SUPPORT NETWORK

Wind River Diab Compiler Linker User's Guide, 5.9.7

30 January 2020

WHEN IT MATTERS, IT RUNS ON WIND RIVER

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Wind River Diab Compiler Linker User's Guide, 5.9.7

30 January 2020

TABLE OF CONTENTS

1. WIND RIVER DIAB LINKER

1.1. About the Linker

The linker is a program that combines one or more *binary object modules* produced by compilers and assemblers into one *binary executable file*. It may also write a text *map* file showing the results of its operation.

Each object module/file is the result of one compilation or assembly. Object files are either stand-alone, typically with the extension **.o**, or are collected in *archive libraries*, also called *libraries*. Library files typically have the extension "**.a**".

An object module contains *sections* of code (also called "text"), and "data", with names such as **.text**, **.data** (variables having initial values), **.bss** (blank sections — uninitialized variables), and various housekeeping sections such as a symbol table or debug information.

The linker reads the sections from the object modules input to it, and based on command-line options and a *linker command file*, combines these *input sections* into *output sections*, and writes an *executable file* (usually; it is also possible to output a file which can be linked again with other files in a process called incremental linking).

A section may contain a reference to a symbol not defined in it — an *undefined external*. Such an external must be defined as *global* in some other object file. A global definition in one object file may be used to *satisfy* the undefined external in another.

As compiled or assembled into an input object file, the first byte of each input section is at address 0 (typically). But when finally located in memory as part of some output section, the input section will not be at address 0 (except for the first input section in an output section that is actually located at 0). Any absolute references to bytes in the section from within the section will therefore be "wrong" and will require *relocation*. The input object file contains sections of *relocation information* which the linker will use to adjust such absolute references. Relocation information is used to make other similar adjustments as well.

Given the definitions above, in the abstract, the linking process consists of six steps:

1. Read the command line and linker command file for directions.
2. Read the input object files and combine the input sections into output sections per the directions in the linker command file. Globals in one object file may satisfy undefined externals in another.
3. Search all supplied archive libraries for modules which satisfy any remaining undefined externals.
4. Locate the output sections at specific places in memory per the directions in the linker command file.
5. Use the relocation information in the object files to adjust references now that the absolute addresses for sections are known.
6. If requested, write a *link map* showing the location of all input and output sections and symbols.

Linking Example

This section provides an example of the above linking process. Consider the following two C files:

File **f1.c**:

```
int a = 1;
int b;

main()
{
        b = 2;
        f2(3);
}
```

File **f2.c**:

```
extern int a, b;

f2(int c)
{
        printf("a:%d, b:%d, c:%d\n", a, b, c);
}
```

The compilation command

```
    dcc -O -c f1.c f2.c
```

generates the object files **f1.o** and **f2.o**.

The contents of the two object files are shown in the following table.

Section	**Type of Data**	**Contents**
`f1.o:`		
`  .text`	Code	Instructions of function main( ).
`  .data`	Variables	Initialized variable **main**, defined in **.text** section. Symbol **a**, defined in the **.data** section. Symbol **b**, **COMMON** block of size 4. Symbol **f2**, undefined external symbol.
`f2.o:`		
`  .text`	Code	Instructions of function f2( ) and the string used in printf( ). 对变量a的引用
`  .rela.text`	Relocation entries	Reference to the variable **a** inside f2( ). Reference to the variable **printf** inside f2( ). Reference to the **printf** string inside f2( ).
`  .symtab`	Symbol table entries	Symbol _**f2**, defined in the **.text** section. Symbol _**a**, undefined external symbol. Symbol _**b**, undefined external symbol. Symbol _**printf**, undefined external symbol.

Section	Type of Data	Contents
		Local symbol for the **printf** string, defined in the **.text** section.

Note that for ColdFire, the actual symbol names are prefixed with an underscore ("_") by the compiler to avoid name clashes with register names, etc.

Invoking the linker explicitly using the **dld** command is fully described in the *dld Linker Command* chapter. However, the easiest way to invoke the linker is to use one of the compiler drivers, for example, **dcc**, as follows:

```
    dcc f1.o f2.o -o prog
```

The driver notes that the input files are objects (**f1.o** and **f2.o**) and invokes the linker immediately, supplying default values for the library, library search paths, linker command file, etc. To see how the linker is invoked, add the option **-#** to the above command; this option directs the driver to display the commands it uses to invoke the subprograms.

大致上
Schematically, the result will be as follows:

```
dcc f1.o f2.o -o prog -#
dld -YP,search-paths -l:crt0.o f1.o f2.o -o prog -lc
        versionDir/conf/default.dld
```

The -YP option specifies directories which the linker will search for libraries specified with "-l" options and files specified with "-l:*filename*" options. **crt0.o** is the C start-up module. The -lc option directs the linker to search for a library named **libc.a** in the paths specified by -YP.

With this command, the linker will proceed as follows:

1. The text file is assumed to be a linker command file (**default.dld** here), input object files are scanned in order (**crt0.o**, **f1.o**, and **f2.o**), and archive libraries are searched as necessary for undefined externals (the library filename **libc.a** is constructed from the option -lc).
   - In this link, the file **printf.o** is loaded from **libc.a** because **printf** is not defined in the **f1.o** or **f2.o** objects. **printf.o**, in turn, needs some other files from **libc.a**, such as **fwrite.o**, **strlen.o**, and **write.o**.
2. Per the directions in the **default.dld** linker command file, input sections with the same name are combined into one output section. In this instance all **.text** sections from **crt0.o**, **f1.o**, **f2.o**, **printf.o**, **fwrite.o**, etc. are concatenated into a single output **.text** section. This also done for the other input sections. The linker command language can be used to specify how sections should be grouped together and where they should be placed in memory.
3. All "common blocks" not defined in **.text** or **.data** are placed last in the **.bss** section. See COMMON Sections on page 7 for details. In this case four bytes for the variable **b** are allocated in **.bss** section.
4. Once the location of all output sections is known, the linker assigns addresses to all symbols. By default, the linker puts the **.text** section in one area of memory, and concatenates the **.data** and the **.bss** sections and locates the result in another area in higher memory. However these defaults are seldom adequate in an embedded system, and memory layout is usually controlled by a linker command file (*versionDir* **/conf/default.dld** in this example). 调整输出段的位置
5. All input sections are copied to the output file. While copying the raw data, the linker adjusts all address references indicated by the relocation entries. Note that there is no space in the input object file or output executable for **.bss** sections because they will be initialized by the system at execution time. The same is true for **.zbss** sections, if present (TriCore); and for **.sbss** sections, if present (ColdFire, MIPS, PowerPC, RH850, and TriCore). .bss段没有内存空间，会在执行时间初始化
6. An updated symbol table is written (unless suppressed by the -s, strip option).

