Topics

8	**Linker Description**	**8-5**
8.1	Linker Development Flow	8-6
8.2	Invoking the Linker	8-7
8.3	Linker Options	8-9
8.3.1	Relocation Capabilities (-a and -r Options)	8-10
8.3.2	C Language Options (-c and -cr Options)	8-11
8.3.2	Define an Entry Point (-e global symbol Option)	8-11
8.3.3	Set Default Fill Value (-f cc Option)	8-11
8.3.4	Make All Global Symbols Static (-h Option)	8-11
8.3.3	Define Heap Size (-heap constant Option)	8-12
8.3.5	Alter the Library Search Algorithm (-i dir Option/C_DIR)	8-12
8.3.6	Create a Map File (-m filename Option)	8-14
8.3.9	Ignore the Memory Directive Fill Specification (-n option)	8-14
8.3.7	Name an Output Module (-o filename Option)	8-14
8.3.8	Specify a Quiet Run (-q Option)	8-14
8.3.9	Strip Symbolic Information (-s Option)	8-15
8.3.4	Define Stack Size (-sstack and -hstack Options)	8-15
8.3.5	Introduce an Unresolved Symbol (-u symbol Option)	8-15
8.3.6	Exhaustively Read Libraries (-x option)	8-15
8.4	Command Files	8-16
8.5	Object Libraries	8-19
8.6	The MEMORY Directive	8-20
8.6.1	Default Memory Model	8-20
8.6.2	MEMORY Directive Syntax	8-20
8.7	The SECTIONS Directive	8-23
8.7.1	Default Sections Configuration	8-23
8.7.2	SECTIONS Directive Syntax	8-23
8.7.3	Specifying the Address of Output Sections (Allocation)	8-26
8.7.4	Specifying Input Sections	8-28
8.8	Specifying a Section’s Runtime Address	8-31
8.8.1	Specifying Two Addresses	8-31
8.8.2	Uninitialized Sections	8-32
8.8.3	Referring to the Load Address by Using the .label Directive	8-32
8.9	Using UNION and GROUP Statements	8-35
8.9.1	Overlaying Sections With the UNION Statement	8-35
8.9.2	Grouping Output Sections Together	8-38
8.10	Overlay Pages	8-39
8.10.1	Using the MEMORY Directive to Define Overlay Pages	8-39
8.10.2	Using Overlay Pages With the SECTIONS Directive	8-39
8.10.3	Syntax of Page Definitions	8-39
8.11	Default Allocation Algorithm	8-39
8.11.1	Default Allocation	8-39

8.11.2 General Rules for Forming Output Sections 8-40
8.12 Special Section Types (DSECT, COPY, and NOLOAD) 8-41
8.13 Assigning Symbols at Link Time 8-42
8.13.1 Syntax of Assignment Statements 8-42
8.13.2 Assigning the SPC to a Symbol 8-42
8.13.3 Assignment Expressions 8-43
8.13.4 Symbols Defined by the Linker 8-44
8.14 Creating and Filling Holes 8-45
8.14.1 Initialized and Uninitialized Sections 8-45
8.14.2 Creating Holes 8-45
8.14.3 Filling Holes 8-47
8.14.4 Explicit Initialization of Uninitialized Sections 8-48
8.14.5 Examples of Using Initialized Holes 8-49
8.15 Partial (Incremental) Linking 8-50
8.16 Linking C Code 8-51
8.16.1 Runtime Initialization 8-51
8.16.2 Object Libraries and Runtime Support 8-51
8.16.3 Setting the Size of the Stack and Heap Sections 8-52
8.16.4 Autoinitialization (ROM and RAM Models) 8-52
8.16.5 The -c and -cr Linker Options 8-52
8.17 Linker Example 8-52

Examples

Ex.	Title	Page
8.1	Linker Command File	8-16
8.2	Command File With Linker Directives	8-17
8.3	The SECTIONS Directive	8-24
8.4	The Most Common Method of Specifying Section Contents	8-28
8.5	Copying a Section From ROM to RAM	8-33
8.6	Illustration of the Form of the UNION Statement	8-35
8.7	Illustration of Separate Load Addresses for UNION Sections	8-37
8.8	Overlay Page	8-39

Figures

Fig.	Title	Page
8.1	Linker Development Flow	8-6
8.2	Section Allocation	8-25
8.3	Runtime Execution	8-34
8.4	Runtime Memory Allocation	8-36
8.5	Load and Run Memory Allocation	8-37
8.6	Initialized Hole	8-49
8.7	RAM Model of Autoinitialization	8-52
8.8	ROM Model of Autoinitialization	8-52
8.7	Linker Command File, demo.cmd	8-53
8.8	Output Map File, demo.map	8-54

Tables

Table	Title	Page
8.1	Linker Options Summary	8-9
8.2	Operators in Assignment Expressions	8-44

Notes

Note	Title	Page
8.1		
	Filling Memory Ranges	8-22
	Compatibility With Previous Versions	8-23
	Binding and Alignment or Named Memory Are Incompatible	8-27
	You Cannot Specify Addresses for Sections Within a Group	8-38
	The Sections Directive	8-39
	Filling Sections	8-48
	Unions and Overlay Pages Are Not the Same	8-54
	The PAGE Option	8-54

8 Linker Description

The MSP430 family linker creates executable modules by combining COFF object files. The concept of COFF *sections* is basic to linker operation.

As the linker combines object files, it:

- allocates sections into the target system's configured memory
- relocates symbols and sections to assign them to final addresses
- resolves undefined external references between input files

The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation and provides two powerful directives, MEMORY and SECTIONS, that allow you to:

- define a memory model that conforms to target system memory
- combine object file sections
- allocate sections into specific areas of memory
- define or redefine global symbols at link time

8.1 Linker Development Flow

The following figure illustrates the linker's role in the assembly language development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable COFF object module that can be downloaded to one of several development tools or executed by a MSP430 device.

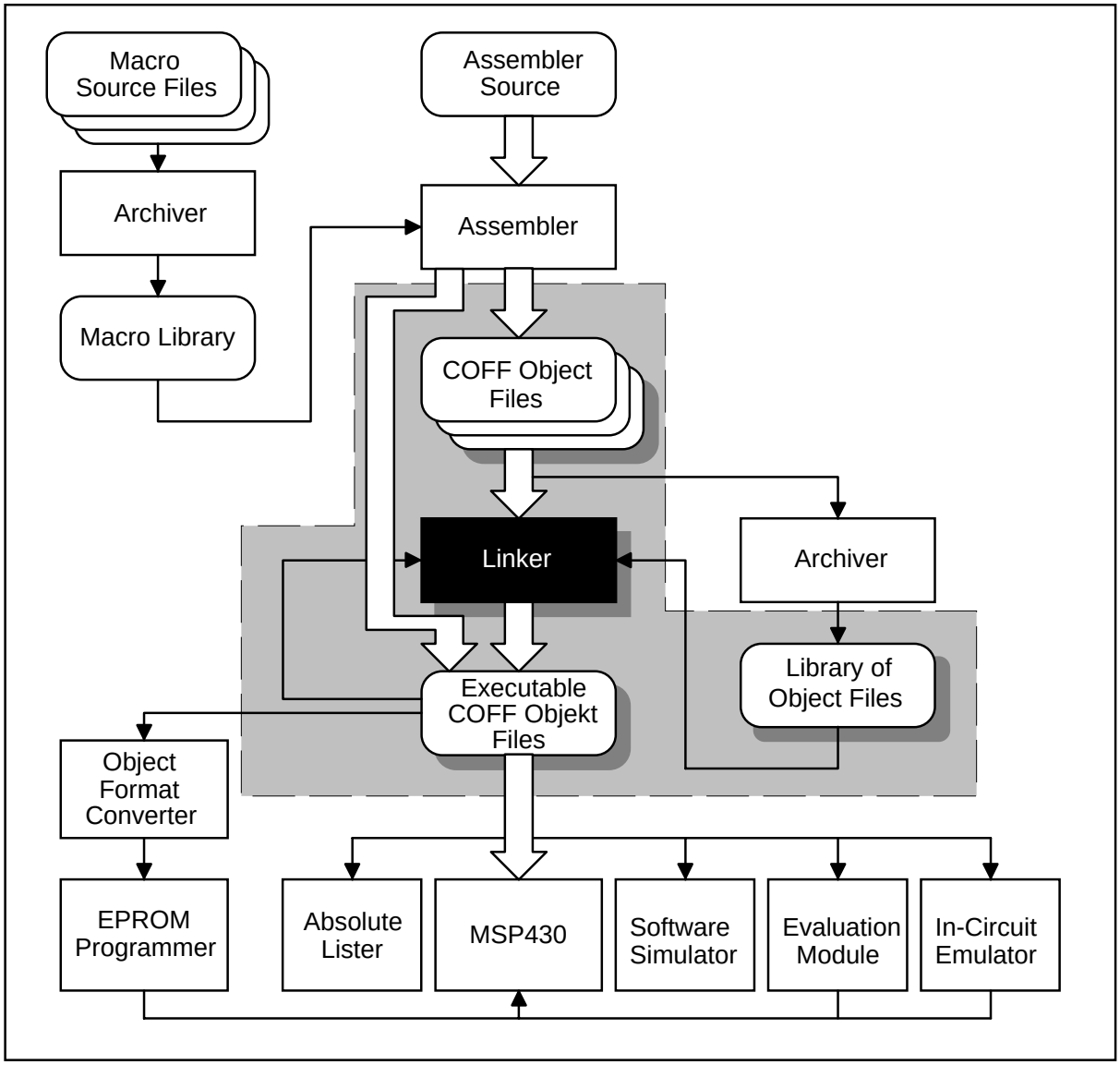

Figure 8.1: Linker Development Flow

8.2 Invoking the Linker

The general syntax for invoking the linker is:

lnk430 [-option] *filename*$_1$ ... *filename*$_n$

lnk430 is the command that invokes the linker.

options

can appear anywhere on the command line or in a linker command file. (Options are discussed in Section 8.3.)

filenames

can be object files, linker command files, or archive libraries. The default extension for all input files is ***.obj***; any other extension must be explicitly specified. The linker can determine whether the input file is an object file or an ASCII file that contains linker commands. The default output filename is ***a.out***.

