf* is a pointer to the files ELF descriptor.

The function `ld_section()` is called for each section of the input file before any processing of the sections data is carried out:

```
void ld_section(const char * name, Elf32_Shdr * shdr,
     Elf32_Word sndx, Elf_Data * data, Elf * elf);
```

name is the input section name. *shdr* is a pointer to the associated section header. *sndx* is the sections index within the input file. *data* is a pointer to the associated data buffer. *elf* is a pointer to the files ELF descriptor.

Modification of the data is permitted by reallocating the data itself and reassigning the `Elf_Data` buffers `d_buf` pointer. Any modification to the data should insure the correct setting of the `Elf_Data` buffers `d_size` element. For input sections that will become part of the output image, setting the `d_size` element to zero will effectively remove the data from the output image.

Note – Any sections that are stripped by the use of the link-editors `-s` option will not be reported to any `ld_section()` routines.

Support Interface Example

The following example creates a support library that prints the section names of any relocatable object files processed as part of a link-edit.

```
$ cat support.c
#include        <link.h>

static int      indent = 0;

void
ld_start(const char * name, const Elf32_Half type,
    const char * caller)
{
        (void) printf("output image: %s\n", name);
}

void
ld_file(const char * name, const Elf_Kind kind, int flags,
    Elf * elf)
{
        if (flags & LD_SUP_EXTRACTED)
                indent += 2;
        else
                indent = 2;

        (void) printf("%*sfile: %s\n", indent, "", name);
}

void
ld_section(const char * name, Elf32_Shdr * shdr, Elf32_Word sndx,
    Elf_Data * data, Elf * elf)
{
        Elf32_Ehdr *    ehdr = elf32_getehdr(elf);

        if (ehdr->e_type == ET_REL)
               (void) printf("%*s   section [%ld]: %s\n", indent,
                     "", sndx, name);
}
```

This support library is dependent upon `libelf` to provide the ELF access function `elf32_getehdr(3E)` which is used to determine the input file type:

```
$ cc -o support.so.1 -G -K pic support.c -lelf
```

The following link-edit shows the section diagnostics resulting from the construction of a trivial application from a relocatable object and a local archive library. The invocation of the support library, in addition to default debugging information processing, is brought about by the `-S` option usage:

```
$ LD_OPTIONS=-S./support.so.1:libldstab.so.1 cc -o prog main.c \
 -L. -lfoo
output image: prog
  file: /opt/COMPILER/crti.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /opt/COMPILER/crt1.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /opt/COMPILER/values-xt.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: main.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: ./libfoo.a
    file: ./libfoo.a(foo.o)
       section [1]: .shstrtab
       section [2]: .text
       .......
  file: /usr/lib/libc.so
  file: /opt/COMPILER/crtn.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /usr/lib/libdl.so.1
```

Note – The number of sections displayed in this example have been reduced to simplify the output. Also, the files included by the compiler driver can vary.

Debugging Aids

Provided with the Solaris linkers is a debugging library that allows you to trace the link-editing process in more detail. This library helps you understand, or debug, the link-edit of your own applications or libraries. This is a *visual* aid, and although the type of information displayed using this library is expected to remain constant, the exact format of the information might change slightly from release to release.

Some of the debugging output might be unfamiliar if you do not have an intimate knowledge of ELF. However, many aspects can be of general interest to you.

Debugging is enabled by using the -D option, and all output produced is directed to the standard error. This option must be augmented with one or more tokens to indicate the type of debugging required. The tokens available can be displayed by using -Dhelp. For example:

```
$ ld -Dhelp
debug:
debug:           For debugging the link-editing of an application:
debug:                  LD_OPTIONS=-Doption1,option2 cc -o prog ...
debug:           or,
debug:                  ld -Doption1,option2 -o prog ...
debug:          where placement of -D on the command line is significant
debug:           and options can be switched off by prepending with `!'.
debug:
debug:
debug: args      display input argument processing
debug: detail    provide more information in conjunction with other
debug:           options
debug: entry     display entrance criteria descriptors
debug: files     display input file processing (files and libraries)
debug: help      display this help message
debug: libs      display library search paths; detail flag shows actual
debug:           library lookup (-l) processing
debug: map       display map file processing
debug: reloc     display relocation processing
debug: sections  display input section processing
debug: segments  display available output segments and address/offset
debug:           processing; detail flag shows associated sections
debug: support   display support library processing
debug: symbols   display symbol table processing;
debug:           detail flag shows resolution and linker table addition
debug: versions  display version processing
```

Note – The above is an example, and shows the options meaningful to the link-editor. The exact options might differ from release to release.