Note: While **.bss** sections do not occupy space in the linker output file, if converted to Motorola S-Records, S-Records are generated to set the space to zeros. The same behavior is true for **.zbss** sections (for TriCore), and **.sbss** sections (for ColdFire, MIPS, PowerPC, RH850, and TriCore).

To suppress this, use the -v option to the -R command for **ddump** (see the Wind River Diab Compiler Utilities Reference).

Note: The linker accepts object files in ELF (Executable and Linking Format) and COFF (Common Object File Format) formats. It generates ELF if any modules are ELF, COFF if all are COFF. See also the -Xelf and -Xcoff entries in Linker -X Options on page 67. Mixing ELF and COFF modules is not recommended — they have incompatible calling conventions.

The order of files on a command line (for either **dld** or **dcc**) will affect the placement order of global symbols in ELF output files; however, the resulting executable program will not be affected.

1.2. Symbols Created By the Linker

If necessary, the linker creates the symbols listed below at the end of the link process.

The linker does not recreate symbols that the user has already defined, or create symbols that are never referred to in any module.

These symbols can be used in C or assembly programs, for example, in startup code to initialize **.bss** sections to zero, or to "copy ROM to RAM" (see chapter 3: *Linker Command Language*, *Example: Copying Code from ROM to RAM Without Copytables* for an example of the latter). That is, a module may declare these symbols as external and use them without ever defining them in any module. The linker will then create the symbols as described during the linking process, and satisfy the external by referring to the created symbol.

链接过程会创建external 符号

.endof*section-name*

Address of the last byte of the named section. (Note 1)

.sizeof*section-name*

Size in bytes of the named section. (Note 1)

.startof*section-name*

Address of the first byte of the named section. (Note 1)

etext, **_etext**

First address after final input section of type **TEXT**. (Note 2)

edata, **_edata**

First address after final input section of type **DATA**. (Note 2)

end, **_end**

First address after highest allocated memory area.

sdata, **_sdata**

First address of first input section of type **DATA**. (Note 2)

stext, **_stext**

First address of first input section of type **TEXT**. (Note 2)

The following symbols are required for shared library support in VxWorks. Note that they are not generated by the linker, but are defined in the **.dld** file.

__GLOBAL_OFFSET_TABLE_, __PROCEDURE_LINKAGE_TABLE_, __DYNAMIC

Base addresses for access to data in the **.got**, **.plt**, and **.dynamic** sections.

Notes:

1. **.endof...** , **.sizeof...** , and **.startof...** cannot be used in C code because C identifiers must include only alphanumeric characters and underscores. But they can be used in assembly code. See the *Unary Operators* section in the *Wind River Diab Compiler User's Guide*.
2. See *type-spec* in Type Specification and KEEP Directive on page 27, for a discussion of output section types **DATA** and **TEXT**. As noted there, if an output section contains more than one type of input section, then its type is a union of the input section types. In this case, the symbols related to the **DATA** and **TEXT** sections as described above are not well-defined.

For example, the following prints the first address after the highest allocated memory area:

```
extern char end;

main() {
        printf("Free memory starts at 0x%x\n",end);
}
```

Note that for ColdFire, MIPS, PowerPC, RH850, and TriCore, **end** must be an incomplete array to ensure that it is not placed in the small data area, as follows:

```
    extern char end[];
```

Symbols in the Linker Command File on page 5
The default linker command files define additional symbols.

1.2.1. Symbols in the Linker Command File

链接器命令行文件中的符号

The default linker command files define additional symbols.

Many of the symbols are required by the initialization code in **init.c** or the startup module **crt0.s**. See the *Use in an Embedded Environment* chapter of the *Wind River Diab Compiler User's Guide*. For more details refer to the linker command file for your architecture. See chapter 2: *dld Linker Command* and chapter 3: *Linker Command Language*.

Some examples:

__DATA_ROM

Address of initialized data in ROM

__DATA_RAM

Address of initialized data in RAM

__DATA_END

End of allocated initialized data

__BSS_START

Start of uninitialized data, to be cleared when the application loads

__BSS_END

End of uninitialized data

_SDA_BASE_, __SDA_BASE_

For ColdFire, MIPS, PowerPC, RH850, and TriCore.

The base address for access to data in the **.sdata** and **.sbss** sections. Defined as the base of the output **.sdata** section + 0x7ff0, but only if at least one of these sections is present. Loaded into the register identified below by startup module **crt0.o**.

- ColdFire: **a5**
- MIPS: **$28**
- PowerPC: **r13**
- RH850: **r4**
- TriCore: **%a0**

For more information, see the *Wind River Diab Compiler User's Guide*.

_SDA2_BASE_, __SDA2_BASE_

For MIPS, PowerPC, and RH850.

The base address for access to data in the **.sdata2** section. Defined as the base of the output **.sdat** 2 section + 0x7ff0, but only if this section is present. Loaded into the register identified below by startup module **crt0.o**.

- MIPS: **$23**
- PowerPC: **r2**
- RH850: **r5** (tp)

The following are for TriCore:

_SMALL_DATA_

Equivalent to **__SDA_BASE_**, to conform to the TriCore ABI.

__ZBSS_START, __ZBSS_END

Start and ending addresses for uninitialized data accessed via 18-bit absolute addressing.