There are three methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, file1.obj and file2.obj, and creates an output module named link.out.

  ```
  lnk430 file1.obj file2.obj -o link.out
  ```

- Enter the **lnk430** command with no filenames and no options; the linker will prompt for them:

  ```
  Command files :
  Object files [.obj] :
  Output files [ ] :
  Options :
  ```

  For *command files,* enter one or more command file names.

  For *object files,* enter one or more object file names. The default extension is *.obj.* Separate the filenames with spaces or commas; if the last character is a comma, the linker will prompt for an additional line of object file names.

  The *output file* is the name of the linker output module. This overrides any -o options entered with any of the other prompts. If there are no -o options and you do not answer this prompt, the linker will create an object file with a default filename of ***a.out***.

  The *options* prompt is for additional options, although you can also enter them in a command file. Enter them with hyphens, just as you would on the command line.

- Put filenames and options in a linker command file. For example, assume the file linker.cmd contains the following lines:

  ```
  -o link.out
  file1.obj
  file2.obj
  ```

  Now you can invoke the linker from the command line; specify the command file name as an input file:

  ```
  lnk430 linker.cmd
  ```

When you use a command file, you can also specify other options and files on the command line. For example, you could enter:

```
lnk430 -m link.map linker.cmd file3.obj
```

The linker reads and processes a command file as soon as it encounters it on the command line, so it links the files in this order: file1.obj, file2.obj, and file3.obj. This example creates an output file called link.out and a map file called link.map.

8.3 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (-). The order in which options are specified is unimportant, except for the -l and -i options. Options are separated from arguments (if they have them) by an optional space.

Option	Description
-a	Produce an absolute, executable module. This is the default; if neither -a nor -r is specified, the linker acts as if -a is specified.
-ar	Produce a relocatable, executable object module.
-e *global symbol*	Define a *global symbol* that specifies the primary entry point for the output module.
-f *fill value*	Set the default fill value for holes within output sections; *fill value* is a 16–bit constant.
-h	Make all global symbols static.
-i *dir* †	Alter the library–search algorithm to look in *dir* before looking in the default location. This option must appear before the -l option.
-l *filename* †	Name an archive library file as linker input; *filename* is an archive library name.
-m *filename* †	Produce a map or listing of the input and output sections, including holes, and place the listing in *filename.*
-o *filename* †	Name the executable output module. The default filename is a.*out.*
-q	Request a quiet run (suppress the banner).
-r	Retain relocation entries in the output module.
-s	Strip symbol table information and line number entries from the output module.
-u *symbol*	Place an unresolved external *symbol* into the output module's symbol table.
-x	Force rereading of libraries. Resolves “back" references.
-z *filename* †	Produce an additional byte formatted ASCII file loadable by the evaluation module. The default filename is the output filename with the extension .txt.

† The *filename* must follow operating system conventions.

Table 8.1: Linker Options Summary

8.3.1 Relocation Capabilities (-a and -r Options)

One of the tasks the linker performs is *relocation.* Relocation is the process of adjusting all references to a symbol when the symbol's address changes. The linker supports two options (-a and -r) that allow you to produce an absolute or a relocatable output module. Default is -a.

- **Producing an Absolute Output Module (-a Option)**

  When you use the **-a** option without the -r option, the linker produces an *absolute, executable* output module. Absolute files contain *no* relocation information. Executable files contain the following:
  - special symbols defined by the linker
  - an optional header that describes information such as the program entry point
  - *no* unresolved references

  This example links file1.obj and file2.obj and creates an absolute output module called a.out:

  ```
  lnk430 -a file1.obj file2.obj
  ```

- **Producing a Relocatable Output Module (-r Option)**

  When you use the **-r** option without the -a option, the linker retains relocation entries in the output module. If the output module will be relocated (at load time) or relinked (by another linker execution), use -r to retain the relocation entries.

  The linker produces an *unexecutable* file when you use the -r option without -a. A file that is not executable does not contain special linker symbols or an optional header. The file may contain unresolved references, but these references do not prevent creation of an output module.

  This example links file1.obj and file2.obj and creates a relocatable output module called a.out:

  ```
  lnk430 -r file1.obj file2.obj
  ```

  The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking)

- **Producing an *Executable* Relocatable Output Module (-ar)**

  If you invoke the linker with both the -a and -r options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references, but the relocation information is retained.

  This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

  ```
  lnk430 -ar file1.obj file2.obj -o xr.out
  ```

  Note that you can string the options together (lnk430 -ar) or you can enter them separately (lnk430 -a -r).

- **Relocating or Relinking an Absolute Output Module**

  The linker issues a warning message (but continues executing) when it encounters a file that contains no relocation or symbol table information. Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

8.3.2 Define an Entry Point (-e *global symbol* Option)

The memory address that a program begins executing from is called the **entry point.** When a loader loads a program into target memory, the program counter must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four possible values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table. Possible entry point values include:

- The value specified by the -e option. The syntax is **-e** *global symbol* where *global symbol* defines the entry point and must appear as an external symbol in one of the input files.
- Zero (default value).

This example links file1.obj and file2.obj. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
lnk430 -e begin file1.obj file2.obj
```

8.3.3 Set Default Fill Value (-f *cc* Option)

The -f option fills the holes formed within output sections or initializes uninitialized sections when they are combined with initialized sections. This allows you to initialize memory areas during link time without reassembling a source file. The argument *cc* is a 16–bit constant (up to four hexadecimal digits). If you do not use -f, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCD:

```
lnk430 -f 0ABCDh file1.obj file2.obj
```

8.3.4 Make All Global Symbols Static (-h Option)

The -h option makes output global symbols static. This is useful when you are using partial linking to link related object files into self–contained modules, then relinking the modules together into a final system. If there are global symbols in one module that have the same name as global symbols in other modules, but you want to treat them as separate symbols, use the -h option when building the modules. The global symbols in the modules, which would normally be visible to the other modules and cause possible redefinition problems in the final link, are made static so they are not visible to the other modules.

For example, assume b1.obj, b2.obj, and b3.obj are related and reference a global variable GLOB. Also assume that d1.obj, d2.obj, and d3.obj are related and reference a separate global variable GLOB. You can link the related files together with the following commands:

```
lnk430 -h -r b1.obj b2.obj b3.obj -o bpart.out
lnk430 -h -r d1.obj d2.obj d3.obj -o dpart.out
```

The -h option guarantees that bpart.out and dpart.out will not have global symbols and therefore two distinct versions of GLOB exist. The -r option is used to allow bpart.out and dpart.out to retain their relocation entries. These two partially linked files can then be linked together safely with the following command:

```
lnk430 bpart.out dpart.out -o system.out
```

8.3.5 Alter the Library Search Algorithm (-i *dir* Option/C_DIR)

Usually, when you want to specify a library as linker input, you simply enter the library name as you would any other input filename; the linker looks for the library in the current directory. For example, suppose the current directory contains the library object.lib. Assume that this library defines symbols that are referenced in the file file1.obj. This is how you link the files:

```
lnk430 file1.obj object.lib
```

If you want to use a library that is not in the current directory, use the -l (lowercase “L") linker option. The syntax for this option is **-l** *filename.* The *filename* is the name of an archive library; the space between -l and the filename is optional.

You can augment the linker's directory search algorithm by using the -i linker option or the environment variable. The linker searches for object libraries in the following order:

1) It searches directories named with the -i linker option.
2) It searches directories named with the environment variable C_DIR.
3) If C_DIR is not set, it searches directories named with the assembler's environment variable, A_DIR.
4) It searches the current directory.

-i Linker Options

The -i linker option names an alternate directory that contains object libraries. The syntax for this option is **-i** *dir. dir* names a directory that contains object libraries; the space between -i and the directory name is optional. When the linker is searching for object libraries named with the -l option, it searches through directories named with -i first. Each -i option specifies only one directory, but you can use several -i options per invocation. When you use the -i option to name an alternate directory, it must precede the -l option on the command line or in a command file.

As an example, assume that there are two archive libraries called r.lib and lib2.lib. The table below shows the directories that r.lib and lib2.lib reside in, how to set environment variable, and how to use both libraries during a link.

	Pathname	Invocation Command
DOS	\ld and \ld2	lnk430 f1.obj f2.obj -i\ld -i\ld2 -lr.lib -llib2.lib

Environment Variable (C_DIR)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named **C_DIR** to name alternate directories that contain object libraries. The command for assigning the environment variable is:

DOS `set C_DIR=`pathname;another pathname ...

The *pathnames* are directories that contain object libraries. Use the -l option on the command line or in a command file to tell the linker which libraries to search for.

As an example, assume that two archive libraries called r.lib and lib2.lib reside in ld and ld2 directories. The table below shows the directories that r.lib and lib2.lib reside in, how to set the environment variable, and how to use both libraries during a link.

	Pathname	Invocation Command
DOS	`\ld and \ld2`	`set C_DIR=\ld;\ld2` `lnk430 f1.obj f2.obj -l r.lib -l lib2.lib`

Note that the environment variable remains set until you reboot the system or reset the variable by entering:

DOS `set C_DIR=`

The assembler uses an environment variable named **A_DIR** to name alternate directories that contain copy/include files or macro libraries. If C_DIR is not set, the linker will search for object libraries in the directories named with A_DIR.

8.3.6 Create a Map File (-m *filename* Option)

The -m option creates a link map listing and puts it in *filename.* This map describes:

- Memory configuration.
- Input and output section allocation.
- The addresses of external symbols after they have been relocated.

The map file contains the name of the output module and the entry point; it may also contain up to three tables:

- A table showing the new memory configuration **if** any nondefault memory is specified.
- A table showing the linked addresses of each output section and the input sections that make up the output sections.
- A table showing each external symbol and its address. This table has two columns: the left column contains the symbols sorted by name, and the right column contains the symbols sorted by address.

This example links file1.obj and file2.obj and creates a map file called map.out:

```
lnk430 file1.obj file2.obj -m map.out
```

8.3.7 Name an Output Module (-o *filename* Option)

The linker creates an output module when no errors are encountered. If you do not specify a filename for the output module, the linker gives it the default name a.out. If you want to write the output module to a different file, use the -o option. The *filename* is the new output module name.

This example links file1.obj and file2.obj and creates an output module named run.out:

```
lnk430 -o run.out file1.obj file2.obj
```

8.3.8 Specify a Quiet Run (-q Option)

The -q option suppresses the linker's banner when -q is the first option on the command line or in a command file. This option is useful for batch operation.

8.3.9 Strip Symbolic Information (-s Option)

The -s option creates a smaller output module by omitting symbol table information and line number entries. The -s option is useful for production applications when you must create the smallest possible output module.

This example links file1.obj and file2.obj and creates an output module, stripped off line numbers and symbol table information, named nosym.out:

```
lnk430 -o nosym.out -s file1.obj file2.obj
```

Note that using the -s option limits later use of a symbolic debugger and may prevent a file from being relinked.

8.3.5 Introduce an Unresolved Symbol (-u *symbol* Option)

The -u option introduces an unresolved symbol into the linker's symbol table. This forces the linker to search through a library and include the member that defines the symbol. Note that the linker must encounter the -u option *before* it links in the member that defines the symbol.

For example, suppose a library named rts.lib contains a member that defines the symbol symtab; none of the object files you are linking reference symtab. Suppose you plan to relink the output module, however, and you would like to include the library member that defines symtab in this link. Using the -u option as shown below forces the linker to search rts.lib for the member that defines symtab and to link in the member.

```
lnk430 -u symtab file1.obj file2.obj rts.lib
```

If you do not use -u, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

8.3.6 Exhaustively Read Libraries (-x option)

The linker normally reads input files, archive libraries included, only once: when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library (this is called a *back reference*), the reference will not be resolved.

You can force the linker to repeatedly reread all libraries with the -x option. The linker will continue to reread libraries until no more references can be resolved. For example, if a.lib contains a reference to a symbol defined in b.lib, and b.lib contains a reference to a symbol defined in a.lib, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
lnk430 -la.lib -lb.lib -la.lib
```

or you can force the linker to do it for you:

```
lnk430 -x -la.lib -lb.lib
```

Linking with the -x option may be slower, so you should use the option only as needed.

8.4 Command Files

Linker command files allow you to put linking information in a file; this is useful when you often invoke the linker with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line. Command files are ASCII files that contain one or more of the following:

- Input file names, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line.
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration. The SECTIONS directive controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols.

To invoke the linker with a command file, enter the **lnk430** command and follow it with the name of the command file:

lnk430 *command file name*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it.

The example shows a sample linker command file called link.cmd.

```
/************************************************************/
/*                  Sample Linker Command File              */
/************************************************************/
a.obj                    /*  First input filename           */
b.obj                    /*  Second input filename          */
-o prog.out              /*  Option to specify output file  */
-m prog.map              /*  Option to specify map file     */
```

Example 8.1: Linker Command File

This sample file contains only filenames and options. (Note that you can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
lnk430 link.cmd
```

You can place other parameters on the command line when you use a command file:

```
lnk430 -r link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters it, so a.obj and b.obj are linked into the output module before c.obj and d.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
lnk430 names.lst dir.cmd
```

One command file can call another command file; this type of nesting is limited to 16 levels. If a command file calls another command file as input, this statement must be the *last* statement in the calling command file.

Blanks and blank lines that appear in a command file are insignificant except as delimiters. This applies to the format of linker directives in a command file, also. The following example shows a sample command file that contains linker directives. (Linker directive formats are discussed in later sections.)

```
/************************************************************/
/*        Sample Linker Command File with Directives        */
/************************************************************/
a.obj b.obj c.obj                    /* Input filenames       */
-o prog.out -m prog.map              /* Options               */

MEMORY                               /* MEMORY directives     */
{
  RAM:   origin = 200h       length = 0100h
  ROM:   origin = 0F000h     length = 1000h
}

SECTIONS                             /* SECTION directives    */
{
  .text:   > ROM
  .data:   > ROM
  .bss:    > RAM
}
```

Example 8.2: Command File With Linker Directives

The following names are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

align	GROUP	origin
ALIGN	l (lowercase L)	ORIGIN
attr	len	page
ATTR	length	PAGE
block	LENGTH	range
BLOCK	load	run
COPY	LOAD	RUN
DSECT	MEMORY	SECTIONS
f	NOLOAD	spare
FILL	o	type
fill	org	TYPE
group		UNION

Constants in Command Files

Constants can be specified with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assember or the scheme used for integer constants in "C" syntax.

8.5 Object Libraries

An object library is a partitioned archive file that contains complete object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the MSP430 archiver to build and maintain libraries.

Using object libraries can reduce link time and can reduce the size of the executable module. If a normal object file that contains a function is specified at link time, it is linked whether it is used or not; however, if that same function is placed in an archive library, it is included only if it is referenced.

The order in which libraries are specified is important because the linker includes only those members that resolve symbols that are undefined when the library is searched. The same library can be specified as often as necessary; it is searched each time it is included, or the -x option may be used. A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

This example links several files and libraries. Assume the following:

- Input files f1.obj and f2.obj both reference an external function named clrscr.
- Input file f1.obj references the symbol origin.
- Input file f2.obj references the symbol fillclr.
- Library libc.lib, member 0, contains a definition of origin.
- Library liba.lib, member 3, contains a definition of fillclr.
- Member 1 of both libraries defines clrscr.

If you enter lnk430 f1.obj liba.lib f2.obj libc.lib:

- Member 1 of liba.lib satisfies both references to clrscr because the library is searched and clrscr is defined before f2.obj references it.
- Member 0 of libc.lib satisfies the reference to origin.
- Member 3 of liba.lib satisfies the reference to fillclr.

If, however, you enter lnk430 f1.obj f2.obj libc.lib liba.lib, the references to clrscr are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the -u option to force the linker to include a library member. The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
lnk430 -u rout1 libc.lib
```

If any members of libc.lib define rout1, the linker includes those members. Note that it is not possible to control the allocation of individual library members; members are allocated according to the SECTIONS directive default allocation algorithm.

8.6 The MEMORY Directive

The linker determines where output sections should be allocated into memory; the linker must have a model of target memory to accomplish this task. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses the model to determine which memory locations can be used for object code.

The memory configurations of MSP430 systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use the MEMORY directive to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

8.6.1 Default Memory Model

The linker's default memory model is based on the MSP430 architecture. This model assumes that the following memory is available:

- 256 bytes of RAM, beginning at location 200h
- 4K bytes of ROM, beginning at location 0F000h.

If you do not use the MEMORY directive, the linker uses this default memory model.

8.6.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range of memory has several characteristics:

- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

When you use the MEMORY directive, be sure to identify all the memory ranges that are available to load code into. Any memory that you do not explicitly account for with the MEMORY directive is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. For example, you could use the MEMORY directive to specify a memory configuration as follows:

```
/*******************************************************/
/* Sample command file with MEMORY directive           */
/*******************************************************/
file1.obj     file2.obj    /* Input files */
–o prog.out             /* Options  */

MEMORY
{
     RAM:      origin =  200h     length =  100h
     ROM:      origin =  0F000h   length = 1000h
}
```

You could then use the SECTIONS directive to link the .bss section into the memory area named RAM, .text into ROM, and .data into ROM.