As most compiler drivers will interpret the -D option during their preprocessing phase, the LD_OPTIONS environment variable is a suitable mechanism for passing this option to the link-editor.

The following example shows how input files can be traced. This can be especially useful in determining what libraries have been located, or what relocatable objects have been extracted from an archive during a link-edit:

```
$ LD_OPTIONS=-Dfiles cc -o prog main.o -L. -lfoo
.............
debug: file=main.o  [ ET_REL ]
debug: file=./libfoo.a  [ archive ]
debug: file=./libfoo.a(foo.o)  [ ET_REL ]
debug: file=./libfoo.a  [ archive ] (again)
.............
```

Here the member `foo.o` is extracted from the archive library `libfoo.a` to satisfy the link-edit of `prog`. Notice that the archive is searched twice (again) to verify that the extraction of `foo.o` did not warrant the extraction of additional relocatable objects. More than one "again" display indicates that the archive is a candidate for ordering using `lorder(1)` and `tsort(1)`.

By using the `symbols` token you can determine what symbol caused an archive member to be extracted, and which object made the initial symbol reference:

```
$ LD_OPTIONS=-Dsymbols cc -o prog main.o -L. -lfoo
.............
debug: symbol table processing; input file=main.o  [ ET_REL ]
.............
debug: symbol[7]=foo  (global); adding
debug:
debug: symbol table processing; input file=./libfoo.a  [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo  (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo(foo.o)  [ ET_REL ]
.............
```

Here the symbol `foo` is referenced by `main.o` and is added to the link-editor's internal symbol table. This symbol reference causes the extraction of the relocatable object `foo.o` from the archive `libfoo.a`.

Note – The above output has been simplified for this document.

By using the `detail` token together with the `symbols` token, the details of symbol resolution during input file processing can be observed:

```
$ LD_OPTIONS=-Dsymbols,detail cc -o prog main.o -L. -lfoo
.............
debug: symbol table processing; input file=main.o  [ ET_REL ]
.............
debug: symbol[7]=foo  (global); adding
debug:   entered  0x000000 0x000000 NOTY GLOB  UNDEF REF_REL_NEED
debug:
debug: symbol table processing; input file=./libfoo.a  [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo  (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo.a(foo.o)  [ ET_REL ]
debug: symbol[1]=foo.c
.............
debug: symbol[7]=bar  (global); adding
debug:   entered  0x000000 0x000004 OBJT GLOB  3     REF_REL_NEED
debug: symbol[8]=foo  (global); resolving [7][0]
debug:       old  0x000000 0x000000 NOTY GLOB  UNDEF main.o
debug:       new  0x000000 0x000024 FUNC GLOB  2     ./libfoo.a(foo.o)
debug:  resolved  0x000000 0x000024 FUNC GLOB  2     REF_REL_NEED
```

Here, the original undefined symbol `foo` from `main.o` has been overridden with the symbol definition from the extracted archive member `foo.o`. The detailed symbol information reflects the attributes of each symbol.

From the above example, it should be apparent that using some of the debugging tokens can produce a wealth of output. In cases where are interested only in the activity around a subset of the input files, the `-D` option can be placed directly in the link-edit command-line, and toggled on and off. For example:

```
$ ld .... -o prog main.o -L. -Dsymbols -lbar -D!symbols ....
```

Here the display of symbol processing will be switched on *only* during the processing of the library `libbar`.

Note – To obtain the link-edit command-line it might be necessary to expand the compilation line from any driver being used. See "Using a Compiler Driver" on page 9 for more details.