__ZDATA_START, __ZDATA_END

Start and ending addresses for initialized data accessed via 18-bit absolute addressing.

__CSA_LOW, __CSA_HIGH

Start and ending addresses for the Context Save Area (CSA). See the TriCore Architecture Manual for more on Context Save Areas.

_LITERAL_DATA_

Data items in the literal data section (**.ldata**) are relocated relative to the literal data pointer register (**%a1**). **%a1** is initialized using this value.

Parent topic: Symbols Created By the Linker on page 4

1.3. .Abs Sections

Input files may contain sections with names of the form **.abs.***nnnnnnnn* , where *nnnnnnnn* is eight hexadecimal digits (zero-filled if necessary). Such sections will automatically be located at the address given by *nnnnnnnn* .

The compiler generates such sections in response to **#pragma section** directives of the form

# pragma section *class_name* [*addr_mode*] [*acc_mode*] [**address=***n*]

where the value given the **address=***n* clause becomes the *nnnnnnnn* in the section name.

1.4. COMMON Sections

Refer to the information on **COMMON** sections.

Common variables are public variables declared either:

- In compiled code outside of any function, without the **extern** or **static** qualifier, and which are not initialized, e.g. at the module level:

```
        int x[10];
```

- With **.comm** or **.lcomm** in assembly language.

Such variables are assigned to an artificial **COMMON** section:

```
    .mysection : { *[COMMON] }
```

The linker gathers all common variables together and appends them to the end of the output section named **.bss** ; that is, the combined artificial **COMMON** sections for all modules becomes the end of the **.bss** output section.

These are the standard actions if the -Xbss-common-off option is not used. If the -Xbss-common-off option is used:

- There must be exactly one definition of each such variable in the modules of a link, with all other declarations being **extern** or **.xref**, or the linker will report an error.
- Each such variable will be part of the **.bss** section for the module in which it is defined. Because the location of individual sections may be controlled on a per file basis when linking, such variables can be located more precisely.

If an incremental link is requested (option **-r), COMMON** sections are allocated only if the -a option is also given.

Small Data Variables

Because the compiler does not create a “small common,” small data variables are treated as if covered by -Xbss-common-off (for ColdFire, MIPS, PowerPC, RH850, and TriCore).

For example, with -Xsmall-data set to its default value (0 for ColdFire, 8 for MIPS or PowerPC), an **int** variable will be located in the

- **.bss** section for ColdFire
- **.sbss** section for MIPS or PowerPC

Linker Command File Requirements with COMMON

As noted above, by default the linker places **COMMON** sections at the end of output section **.bss**. If there is no **.bss** section, then the linker command file must include a *section-contents* of the form **[COMMON]** (see Section Contents on page 23).

SCOMMON Section

The linker can process an **SCOMMON** section, typically holding “small” common variables and sometimes produced by other tools. The syntax in a linker command file is:

```
    .mysection : { *[SCOMMON] }
```

This section is not normally used by the Wind River tools. Just as the **COMMON** section is appended to the **.bss** output section, the **SCOMMON** section, if present, is appended to the **.sbss** output section; if there is no **.sbss** output section, it is appended to the **.bss** output section. If neither the **.sbss** nor **.bss** output section exists, then the linker command file must contain a *section-contents* of the form **[SCOMMON]** (see Section Contents on page 23).

1.5. COMDAT Groups

A COMDAT group is created by using “**o**” for the section type in an assembler **.section** directive.

For more information, see the *Wind River Diab Compiler User's Guide*. The compiler also automatically generates a group for each instantiation of each member function or static class variable in a template in each module where the member function or variable is used.

When the linker encounters identical COMDAT groups, it removes all except one instance and resolves all references to symbols in the COMDAT group to the single instance.

If a non-COMDAT group is present along with one or more identical COMDAT groups, the linker will still collapse the COMDAT groups to one instance, but will treat the symbols in the COMDAT group as *weak*. See the *Wind River Diab Compiler User's Guide* for the treatment of weak symbols.

1.6. Sorted Sections

The **GROUP** definition is the usual way for a user to explicitly control the order of input sections in an output section. A second mechanism for controlling input section order, called *sorted sections* is described here.

For more information on GROUP definitions, see GROUP Definition on page 39.

An input section is a sorted section if its name begins with a period and ends with “**$***nn*”, where *nn* is a two-digit decimal number, for example **.init$15**. The first part of the name (before the **$***nn*) is called the *common section name* and the **$***nn* part is called the *priority*. Input sections can also be assigned priority in the linker command file.

As described beginning on Section-Definition on page 22, a *section-definition* defines an *output section* and may include a list of input sections. The order in the output section of the input sections is undefined. However if the list of input sections includes a common section name, then all input sections having that common section name will be placed together and will be sorted in the output section in order of their ascending priority numeric priority.

An input section having the common section name but no priority suffix is given priority 50. The order among sorted sections with the same priority is undefined.

This sorted section feature is used by the compiler to order sections when generating initialization code. See the *Run-time Initialization and Termination* section of the *Wind River Diab Compiler User's Guide* for details.

1.7. Warning Sections

If a section is named **.warning**, the linker prints the text from that section to standard output as a warning message if any section is loaded from the file.

The warning is printed only during the final linking; incremental linking will put such sections into the output file. This is useful when the library has stub functions that need to be replaced.

Example:

```
#pragma section DATA ".warning" N

char __warning[] = "No chario output routine has been given.\n"
        "Printing through write() or printf() will not work.\n";

#pragma section DATA

int __outchar(int c, int last)
{
}
```

The linker prints the following message:

```
dld: warning:
No chario output routine has been given.
Printing through write() or printf() will not work.
```

1.8. .Frame_info Sections

The compiler generates **.frame_info** sections for C++ programs when exception-handling is enabled.