The general syntax for the MEMORY directive is:

MEMORY
{
name 1 [(attr)] **: origin =** constant , **length =** constant, **fill =** constant
.
.
name n [(attr)] **: origin =** constant , **length =** constant, **fill =** constant
}

name Names a memory range. A memory name may be 1 to 8 characters; valid characters include A-Z, a-z, $, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not retained in the output file or in the symbol table.

attr Specifies 1 to 4 attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes can restrict the allocation of output sections into certain memory ranges. If you do not use any attributes, you can allocate any output section into any range with no restrictions. Any memory for which no attributes are specified (including all memory in the default model) has all four attributes. Valid attributes include:

- **R** Specifies that the memory can be read.
- **W** Specifies that the memory can be written to.
- **X** Specifies that the memory can contain executable code.
- **I** Specifies that the memory can be initialized.

origin Specifies the starting address of a memory range and may be abbreviated as *org* or *o*. The value, specified in bytes, is a long integer constant and may be decimal, octal, or hexadecimal.

length Specifies the length of a memory range and may be abbreviated as *len* or *l*. The value, specified in bytes, is a long integer constant and may be decimal, octal, or hexadecimal.

fill Specifies a fill character for the memory range and may be abbreviated as *f*. Fills are optional. The value is a two–byte integer constant and may be decimal, octal, or hexadecimal. The fill value will be used to fill any areas of the memory range that are not allocated to a section.

> **Note: Filling Memory Ranges**
>
> If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with zeros) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFh:

```
MEMORY
{
       RFILE (RW) : o = 02h, l = 0FEh, f = 0FFFFh
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use the MEMORY directive to specify the target system's memory model, you can use the SECTIONS directive to allocate output sections into specific named memory ranges or into memory that has specific attributes.

8.7 The SECTIONS Directive

The SECTIONS directive tells the linker how to combine sections from input files into sections in the output module and where to place the output sections in memory. In summary, the SECTIONS directive:

- Describes how input sections are combined into output sections.
- Defines output sections in the executable program.
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space).
- Permits renaming of output sections.

8.7.1 Default Sections Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections.

8.7.2 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

SECTIONS
{
name : [*property*, *property*, *property*, ...]
name : [*property*, *property*, *property*, ...]
name : [*property*, *property*, *property*, ...]
}

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) After the section *name* is a list of properties that define the section's contents and how it is allocated. The properties may be separated by optional commas. Possible properties for a section are:

load allocation defines where in memory the section is to be loaded.
Syntax: **load =** *allocation* **or**
allocation **or**
> *allocation*

run allocation defines where in memory the section is to be run.
Syntax: **run =** *allocation* **or**
run > *allocation*

input sections defines the input sections composing the section.
Syntax: { input_sections }

section type defines flags for special section types.
Syntax: **type = COPY** **or**
type = DSECT **or**
type = NOLOAD
For more information on section types, see Section 8.12.

fill value defines the value used to fill uninitialized "holes"
Syntax: **fill =** *value* **or**
name: ... { ... } **=** *value*
For more information on creating and filling holes, see Section 8.14.

The example shows a SECTIONS directive in a sample linker command file. The figure on the next page shows how these sections are allocated in memory.

```
/************************************************************/
/* Sample command file with SECTIONS directives             */
/************************************************************/
file1.obj file2.obj                    /* Input files       */
-o prog.out                            /* Options           */

SECTIONS
{
  .text:    load = ROM
  .const:   load = ROM, run = 0D000h
  .bss:     load = RAM
  .vectors: load = 0FFE0h
    {
      t1.obj(.intvec1)
      t2.obj(.intvec2)
      endvec = .;
    }
  .data:     align = 16
}
```

Example 8.3: The SECTIONS Directive

The figure shows the five output sections defined by the sections directive in the last example: .vectors, .text, .const, .bss, and .data.

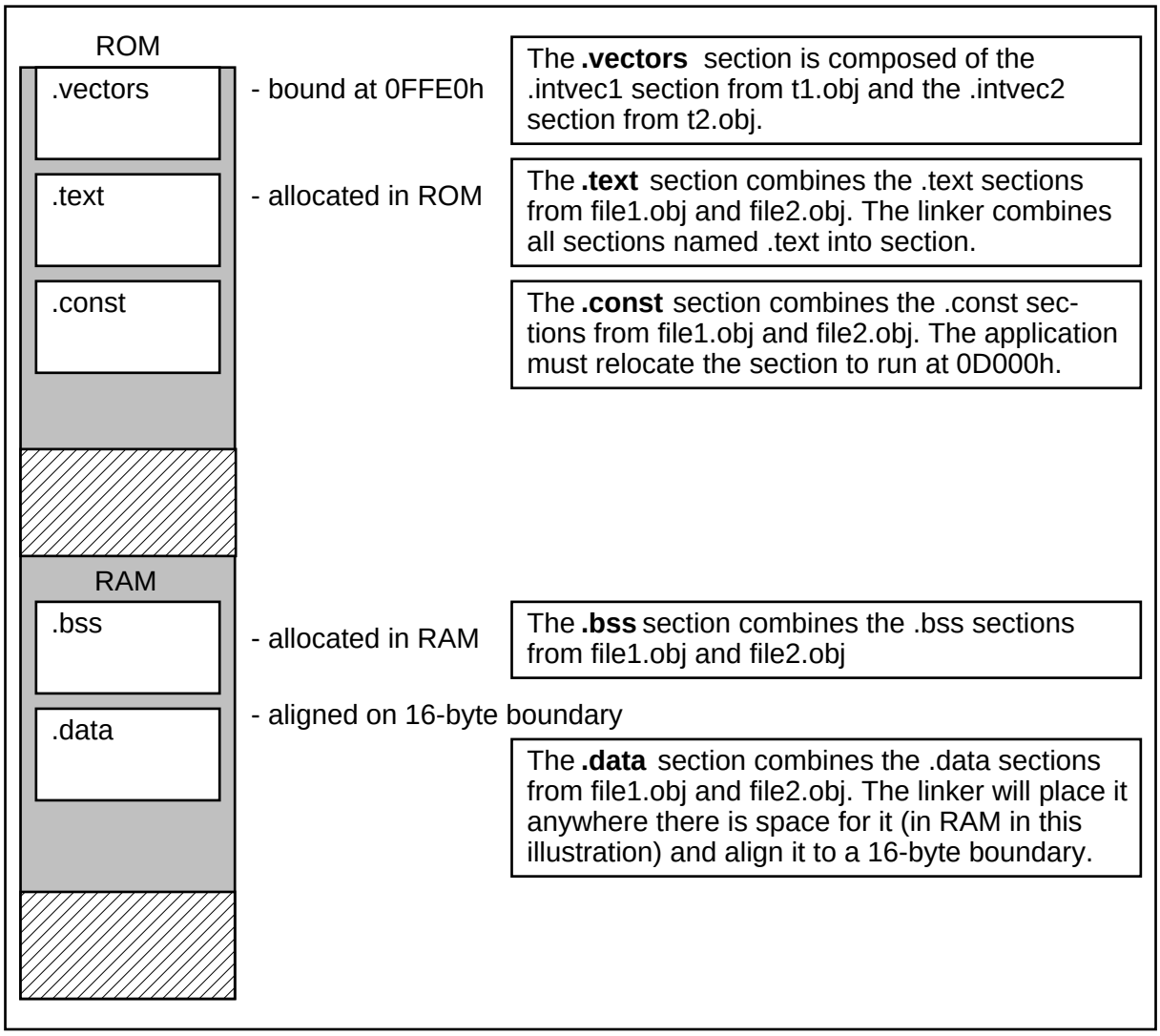

Figure 8.2: Section Allocation

8.7.3 Specifying the Address of Output Sections (Allocation)

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. In any case, the process of locating the output section in the target's memory and assigning its address(es) is called allocation.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equals sign or greater–than sign, and a value optionally enclosed in parentheses. If load and run allocation is separate, all parameters following the keyword LOAD apply to load allocation, and those following RUN apply to run allocation. Possible allocation parameters are:

binding allocates a section at a specific address
```
.text: load = 0x1000
```

memory allocates the section into a range defined in the MEMORY directive with the specified name or attributes
```
.text: load > ROM
```

alignment specifies that the section should start on an address boundary
```
.text: align = 0x100
```

blocking specifies that the section must fit between two address boundaries: for example, on a single data page.
```
.text: block(0x100)
```

For the load (usually the only) allocation, you may simply use a greater–than sign and omit the LOAD keyword:

```
.text: > ROM            .text: {...} > ROM
.text: > 0x4000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > ROM align 16
```

Or if you prefer, use parentheses for readability:

```
.text: load = (ROM align(16))
```

Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x4000
```

This example specifies that the .text section must begin at location 4000h. The binding address must be a 16–bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Note: Binding and Alignment or Named Memory Are Incompatible

You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

Memory

You can allocate a section into a memory range that is defined by the MEMORY directive. This example names ranges and links sections into them:

```
MEMORY
{
     ROM (RIX) :      origin = 0h,      length = 1000h
     RAM (RWIX):      origin = 0D000h,  length = 1000h
}
SECTIONS
{
     .text  :   > ROM
     .data  :   > RAM,         ALIGN=64
     .bss   :   > RAM
}
```

In this example, the linker places .text into the area called ROM. The .data and .bss output sections are allocated into RAM. You can align a section within a named memory range; the .data section is aligned on a 64–word boundary within the RAM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
     .text:  > (X)   /* .text --> executable memory      */
     .data:  > (RI)  /* .data --> read or init memory    */
     .bss :  > (RW)  /* .bss  --> read or write memory   */
}
```

In this example, the .text output section can be linked into either the ROM or RAM area because both areas have the X attribute. The .data section can also go into either ROM or RAM because both areas have the R and I attributes. The .bss output section, however, must go into the RAM area because only RAM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no other sections had been bound to addresses that would interfere with this allocation process, the .text section would start at address 0. If a section must start on a specific address, use binding instead of named memory.

Alignment and Blocking

You can tell the linker to place an output section at an address that falls on an n–byte boundary, where n is a power of 2. For example:

```
.text: load = align(32)
```

allocates .text so that it falls on a 32-byte boundary.

Blocking is a weaker form of alignment that places a section so that it is allocated anywhere **within** a “block" of size *n*. As with alignment, *n* must be a power of 2. For example:

```
bss: load = block(0x1000)
```

allocates .bss so that the entire section is contained in a single 4K–byte data page.

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

8.7.4 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. The linker combines input sections by concatenating them in the order specified. The size of an output section is the sum of the sizes of the input sections that make up the output section.

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

Example 8.4: The Most Common Method of Specifying Section Contents

The linker takes all the .text sections from the input files and combines them into the .text output section. The linker concatenates the .text input sections in the order that it encounters them in the input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for *any* output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
     .text :               /* Build .text output section               */
     {
       f1.obj(.text)       /* Link .text section from f1.obj           */
       f2.obj(sec1)        /* Link sec1 section from f2.obj            */
       f3.obj              /* Link ALL sections from f3.obj            */
       f4.obj(.text,sec2)  /* Link .text and sec2 from f4.obj          */
     }
}
```

Note that it is not necessary for an input section to have the same name as another it is combined with or as the output section it becomes part of. If a file is listed with no sections, **all** of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example and these .text sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after f4.obj(sec2).