A section is created for any function that might appear on the call stack between a **try** and a **throw**; the linker concatenates these into a searchable table that is used for stack-unwinding and object clean-up after an exception occurs. For each function, the table contains a small (8- to 24-byte) record that includes pointers to structures in the **.data** section. Since the C++ support functions in **libd.a** are compiled with exception-handling enabled, most C++ programs have at least some **.frame_info** data.

By default, C functions do not have **.frame_info** sections. To generate **.frame_info** sections for C functions—essential in mixed programs in which C++ exceptions may propagate back through C functions—use the -Xframe-info compiler option. Throwing an exception through C code that is not compiled with -Xframe-info results in a call to the C++ standard-library terminate( ) function. Pure C++ applications and applications that only call C from C++, never the other way around, do not need to use -Xframe-info.

1.9. Branch Islands

This section applies to ARM, ColdFire, MIPS, PowerPC, RH850, and TriCore architectures (not x86).

Using branch islands, the linker can extend the range of a branch instruction. For example, if a branch instruction uses a 16-bit address, the linker can insert a branch island that allows the target symbol to be located beyond the 16-bit limit.

扩展指令长度范围

The branch island is inserted at an intermediate location between the branch instruction and the target address. The linker sets the target of the original branch instruction to an address in the branch island. It then creates a new instruction at that address that branches to the original target address.

For ColdFire M68K processors, the linker can create branch islands for 16-bit (short) branches.

For PowerPC processors, the linker can create branch islands for 16-bit (short) and 26-bit (long) branches.

Related options (not all are available for each architecture):

- -Xbranch-islands
- -Xmax-short-branch
- -Xmax-long-branch
- -Xpic-only

For information about these options, see Linker -X Options on page 67.

1.10. Interworking for ARM and Thumb

For processors that implement the Thumb instruction set, ARM and Thumb code can be mixed.

不同指令基混合

Small code sections called veneers are generated by the linker to change the instruction-set state. Use the option -Xinterwork if Thumb subroutines in the object file might need to return to ARM code or if ARM subroutines might need to return to Thumb code. See the *ARM Developer Guide* for more details.

2. DLD LINKER COMMAND

2.1. Dld Command

Refer to the following information about the dld command.

The linker is invoked by the following command:

```
    dld  [options ]  input-file  . . .
```

The options are described in the Linker Basic Options on page 55 and Linker -X Options on page 67.

The linker decides what to do with each *input-file* given on the command line by examining its contents to determine its type. Each file is either an object file, an archive library file, or a text file containing directives to the linker:

- **Object files**: these are loaded in the order given on the command line.
  - **Archive files**: if there is a reference to an unresolved external symbol after loading the objects, then any archive library files given on the command line (or specified with -l options) are searched for the symbol, and the first object module defining the missing symbol is loaded from the libraries.
  - Library search order depends on the use of the -L, -Y L, -Y U, -Y P, and --Xrescan-libraries options. For more information, see the entries for -Y and -Xrescan-libraries... in Linker Basic Options on page 55 and Linker -X Options on page 67.
  - Archive libraries may be built with the **dar** tool. Archive libraries built by other archivers must conform to the ELF or COFF formats accepted by the linker.
- **Text files**: a text file is interpreted as a file of linker commands. These commands are described in chapter 3: *Linker Command Language*. More than one linker command file is allowed.

Linker Command Structure

A typical linker command will be as follows in outline (where "…" means repetition, and on one line when entered):

```
 dld -YP, search-paths  -o  output-file-name  -l:startup-object-file  object-file ...
          library...  -l libs...  linker-command-file
```

where:

-YP,*search-path* s

Directories to search for files named by -l options and -l: options. The paths for the default directories are based on the default target. See the -t entry in Linker Basic Options on page 55.

(Search paths can also be specified using other -Y options and the -L option as described later).

-o *output-file-name*

Options to specify the name of the output file (the default is **a.out** if no -o option is given).

-l:*startup-object-file*

Startup object file. Link this file first to help establish the order of various initialization sections. Searched for in the directories specified by -Y or -L options (no path prefix allowed). Because the first character after -l is "**:**" the search is for a file with the exact name following the colon. Contrast with -l below.

Alternatively, the startup object file can be named directly on the command line, in which case a path prefix is allowed.

object-file ...

The object files to be linked.

library ... -l*libs* ...

Libraries to be searched for modules defining otherwise undefined external symbols. Libraries can be given directly on the command line with path prefix, or searched for in the -Y or -L directories by using the -l*name* form. In the latter case, the library name **lib***name* **.a** is constructed from *name* and no path prefix is allowed.

linker-command-file

Text file of linker commands. A path prefix is allowed.

To get a map to **stdout**, add the -m, -m2, or -m6 option (with increasing detail).

A good way to gain experience with linker command lines, and to see default values for the parts of the command line outlined above, is to invoke **dcc** or **dplus** with the -# option to show the command line for each subprogram. See the entry for -#, -##, -### in the *Wind River Diab Compiler Options Reference* . For example, the following command line:

```
    dcc -# -o hello.out hello.c
```

compiles and links **hello.c** and prints the full commands on the console output.

2.2. Linker Defaults

In addition to application input object files, the linker typically needs a linker command file to direct the link, libraries to satisfy undefined externals, and often a startup object file.

When the linker is invoked explicitly with the **dld** command, there will be no default linker command file, no libraries, and no startup file—all must be specified using command-line options as described in this chapter.

当链接器由dcc或dplus驱动程序自动调用时

When the linker is invoked automatically by the **dcc** or **dplus** drivers, it is invoked with options that specify default linker command file, libraries, and startup object file. The defaults are defined in configuration files in the **conf** directory. For complete details see appendix A: *Configuration Files* in the *Wind River Diab Compiler User's Guide*.

Some defaults are as follows:

Linker command file:

In most instances, the default is

versionDir **/conf/default.dld**

For PPC processors using the VLE (Variable Length Encoding) instruction set, two other, special VLE linker command files are automatically substituted for **default.dld** for certain compilation environments:

- For the **:cross** and **:simple** environments, **defaultvle.dld** is selected in place of **default.dld**.
- For the **:windiss** environment, **windissvle.dld** is used.

For the TriCore **:windiss** environment, **windisstc.dld** is used.

Generally, the default linker command file should be used as a starting point for creating a custom file for a particular application.

以默认的dld文件为模板，编写自定义文件

To specify a different linker command file when using **dcc** or **dplus**, use the -Wm*file* option (see the Wind River Diab Compiler Options Reference). Note that -Wm is an option to the compiler driver directing its sub-invocation of the linker; -Wm is not a linker option. To provide a linker command file when invoking the linker directly, just name it on the **dld** command line.

Libraries:

the defaults are libraries **libc.a** and, for C++, **libd.a** from the directories associated with the default target, and/or as specified with -l, -L, and/or -Y options on the command line as documented later in this manual.

Startup object file:

the default is **crt.o** from the selected target subdirectory. To specify a different startup object file when using **dcc** or **dplus**, use the -Ws*file* option (see the Wind River Diab Compiler Options Reference). As with -Wm this is a driver, not a linker, option.

To see the defaults for a particular case, execute **dcc** or **dplus** with the -# option to display the command line for the compiler, assembler, and linker as each is automatically invoked.

2.3. Order on the Command Line

Options and files may be intermixed and may be given in any order except that an option which specifies a search directory for -l, that is -L or -Y, must be given before a -l to which it is to apply.

However the following order is recommended:

- options
- object files
- libraries and **-l** options which name libraries
- linker command file

Other options may be mixed in any order. While libraries and objects may be in any order (with the default setting of -Xrescan-libraries; see Linker -X Options on page 67), a link will be faster if there is no need to re-scan a library. The linker may also be more efficient in processing a linker command file if its has encountered all objects first.

2.4. Passing Options with Arguments to the Linker

If you pass options with arguments to the linker from the compiler driver with -Wl or -W:ld:, ensure that you're using the correct command-line syntax.

If you invoke the linker directly from the command line, then using an option with an argument is straightforward:

```
% dld <option> <arg>  file1.o file2.o
```

For example, if you're using the -rpath option to specify **/romfs/lib** as the directory in which the linker should search for shared files, the command line looks like this:

```
% dld -rpath /romfs/lib file1.o file2.o
```

Likewise, it's similarly straightforward to pass an option with no arguments to the linker from the compiler driver with -Wl or -W:ld: (in these examples, we'll use dcc as the driver):

```
% dcc -Wl,<option>  file.c
```

For example, with the -Xstop-on-redeclaration option:

```
% dcc -Wl,-Xstop-on-redeclaration file.c
```

However, when passing options with arguments to the linker from the driver, this syntax won't work:

```
% dcc -Wl<option> <arg>  file.c
```

because *<option>* will be passed to the linker, but *<arg>* will not. (*<arg>* will be "consumed" by the driver instead.)

Instead, use either of the following syntaxes:

```
% dcc -Wl,<option> , -Wl,<arg>  file.c
```

or, more simply:

```
% dcc -Wl,<option> ,<arg>  file.c
```

So, for our -rpath example above, you could use

```
% dcc -Wl,-rpath, -Wl,/romfs/lib file.c
```

or

```
% dcc -Wl,-rpath,/romfs/lib file.c
```

3. LINKER COMMAND LANGUAGE

3.1. About Linker Command Language

Refer to the following uses of the linker command language.

Use the linker command language to:

- Specify input files and options for linking.
- Specify how to combine the input sections into output sections.
- Specify how memory is configured and assign output sections to memory areas.
- Assign addresses or other values to symbols.

A default linker command file, **default.dld**, is present in the **conf** directory. See Linker Defaults on page 12 for its use.

Note: While upper-case or lower-case variants are accepted for some directives, specifications or statements, Wind River recommends that you only use them as described in this document.

3.2. Example bubble.dld

Some examples in this chapter are drawn from the **bubble.dld** command file for the “bubble sort” program.

Example

For more information on this example, see the *Wind River Diab Compiler Getting Started* manual.

This example is for PowerPC. For other architectures, see the directories under **diab/***versionDir* **/example/***archName* .

Table 1. bubble.dld Linker Command File Extract

Linker Commands	Explanation
`-Xgenerate-copytables`	Instruct the linker to create copy tables.

```
MEMORY
{
    rom1:   org = 0x20000, len = 0x10000
    rom2:   org = 0x30000, len = 0x10000
    ram:    org = 0x80000, len = 0x30000
    stack:  org = 0xb0000, len = 0x10000}
```
	Define memory areas.

```
                .text: { *(.text) *(.init) *(
.fini) }
```

Linker Commands	Explanation

```
                  .ctors (=TEXT) ALIGN(4):
                             { ctordtor.o(.ctors
) *(.ctors) }
                         .dtors (=TEXT) ALIGN(4
):
                             { ctordtor.o(.dtors
) *(.dtors) }
```
	.ctors and **.dtors** sections for startup and termination invocation.

<pre> .sdata2 : {}</pre>	Also, locate **.sdata2** with the **.text** sections (**.sdata2** is used for "small" **const** variables; and it is of 0 length in the example).

```
                  .copytable : {}
      } > rom1
```
	.copytable section defines location of linker generated copy table.

```
     GROUP: {
                  .text2 : { *(.text2)}
     } > rom2
```
	Collect all **.text2** sections and locate in **rom2**.

```
     GROUP LOAD (>rom2) COPYTABLE : {

         .data (DATA) : {}
```
	Group **.data**, **.sdata**, **.sbss**,and **.bss** output sections together in the order given. Collect initialized data sections (.**data**) from all input files "**{ }**" into a single output **.data** section and locate logically in RAM memory. Due to the **GROUP**'s **LOAD** specification with a **COPYTABLE** specification, place the data after the **.text2** section in **rom2**, and create a copy table entry, with physical address being set to the start address of **.data** in **rom2**, with the logical address being set to the start address of **.data** in RAM, and with **size** being set to the size of the **.data** section. **__init_main ( )** moves the data from **rom2** to **ram** (using **__copy_multiple( )**).