The specifications in the example on the page before are actually a shorthand method for the following:

```
SECTIONS
{
     .text: { *(.text) }
     .data: { *(.data) }
     .bss:  { *(.bss)  }
}
```

The *(.text) means *the unallocated .text sections from all the input files.* This format is useful when:

- You want the output section to contain all input sections that have a certain name, but the output section name is different from the input sections' names.
- You want the linker to allocate the input sections *before* it processes additional input sections or commands within the braces.

Here's an example that uses this method:

```
SECTIONS
{
     .text   :    {
                                abc.obj(xqt)
                                      *(.text)
                  }
     .data   :    {
                                *(.data)
                                fil.obj(table)
                  }
}
```

In this example, the .text output section contains a named section **xqt** from file *abc.obj*, which is followed by *all* the .text input sections. The .data section contains *all* the .data input sections, followed by a named section *table* from the file *fil.obj*. Note that this method includes all the *unallocated* sections. For example, if one of the .text input sections was already included in another output section when the linker encountered *(.text), the linker could not include that first .text input section in the second output section.

8.8 Specifying a Section's Runtime Address

It may be necessary or desirable at times to load code into one area of memory and run it in another. For example, you may have performance–critical code in a ROM–based system. The code must be loaded into ROM but would run much faster if it were in RAM.

The linker provides a simple way to specify this. In the SECTIONS directive, you can optionally direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = ROM, run = RAM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

8.8.1 Specifying Two Addresses

The load address determines where a loader will place the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does **not** happen automatically just by specifying a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and will load and run at the same address. If you provide both allocations, the section is actually allocated as if it were two different sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. After the keyword *load,* everything having to do with allocation affects the load address until the keyword *run* is seen, after which everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You may also specify run first, then load. Use parentheses to improve readability. Examples:

```
.data: load = ROM, align = 32, run = RAM
```

(align applies only to load)

```
.data: load = (ROM align 32), run = RAM
```

(identical to previous example)

```
.data: run    =    RAM, align 32,
       load   =    align 16
```

(align 32 in RAM for run; align 16 anywhere for load)

8.8.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so the only address of significance is the run address. The linker allocates uninitialized sections only once. If you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run. Examples:

```
.bss: load = 0x1000, run = RAM
```

A warning is issued, load is ignored, space is allocated in RAM. All of the following examples have the same effect. The .bss section is allocated in RAM.

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

8.8.3 Referring to the Load Address by Using the .label Directive

Any reference to a normal symbol in a section refers to its runtime address. However, it may be necessary at runtime to refer to a load–time address. In particular, the code that copies a section from its load address to its run address must know where it was loaded. The .label directive in the assembler defines a special type of symbol that refers to the load address of the section. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address.

```
;-------------------------------------------------------------
;   define a section to be copied from ROM to RAM
;-------------------------------------------------------------
                .sect ".fir"
                .label fir_load      ; load address of section
fir:                                 ; run address of section
;               <code here>          ; code for the section
                ret

                .label fir_end

fir_len .equ fir_end - fir_load

;-------------------------------------------------------------
;  copy .fir section from ROM into RAM
;-------------------------------------------------------------
                .text
                MOV   #fir_len, R4
                MOV   #fir_load, R5
                MOV   #fir, R6

                JMP   L2
L1:             MOV   @R5+, 0(R6)
                INCD  R6
L2:             DECD  R4
                JC    L1


;-------------------------------------------------------------
;  jump to section, now in RAM
;-------------------------------------------------------------
                call fir             ; call runtime address
```

Linker Command File

```
/*************************************************************/
/* PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE               */
/*************************************************************/
MEMORY
{
  ROM:    origin=4000h     length = 4000h
  RAM:    origin=2000h     length = 2000h
}

SECTIONS
{
  .text:  load = ROM
  .fir:   load = ROM, run = RAM
}
```

Example 8.5: Copying a Section From ROM to RAM

The figure illustrates the runtime execution of the last example.

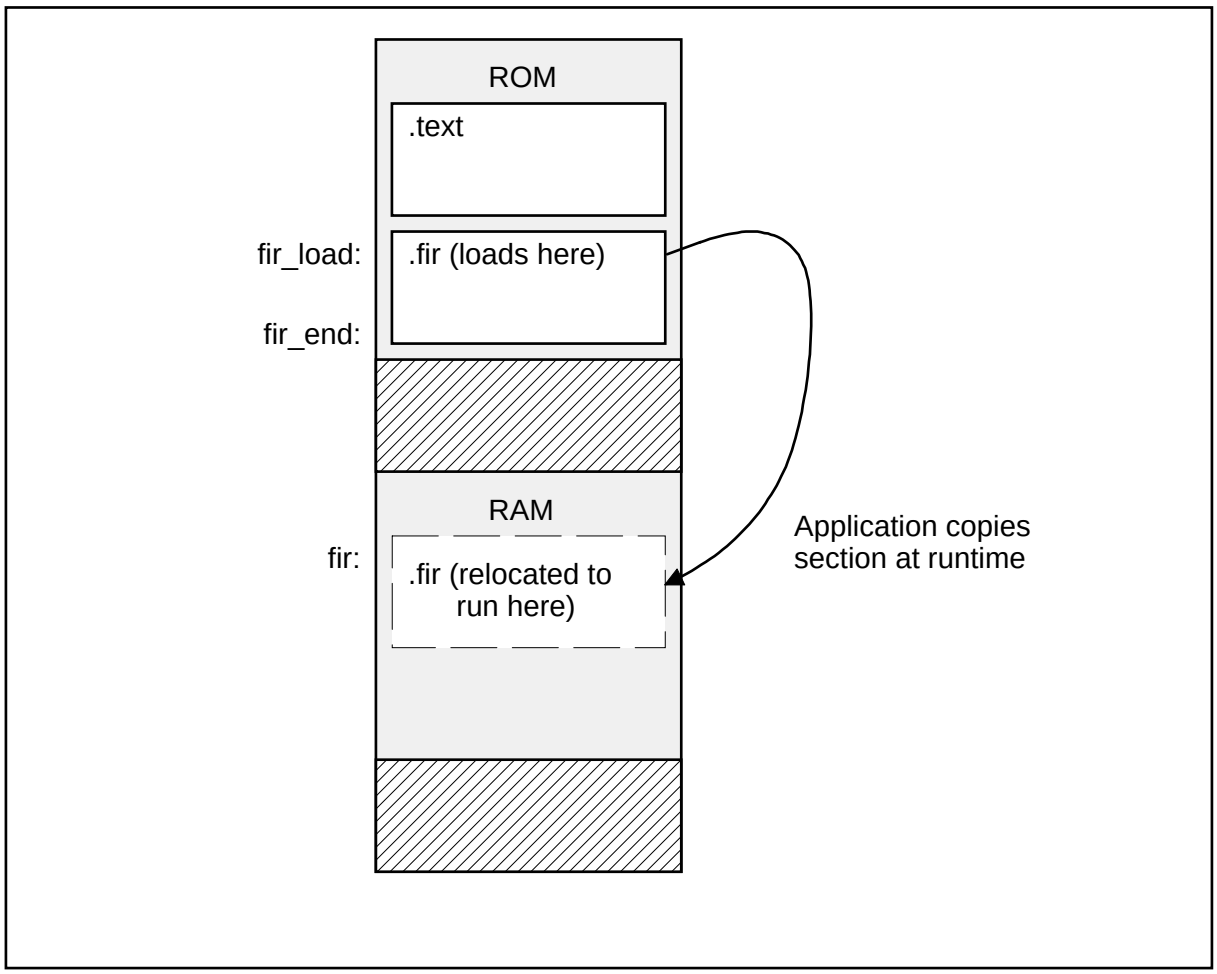

Figure 8.3: Runtime Execution

8.9 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning output sections causes the linker to allocate the same run address to the sections. Grouping output sections causes the linker to allocate them contiguously in memory.