<pre> .sdata (DATA) : {}</pre>	Similarly locate **.sdata** logically in RAM, and due to the **GROUPS**'s **LOAD** specification, place it physically after **.data** in **rom2**. Create a copy table for **.sdata** as well, due to the **GROUP**'s **COPYTABLE** specification.

```
         .sbss (BSS): {}
         .bss  (BSS): {}
```
	Reserve space for all **.bss** and **.sbss** sections in RAM after the **.data** section. Create copy table entries for **.bss** and **.sbss** sections, with the physical address being set to zero, with the logical address being set to the start address of the section in RAM, and with size being set to the size of the section. The fact that the physical address is set to zero marks the section as **bss** and **__copy_multiple( )** zeros out the RAM areas instead of copying. The remaining space is used as heap by **malloc( )**.

```
         __HEAP_START = .;
     } > ram
  }
```
	Define other symbols used by **crt0.s**, **init.c**, and **sbrk.c** to control initialization and memory allocation. Start of heap memory for **sbrk.c**.

Linker Commands	Explanation

`__HEAP_END    = ADDR(ram)+SIZEOF(ram);`
`__SP_INIT     = ADDR(stack)+SIZEOF(stack);`
`__SP_END      = ADDR(stack);`
	End of heap memory for **sbrk.c**.
	Initial address of stack pointer for **crt0.s**.
	Only used when stack probing for **sbrk.c**.

Notes for bubble.dld

Two features of **bubble.dld** are especially noteworthy:

- Use of the **LOAD** specification in combination with the **COPYTABLE** specification to create two images of variables having initial values, a *physical* image containing the initial values and intended for some form of read-only memory, and a *logical* image where the variables will reside during execution. See the LOAD Specification on page 32 and the *Copying Initial Values From ROM to RAM and Initializing .bss* section in the *Wind River Diab Compiler User's Guide* for details.
- Definition of the symbols:
  - **__HEAP_START** and **__HEAP_END** to define the heap for use by malloc( ) and related functions. See the *Dynamic Memory Allocation - the heap, malloc(), sbrk()* section in the *Wind River Diab Compiler User's Guide*.
  - **__SP_INIT** and **__SP_END** to define the stack. See the *Stack Initialization and Checking* section in the *Wind River Diab Compiler User's Guide*.

Note: The Wind River Diab Compiler for ColdFire prepends an underscore to all symbol names. For example, the symbol **__DATA_ROM** (two underscores) in **init.c** is **___DATA_ROM** (four underscores) in the linker command file.

3.3. Syntax Notation

Italic words such as *area-name* represent items you must supply. The required type of each item — symbol name or number, can be gathered from the examples.

The following special characters are parts of commands and are required where shown:

```
    {  }  (  )  ,  ;  >  *
```

The following characters are used only in the command descriptions and not in the linker command language itself. They have the meanings shown:

|

    "or"

[]

    The enclosed construct is optional. When several optional items are adjacent, they may be given in any order.

...

    The preceding item or construct may be repeated.

For example

```
    a [b | c] ...
```

means that **a** is required, then any number of **b** or **c**.

Note: The “{” and “}” characters are part of commands and do not indicate a set of alternatives from which one must be chosen.

Long lists of alternative tokens are given by following the phrase “one of” with a list of the tokens on one or more lines, as in

```
assign-op : one of
    =   +=   -=   *=   /=
```

3.4. Pattern Matching in Linker Command Files

The linker supports UNIX-style (filename) pattern matching. Patterns can be used to specify section contents.

- * matches any string, including the null string.
- ? matches any single character.
- [seq] matches any character in seq, where seq can also be a range given as x-y
- [!seq] or [^seq] matches any character not in seq, where seq can also be a range given as x-y
- For a literal match, wrap the meta-characters in brackets. For example, '[?]' matches the character '?'.

fnmatch(pattern, name, 0) is called to perform the pattern matching. Consult documentation on this function for more information.

Note: Any pattern more complex than * should be enclosed in double quotes in the linker command file.

See also Section Contents on page 23.

3.5. Numbers

Several linker commands require a number, for example to specify an address or a size.

Numbers are hexadecimal if they begin with “0X” or “0x”, else octal if they begin with “0”, else decimal. Hexadecimal digits are “0” - “9”, “a” - “f”, and “A” - “F”; octal digits are “0” - “7”; decimal digits are “0” - “9”.

3.6. Symbols

A *symbol* , once defined, may be used anywhere a number is required except in a **MEMORY** command.

Symbols are defined in object files or by assignment commands (see Assignment Commands on page 40).

A *symbol* defined in an assignment command is an identifier following the rules of the C language with the addition of "**$**" and "**.**" as valid characters. Symbols may be up to 1,000 characters long.

Note: A symbol or filename which does not follow these rules may be given by quoting it with double-quote characters, for example, an object file named "**1234o.o**".

3.7. Expressions

A linker *expression* is allowed anywhere a number is required.

A linker *expression* is one of the following forms from the C language:

```
number symbol unary-op expression
          expression binary-op expression expression ?expression :expression    (expression )
```

where the operators are the following operators from the C language:

unary-op : one of

```
                        !   ~   -
```

binary-op : one of

```
 *   /   %  +   -  >>   <<
                                ==   !=   >  <   <=   >=  &  |  &&  ||
```

The operators have their meaning and precedence as in C. Parentheses can be used to change the precedence.

See also Syntax Notation on page 17.

When a symbol name is used in an expression, the address of that symbol is used. The symbol "`.`" means the current location counter (allowed only within a statement list in a **SECTIONS** command).

The following pseudo functions are valid in expressions. Forward references are permitted.

SIZEOF (*section-name*)

Size of the named output section. See *Example: Empty Sections* for an important limitation when using the **SIZEOF** operator. **SIZEOF** includes alignment gaps between consecutive output sections. See alternative operator **RAWSIZEOF**.

RAWSIZEOF (*section-name*)

For **SIZEOF** the reported size of an output section depends on the alignment of the next output section. In case of there will be an alignment gap between two consecutive output sections **SIZEOF** includes that alignment gap in the computed size of the first section. **RAWSIZEOF** produces a number that does not include this alignment gap.

SIZEOF (*memory-area-name*)

Size of a memory area defined with the **MEMORY** command.

ADDR (*section-name*)

Address of the named output section.

ADDR (*memory-area-name*)

Address of a memory area.

NEXT (*expr*)

First multiple of *expr* that falls into unallocated memory.

HEADERSZ

Total size of all the headers.

FILEOFFSET (*section-name*)

File offset of the named section.

ALIGN (*value*)

((. + *value* **–1) & ~(** *value* **–1))**

3.8. Command File Structure

A command file is a list of commands.

These commands are:

```
MEMORY {memory-area-definition }
SECTIONS {  section-or-group-definition ... }
assignment-command
object-filename
archive-filename
command-line-option
```

The above commands may each be repeated as many times as required and may be given in any order as long as names are defined before use.

Each of these commands is described below except for the last three: in addition to, or instead of, being given as arguments on the command line, object and archive library files and command-line options may be given as commands.

Note: While different object files may be named on both the command line and in a linker command file, do not duplicate the same object filename in both places. This may cause sections from the duplicated object file to be duplicated in memory.

The command language is free format. More than one command may be given on a line, and a command may be written on multiple lines without need for any special continuation character.

Identifiers are as in C with the addition of period "**.**" and "**$**" as a valid identifier characters; identifiers may be up to 1,000 characters long.

Whitespace is generally required as in C around identifiers and numbers but not special characters.

C-style comments are allowed anywhere whitespace would be.

3.9. MEMORY Command

The **MEMORY** command names one or more areas of memory, e.g. ROM, RAM. Each area is defined by a start address and a length in bytes.

The **MEMORY** command is written in the following format:

```
MEMORY {  area-name :  {  origin  |  org  |  o}  = start-address [  ,  ]
                 {  length  |  len  |  l   }  =   number-of-bytes  [,  ]
    ...
```

}

A later *section-definition* command can then direct that an output section be located in a named area. The linker will warn if the total length of the sections assigned to any area exceeds the area’s length. Example:

```
MEMORY {
    rom1:   org = 0x010000, len = 0x10000
    rom2:   org = 0x020000, len = 0x10000
    ram:    org = 0x100000, len = 0x70000
    stack:  org = 0x170000, len = 0x10000
}
```

Symbols (Symbols on page 18) cannot be used within the **MEMORY** command; *start-address* and *number-of-bytes* must be numeric expressions or **ADDR** and **SIZEOF** expressions that use memory area names that have already been defined.

For example:

```
    MEMORY
    {
        ApplicationExceptionTable:
            org = 0x00010000,len = 0x200
        ApplicationLinkInfo:
            org = ADDR(ApplicationExceptionTable) +
            SIZEOF(ApplicationExceptionTable), len=0x200

        rom1:   org = 0x20000, len = 0x10000  /* 3rd 64KB      */
        rom2:   org = 0x30000, len = 0x10000  /* 4th 64KB      */
        ram:    org = 0x80000, len = 0x30000  /* 512KB - 703KB */
        stack:  org = 0xb0000, len = 0x10000  /* 7043B - 768KB */
    }
```

3.10. SECTIONS Command

The **SECTIONS** command does most of the work in a linker command file.

The **SECTIONS** command is written in the following format:

```
SECTIONS {
    section-definition  | group-definition
    ...
}
```

Each input object file consists of *input sections*. The primary task of the linker is to collect input sections and link them into *output sections*. The **SECTIONS** command defines each output section and the input sections to be made part of it. Within the **SECTIONS** command, a **GROUP** statement may be used to collect several output sections together.

Section-Definition on page 22
At a minimum, each *section-definition* defines a new *output section* and specifies the *input sections* that are to be put into that output section.

Section Contents on page 23
section-contents is required in a *section-definition*.

3.10.1. Section-Definition

At a minimum, each *section-definition* defines a new *output section* and specifies the *input sections* that are to be put into that output section.

Optional clauses may:

- Specify an address for the output section or place the output section in a memory area defined by an earlier **MEMORY** command.
- Align the section.
- Fill any holes in the section with a fixed value.
- Define symbols to be used later in the linker command file or in the code being linked.

The full form of a *section-definition* is shown below. For clarity, each clause is written on a separate line and is identified to its right.

Note: The **REGISTER** element is only supported for PowerPC.

Syntax	Element
output-section-name	
[([=] [**BSS** | **COMMENT** | **CONST** | **DATA** | **TEXT** | **BTEXT**] [**OVERLAY**] [**NOLOAD**] [**OPTIONAL**] . . .)]	*type-spec*
[*address-value* | **BIND** (*expression*)]	*address-spec*
[**ALIGN** (*expression*)]	*align-spec*
[**LOAD** (*expression*)]	*load-spec*
[**OVERFLOW** (*size-expression* , *overflow-section-name*)]	*overflow-spec*
[**REGISTER** (register-number [offset])]	*SDA-register-spec* (for PowerPC only)
[**ALIAS** (*reference* , *definition*)]	*alias-spec*
[**RESERVE** (*expression*)]	
[**ENDOF** (*expression*)]	
:	
{ *section-contents* }	
[=*fill-value* | = (*fill-value* [, *size* [, *alignment*]])]	*fill-spec*
[=**CRC16** | **CRC32** (*crc-start* [, *crc-size* [, *crc-initial*]])]	*CRC-spec*
[>*area-name*]	*area-spec*

Most clauses are optional, and section modifiers (those preceding the ":") may be in any order. Thus, the minimum *section-definition* has the form:

```
output-section-name : { section-contents }
```

Note: Exercise caution when naming custom sections. Section names that begin with a dot (**.**) may conflict with the compilation environment's namespace.