8.9.1 Overlaying Sections With the UNION Statement

For some applications, you may wish to allocate more than one section to run at the same address; for example, you may have several routines you want in on–chip RAM at various stages of the program's execution. Or you may want several data objects that you know will not be active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run address.

```
SECTIONS
{
   .text: load = ROM
   UNION: run  = RAM
   {
              .bss1: { file1.obj(.bss) }
              .bss2: { file2.obj(.bss) }
   }
   .bss3: run = RAM { globals.obj(.bss) }
}
```

Example 8.6: Illustration of the Form of the UNION Statement

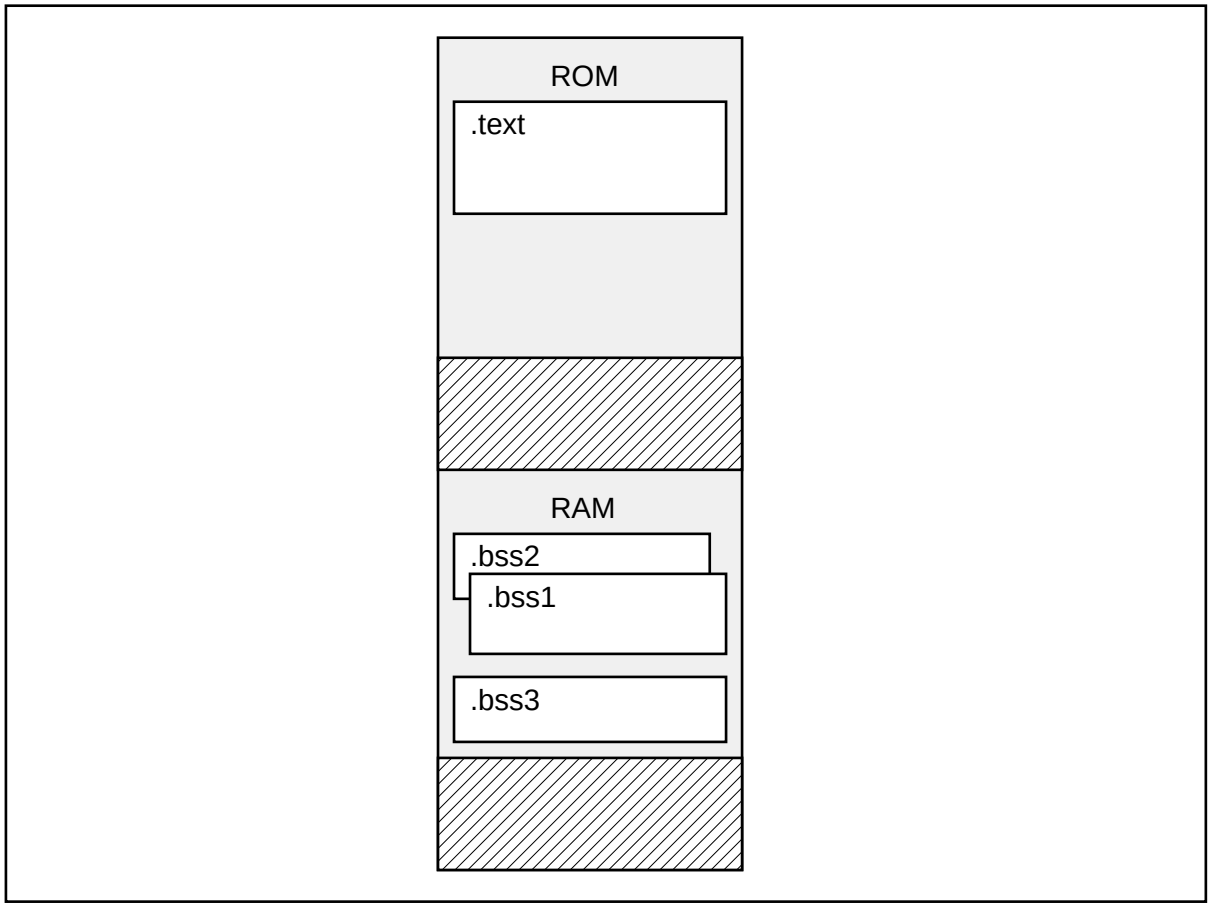

Figure 8.4: Runtime Memory Allocation

In the example on the page before, the .bss sections from file1.obj and file2.obj are allocated *at the same address* in RAM. The union occupies as much space in the memory map as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

Allocation of a section as part of a union affects only its *run address*. **Under no circumstances can sections be overlaid for loading**. If an initialized section is a union member (an initialized section has raw data, such as .text), its load allocation **must** be separately specified. The next example illustrates this.

```
UNION run = RAM
{
   .text1: load = ROM, { file1.obj(.text) }
   .text2: load = ROM, { file2.obj(.text) }
}
```

Example 8.7: Illustration of Separate Load Addresses for UNION Sections

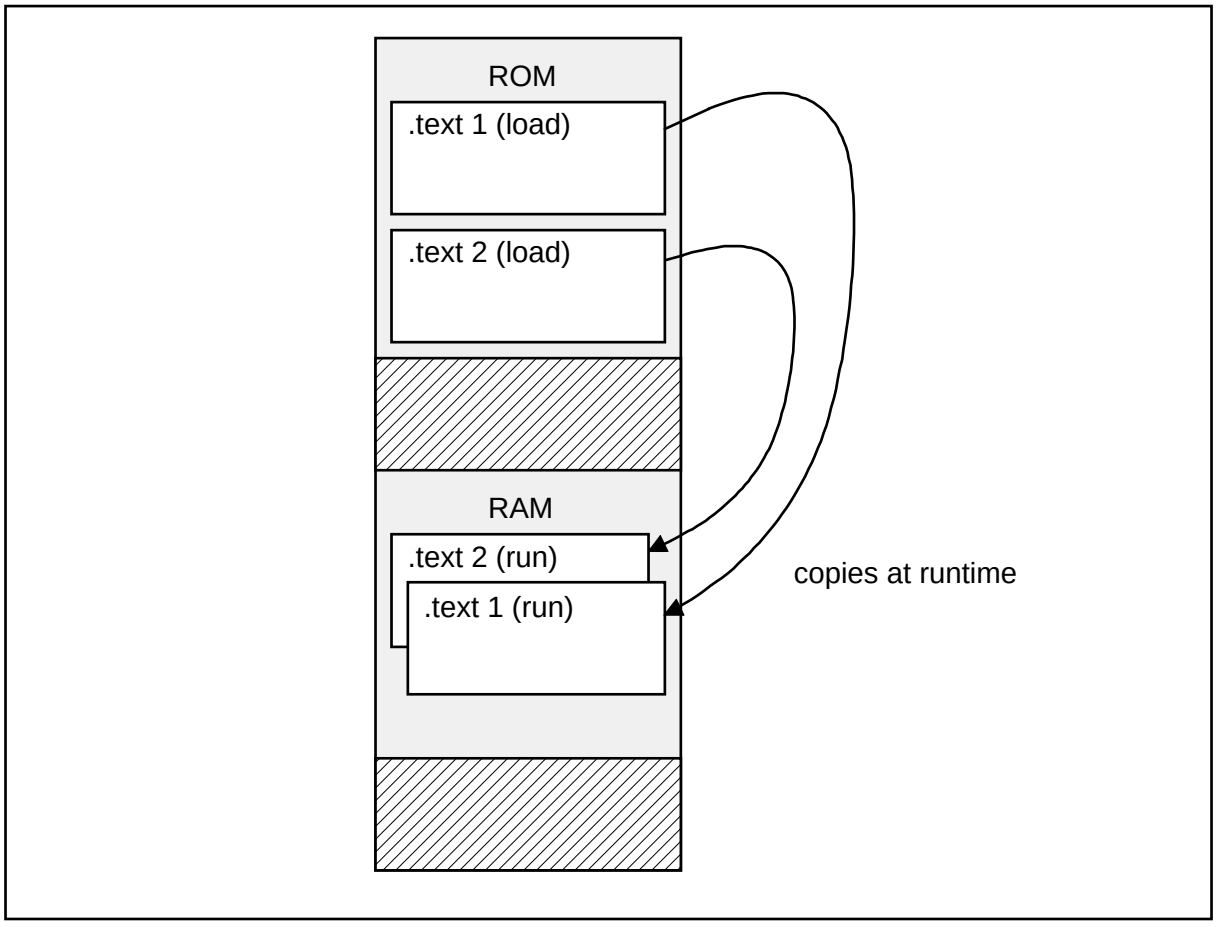

Figure 8.5: Load and Run Memory Allocation

Since the .text sections contain data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it fits in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is redundant to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and, if both are specified, the linker issues a warning and ignores the load address.

8.9.2 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that forces several output sections to be allocated contiguously. For example, assume that a section named *term_rec* contains a termination record for a table in the .data section. You can force the linker to allocate *.data* and *term_rec* together:

```
SECTIONS
{
     .text                /* Normal output section                */
     .bss                 /* Normal output section                */
     GROUP 1000h :        /* Specify a group of sections          */
     {
         .data            /* First section in the group           */
         term_rec         /* Allocated immediately after .data    */
     }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same way a single output section is allocated. In the preceding example, the GROUP is bound to address 1000h. This means that .data is allocated at 1000h, and *term_rec* follows it in memory.

Note: You Cannot Specify Addresses for Sections Within a Group

When you use the GROUP option, binding, alignment, or allocation into named memory can be specified *for the group only*. You cannot use binding, named memory, or alignment for sections *within* a group.

8.11 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. Any memory locations or sections that you choose *not* to specify, however, must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply.

8.11.1 Default Allocation

If you do not use any MEMORY or SECTIONS directives, the linker acts as though the following definitions were specified:

```
MEMORY
{
     RAM    :     origin = 200h   length = 100h
     ROM    :     origin = 0F000h length = 1000h
}
SECTIONS
{
     .bss   :     >   RAM
     .text  :     >   ROM
     .data  :     >   ROM

}
```

All .bss input sections are concatenated to form one .bss output section linked into. All .data input sections are combined to form a .data output section, which is linked into ROM. All .text input sections are concatenated to form a .text output section, which is linked into ROM starting at location 0F000h.

Unless you specify otherwise with a MEMORY directive, the linker assumes the configuration specified above. That is, the only memory that the linker uses to build your program is:

- 256 bytes starting at location 0200h,
- 4K bytes starting at location 0F000h.

If there are additional input sections in the input files (specifically, named sections), the linker links them in after the default sections have been linked. Input sections that have the same name are combined into a single output section with this name. The linker allocates these additional output sections into memory wherever there is room. Usually, it is desirable to use explicit SECTIONS directives to tell the linker where to place named sections.

Note: The SECTIONS Directive

If a SECTIONS directive is specified, the linker performs no part of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described below.

8.11.2 General Rules for Forming Output Sections

An output section can be formed in one of two ways:

Rule 1 As the result of a SECTIONS directive definition.

Rule 2 By combining input sections with the same names into output sections that are not defined in a SECTIONS directive.

If an output section is formed as a result of a SECTIONS directive (rule 1), this definition completely determines the section's contents.

An output section can also be formed when input sections are encountered that are not specified by any SECTIONS directive (rule 2). In this case, the linker combines all such input sections that have the same name into an output section with this name. For example, suppose the files f1.obj and f2.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section to contain them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

After the linker determines the composition of all the output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured, or if there is no MEMORY directive, the linker uses the default configuration.

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. This is the algorithm:

1) Output sections for which you have supplied a specific binding address are placed in memory at that address.
2) Output sections that are included in a specific, named memory range or that have memory attribute restrictions are allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3) Output sections that have zero length are allocated at the beginning of the first appropriate memory area unless they are part of a group.
4) Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

8.12 Special Section Types (DSECT, COPY, and NOLOAD)

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that the program is treated when it is linked and loaded. For example:

```
SECTIONS
{
     sec1: load = 2000h, type =  DSECT     {f1.obj}
     sec2: load = 4000h, type =  COPY      {f2.obj}
     sec3: load = 6000h, type =  NOLOAD    {f3.obj}
}
```

- The DSECT type creates a “dummy section" that has the following qualities:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from f1.obj are allocated, but all the symbols are relocated as though the sections were linked at address 2000h. The other sections can refer to any of the global symbols in sec1.
- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module.
- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for it, it appears in the memory map listing, etc.

8.13 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation–dependent value.

8.13.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

symbol	**=**	*expression;*	assigns the value of expression to symbol
symbol	**+ =**	*expression;*	adds the value of expression to symbol
symbol	**- =**	*expression;*	subtracts the value of expression from symbol
symbol	*** =**	*expression;*	multiplies symbol by expression
symbol	**/ =**	*expression;*	divides symbol by expression

The symbol should be defined externally in the program. If it is not, the linker defines a new symbol and enters it into the symbol table. Assignment statements **must** be terminated with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. cur_tab must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program in order to change tables. Instead, you can use a linker assignment statement to assign cur_tab at link time:

```
prog.obj             /* Input file                          */
cur_tab = Table1;    /* Assign cur_tab to one of the tables */
```

8.13.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the SPC during allocation. The linker's " . " symbol is analogous to the assembler's "$" symbol. The " . " symbol can be used only in assignment statements within a SECTIONS directive because " . " is meaningful only during allocation, and SECTIONS controls the allocation process.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global directive, you can create an external undefined variable called Dstart in the program. Then, assign the value of "." to Dstart:

```
SECTIONS
{
     .text:      {}
     .data:      { Dstart = .; }
     .bss:       {}
}
```

This defines Dstart to be the ultimate linked address of the .data section. (dstart is assigned *before* .data is allocated.) The linker will relocate all references to Dstart.

A special type of assignment assigns a value to the "." symbol. This adjusts the location counter within an output section and creates a hole between two input sections. Any value assigned to "." to create a hole is relative to the beginning of the section, not to the address actually represented by ".".

8.13.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in the next table.
- All numbers are treated as long (32–bit) integers.
- Constants are identified by the linker in the same manner as they are by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address.* No type–checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains **any** relocatable symbols (and zero or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, the symbol is relocatable; if it is assigned the value of an absolute expression, the symbol is absolute.

The linker supports the C language operators listed in the table in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in the table, the linker also has an *align* operator that allows a symbol to be aligned on an *n*–byte boundary within an output section *(n* is a power of 2). For example, the expression

```
. = align(16);
```

aligns the SPC within the current section on the next 16–byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as "." — that is, within a SECTIONS directive.

Group 1 (Highest Precedence)		Group 6	
!	Logical not	&	Bitwise AND
~	Bitwise not		
-	Negative		
Group 2		**Group 7**	
*	Multiplication	|	Bitwise OR
/	Division		
%	Mod		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
-	Minus		
Group 4		**Group 9**	
>>	Arithmetic right shift	||	Logical OR
<<	Arithmetic left shift		
Group 5		**Group 10 (Lowest Precedence)**	
==	Equal to	=	Assignment
! =	Not equal to	+ =	A + = B ® A = A + B
>	Greater than	- =	A - = B ® A = A - B
<	Less than	* =	A * = B ® A = A * B
< =	Less than or equal to	/ =	A / = B ® A = A / B
> =	Greater than or equal to		

Table 8.2: Operators in Assignment Expressions

8.13.4 Symbols Defined by the Linker

The linker automatically defines several symbols that a program can use at runtime to determine where a section is linked. Since these symbols are external, they appear in the link map. Each symbol can be accessed in any assembly language module if it is declared with a .global directive. Values are assigned to these symbols as follows:

.text is assigned the first address of the .text output section.
(It marks the *beginning* of executable code.)

etext is assigned the first address following the .text output section.
(It marks the *end* of executable code.)

.data is assigned the first address of the .data output section.
(It marks the *beginning* of initialized data tables.)

edata is assigned the first address following the .data output section.
(It marks the *end* of initialized data tables.)

.bss is assigned the first address of the .bss output section.
(It marks the *beginning* of uninitialized data.)

end is assigned the first address following the .bss output section.
(It marks the *end* of uninitialized data.)

8.14 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called **holes**. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles such holes and how you can fill holes (and uninitialized sections) with values.

8.14.1 Initialized and Uninitialized Sections

There are two guidelines to remember about the contents of an output section. An output section contains either:

- Raw data for the *entire* section, **or**
- *No* raw data.

A section that has raw data is said to be **initialized.** This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections **always** have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also have raw data.

By default, the .bss section and sections defined with the .usect directive have no raw data (they are **uninitialized**). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in RAM for variables. In the object file, an uninitialized section has a normal section header and may have symbols defined in it; no memory image, however, is stored in the section.

8.14.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must follow the first guideline (above) and supply raw data for the hole.*

Holes can be created only *within* output sections. There can also be space *between* output sections, but such spaces are not holes.

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by " . ") by adding to it, assigning a greater value to it, or aligning it on an address boundary.

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
     outsect:
     {
          file1.obj(.text)
          . += 100h;      /* Create a hole with size 100h   */
          file2.obj(.text)
          . = align(16);  /* Create a hole to align the SPC */
          file3.obj(.text)
     }
}
```

The output section outsect is built as follows:

- The .text section from file1.obj is linked in.
- The linker creates a 256–byte hole.
- The .text section from file2.obj is linked in after the hole.
- The linker creates another hole by aligning the SPC on a 16–byte boundary.
- Finally, the .text section from file3.obj is linked in.

All values assigned to the " . " symbol within a section refer to the *relative address within the section.* The linker handles assignments to the " . " symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement . = align(16) in the example. This statement effectively aligns file3.obj .text to start on a 16–word boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, file3 .text will not be aligned, either.

Expressions that decrement " . " are illegal. For example, it is invalid to use the -= operator in an assignment to " . ". The most common operators used in assignments to " . " are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section. For example:

```
.text:    {     .+= 100h; }       /* Hole at the beginning   */
.data:    {
                *(.data)
                . += 100h;   }    /* Hole at the end         */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* Here is an example of creating a hole in this way:

```
SECTIONS
{
     outsect:
     {
          file1.obj(.text)
          file1.obj(.bss)         /* This becomes a hole */
          }
}
```

Because the .text section has raw data, all of outsect must also contain raw data (first guideline). Therefore, the uninitialized .bss section becomes a hole.

Note that uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

8.14.3 Filling Holes

Whenever there is a hole in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 16–bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1) If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 16–bit constant. For example:

```
SECTIONS
{
          outsect:
     {
          file1.obj(.text)
          file2.obj(.bss) = 0FFh /* Fill this hole */
     }                           /* with 00FFh     */
}
```

2) You can also specify a fill value for all the holes in an output section by using the fill keyword. For example:

```
SECTIONS
{
     outsect: fill = 0FF00h      /* This fills holes        */
                                 /* with 0FF00h                */
     {
          . += 10h;              /* This creates a hole        */
          file1.obj(.text)
          file1.obj(.bss)        /* This creates another hole  */
     }
}
```

3) If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with -f. Suppose the command file link.cmd contains the following SECTIONS directive. For example:

```
SECTIONS
{
      .text: { .= 100; }    /* Create a 100–word hole */
}
```

Now invoke the linker with the -f option:

```
lnk430 -f 0FFFFh link.cmd
```

This fills the hole with 0FFFFh.

4) If you do not invoke the linker with the -f option, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

8.14.4 Explicit Initialization of Uninitialized Sections

An uninitialized section becomes a hole only when it is combined with an initialized section. When uninitialized sections are combined with each other, the resulting output section remains uninitialized and has no raw data in the output file.

However, you can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
     .bss: fill = 1234h       /* Fills .bss with 1234h */
}
```

Note: Filling Sections

Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

8.14.5 Examples of Using Initialized Holes

The MSP430X201 device has 4K bytes of program memory, starting at location 0F000h. The top bytes of this area are reserved for interrupt vectors. Suppose you want to link the .text sections from three object files into a .text output section that begins at address 0F000h. Suppose also that you have a section of initialized interrupt vectors called int_vecs that you want to link at address 0FFE0h. You could fill the space between the end of the .text section and the beginning of the interrupt vectors; the figure shows the space filled with a 1–byte fill value of 0EFh and illustrates the desired memory map for program memory.

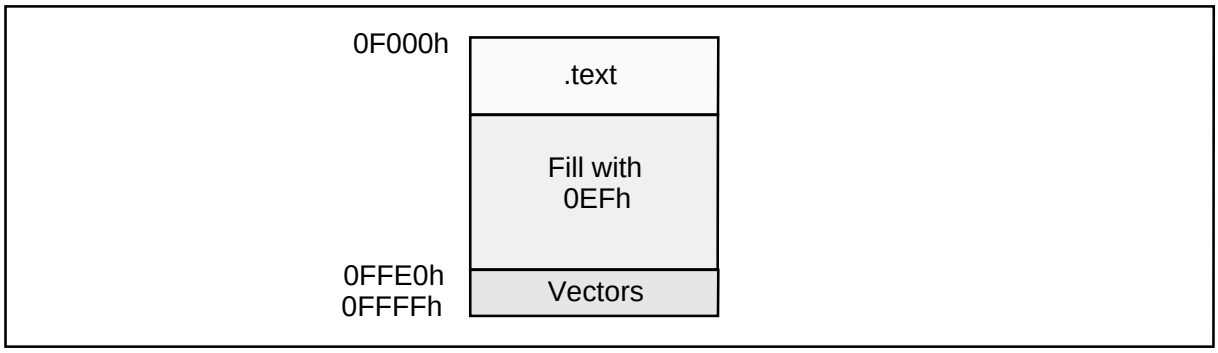

Figure 8.6: Initialized Hole

To obtain the configuration shown in the figure, you must create one large output section that has .text at the beginning, int_vecs at the end, and a hole between filled with 0EFh:

```
SECTIONS
{
     prog 0F000h :fill = OEFEFh   /* Define prog and start at 0F000h and */
          {                       /* Specify a fill value                */
          file1.obj(.text)        /* Link .text sections from each file  */
          file2.obj(.text)
          file3.obj(.text)
          . = 0FE0h;              /* Create hole to 0FE0h (0FFE0h abs)   */
          file1.obj(int_vecs)     /* Link in the vectors section         */
     }
}
```

The fill value must be a 16–bit constant. To have the value 0EFh in each byte, the fill value was specified as 0EFEFh.

Notice that the value 0FE0h, which is assigned to the section program counter (.), is relative to the beginning of the section. Because the section begins at 0F000h, the hole is actually created from the end of the .text section to address 0FFE0h.

8.15 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as **partial linking,** or incremental linking. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- Intermediate files **must** have relocation information. Use the -r option when you link the file the first time.
- Intermediate files **must** have symbolic information. By default, the linker retains symbolic information in its output. Do not use the -s option if you plan to relink a file, because -s strips symbolic information from the output module.
- Intermediate link steps should be concerned only with the formation of output sections and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link step.

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the -r option to retain relocation information in the output file tempout1.out.

```
lnk430 -r -o tempout1 file1.com
```

file1.com contains:

```
SECTIONS
{
     ss1:   {
            f1.obj
            f2.obj
              .
              .
              .
            fn.obj
            }
}
```

Step 2: Link the file file2.com; use the -r option to retain relocation information in the output file tempout2.out.

```
lnk430 -r -o tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
     ss2:   {
            g1.obj
            g2.obj
             .
             .
             .
            gn.obj
            }
}
```

Step 3: Link tempout1.out and tempout2.out:

```
lnk430 -m final.map -o final.out tempout1.out tempout2.out
```

8.17 Linker Example

This example links a program called demo.out. There are three object modules, demo.obj, ctrl.obj, and tables.obj.

Assume the following memory configuration:

Address Range	Memory Contents
200h to 2FFh	internal RAM
1F00h to 1FFFh	Data EEPROM
2000h to 3FF**Fh**	8K external RAM
0F000h to 0FFFFh	4K internal program ROM

The program is built from the following elements:

- Executable code, contained in the .text sections of demo.obj and ctrl.obj, must be linked into program ROM. The symbol SETUP must be defined as the program entry point.
- A set of interrupt vectors, contained in the int_vecs section of tables.obj, must be linked at address 0FFE0h in program ROM.
- A table of coefficients, contained in the .data section of tables.obj, must be linked into .EEPROM. The remainder of EEPROM must be initialized with the value 0A26Eh.
- A set of variables, contained in the .bss section of ctrl.obj, must be linked into the RAM. These variables must be preinitialized to 0FFFFh.
- Another .bss section in demo.obj must be linked into external RAM.

The next two figures illustrate the linker command file and the map file for this example.

```
/*****************************************************************/
/* Specify the Linker Options                                    */
/*****************************************************************/
-e SETUP               /* Define the entry point                 */
-o demo.out            /* Name the output file                   */
-m demo.map            /* Create a load map                      */
/*****************************************************************/
/* Specify the Input Files                                       */
/*****************************************************************/
demo.obj
ctrl.obj
tables.obj
/*****************************************************************/
/* Specify the Memory Configuration                              */
/*****************************************************************/
MEMORY
{
    RAM         :   origin     =      0200h     length = 0100h
    EEPROM      :   origin     =      1F00h     length = 0100h
    RAMEXT      :   origin     =      2000h     length = 2000h
    ROM         :   origin     =     0F000h     length = 1000h
}
/*****************************************************************/
/* Specify the Output Sections                                   */
SECTIONS
    .text: > ROM            /* Link all .text sections into ROM     */

    int_vecs 0FFE0h: {}     /* Link interrupts at FFE0h             */

    .data:                  /* Link the data sections               */
    {
        tables.obj(.data)
        . = 100h;           /* Create a hole to end of the block    */
    } = 0A26Eh > EEPROM     /* Fill and link into EEPROM            */

    ctrl_vars:              /* Create new section for ctrl vars     */
    {
        ctrl.obj(.bss)
    } = 0FFFFh > RAM        /* Fill with 0FFFFh and link to RAM     */

    .bss   > RAMEXT         /* Link all remaining .bss sections     */
}
/*****************************************************************/
/* End of Linker Command File                                    */
/*****************************************************************/
```

Figure 8.7: Linker Command File, demo.cmd

Now invoke the linker by entering the following command:

lnk430 demo.cmd

This creates the map file shown in the next figure and an output file called demo.out that can be run on the MSP430.

```
*******************************************************************
MSP430 COFF Linker                                      Version 1.00
*******************************************************************
Thu Feb 10 09:21:32 1994
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address: 0000f000

MEMORY CONFIGURATION
     name         origin    length      attributes     fill
     RAM        00000200    000000100   RWIX
     EEPROM     00001f00    000000100   RWIX
     RAMEXT     00002000    000002000   RWIX
     ROM        0000f000    000001000   RWIX

SECTION ALLOCATION MAP

output                                  attributes/
section  page   origin     length       input sections
.text       0 0000f000   00000010
              0000f000   00000008       demo.obj (.text)
              0000f008   00000000       tables.obj (.text)
              0000f008   00000008       ctrl.obj (.text)

int_vecs    0 0000ffe0   00000020
              0000ffe0   00000020       tables.obj (int_vecs)

.data       0 00001f00   00000100
              00001f00   00000008       tables.obj (.data)
              00001f08   000000f8       --HOLE-- [fill = a26e]
              00002000   00000000       ctrl.obj (.data)
              00002000   00000000       demo.obj (.data)

ctrl_var    0 00000200   00000004
              00000200   00000004       ctrl.obj (.bss) [fill = ffff]

.bss        0 00002000   00000004       UNINITIALIZED
              00002000   00000004       demo.obj (.bss)
              00002004   00000000       tables.obj (.bss)

GLOBAL SYMBOLS
address   name                          address     name
00002000  .bss                          00001f00     .data
00001f00  .data                         00002000     edata
0000f000  .text                         00002000     .bss
0000f000  SETUP                         00002004     end
00002000  edata                         0000f000    .text
00002004  end                           0000f000    SETUP
0000f010  etext                         0000f010     etext

[7 symbols]
```

Figure 8.8: Output Map File, demo.map