Parent topic: SECTIONS Command on page 21

3.10.2. Section Contents

section-contents is required in a *section-definition* .

section-contents is a sequence of one or more of the forms from separated by whitespace or comment:

(empty)

That is, { } with no explicitly named *section-contents* : include in the output section all sections from all input object files that have the same name as the *output-section-name* . Example:

```
.data : { }
```

Note: The empty form is processed only after the linker has examined and processed all other input specifications. Thus, input sections loaded directly or indirectly as a result of other more explicit specifications will not be re-loaded by an { } form, even if they appear after it.

CLONE(*original*:*clone*)

Instructs the linker to make a copy of the section that includes the symbol *original* and to include it in the link. Within the copy, the linker defines the symbol *clone* at the exact same location (relative to the cloned section) as *original*. In other words, *original* is renamed *clone* in the copy of the section.

The **CLONE** directive may be thought of as constructing an input section. More than one **CLONE** directive may be specified for an output section.

Note the following rules and behaviors:

- The symbol *clone* must not exist in the link. The **CLONE** directive generates a definition for *clone*, and a pre-existing definition will cause a redefinition. If this occurs, the linker issues an error like the following:

  ```
  dld: Attempting to CLONE(original,clone) but
                                  clone already exists dld: error: Cannot CLONE
  ```

  and the link will be aborted unless -Xdont-die is also specified (see Linker -X Options on page 67).
- The entire section that defines *original* is cloned, not just the symbol itself. However only *original* is recognized as being cloned. Should the section define other non-**LOCAL** symbols, they will be flagged as re-declarations.

  Wind River therefore recommends that you use -Xsection-split when you compile code that is meant to be cloned (see Linker -X Options on page 67).
- If *original* is not a symbol that is amenable to being placed in the specified output section, the linker still proceeds with **CLONE** and places *clone* in the output section. For example, you could clone a data symbol into the **.text** section. But the resulting executable might not be usable. In other words, the linker does not perform any checks on the symbols being cloned beyond what is described above.

For an example of **CLONE** use, see *Example: Using CLONE and ALIAS for Core-Local RAM.*

filename

Include all sections from the named object file which have the same name as the *output-section-name*. Example:

```
.data : { test1.o, test2.o }
```

INCBIN(*filename*)

Include the file identified by *filename* as a binary image in the section. The file is added byte-by-byte from the current location counter.

* (*input-section-spec* ...)

input-section-spec may be one of four forms:

section-name

Include the named sections from all input object files but do not include input sections already included earlier. Example:

```
.data : { *(.data) }
```

section-name[*symbol*]

Include the section defining the given symbol. The "**[**" and "**]**" characters do not mean "optional" in this case but rather are to be used as shown. Example:

```
.text : { *(.text[malloc])
                                  }
```

This form is especially useful with option -Xsection-split (see Linker -X Options on page 67).

*

Include all input sections that match the output section type.

Note that to use an "*" as a wildcard as part of a section name, it must be enclosed in double quotes, otherwise it will match all sections. For example, the following will match all sections because it treats **.rodata.** and ***** as two separate items in the input section specification:

(.rodata.*)

To treat **.rodata.*** as a single item, enclose it in double quotes:

(".rodata.*")

input-section-spec **=***n*

Include sections according to *input-section-spec* and assign them priority *n*. (See Sorted Sections on page 8.)

SORT_BY_*XXX* **(***input-section-spec* **)**

Include sections according to *input-section-spec* and sort them according to *XXX*. The *XXX* may be one of the following:

NAME (sort by name)

ALIGNMENT (sort by alignment of sections in ascending order)

ALIGNMENT_DESC (sort by alignment of sections in descending order).

Note that although this is a form of *input-section-spec*, you cannot use it for *input-section-spec* =*n* assignments.

Also note that in order to have the desired sort effect, the *input-section-spec* must resolve to a set of sections that vary in name or alignment. You can generate unique names or alignments, for example, by using -Xsection-split set to **2**, **3**, **6**, or **7** (to work with sorting variables), or by using **#pragma section** and **#pragma use_section** in combination with wild cards in the *input-section-spec*.

As an example, this code

```
 .sbss  (BSS)  :  {
                                  *(SORT_BY_ALIGNMENT_DESC(".sbss"))  }
```

places all input **.sbss** sections into an output **.sbss** section in descending size of alignment.

object-filespec **(** *input-section-spec* ... **)**

Include the named sections from the named object file, where *input-section-spec* is as defined immediately above and *object-filespec* is a pattern expression.

The pattern expression for *object-filespec* follows this syntax:

```
filename  |  {  expression  }
```

where *expression* is one of the following:

! *expression*
expression | *expression*
expression **&** *expression*
(

expression
)
filename

and *filename* is a UNIX-style pattern. For more information, see Pattern Matching in Linker Command Files on page 18.

For example:

```
.rom1
                              : { rom1.o(.data), rom1.o(.sdata) }
```

will read .data and .sdata of rom1.o into .rom1.

```
.rom: {"rom*"(.data), "rom*"(.sdata) }
```

will read in all sections named .data or .sdata from all objects prefixed with rom into .rom.

For example:

- "bar*" matches all strings prefixed with "bar".
- "???" matches any string with a width of 3.
- "[A-Z]*[a-z0-9]" matches all strings prefixed with an uppercase letter and postfixed with a lowercase letter or a digit.
  "[!0-9]*" matches all strings not prefixed with a digit
  "bar[*]" matches exactly "bar*"

archive-filespec[*member-name*] (*input-section-spec* ...)

Include the named sections from the named object file, where *input-section-spec* is as defined above, and *archive-filespec* and *member-name* are pattern expressions, meaning they use the same syntax as described for *object-filespec*, above.

For example:

```
 .text : { libproj.a[malloc.o](.text) }
```

will read in .text of malloc.o in the library libproj.a into .text.

```
text_libfoo.a (TEXT) : libfoo.a[*] (.myText)
```

will read in all sections named .myText from all object files in the archive libfoo.a into the text_libfoo.a (TEXT) output section.

To read in sections named .myText and .myData from only those modules in the library libfoo.a whose name are prefixed with bar, use

```
text_libfoo.a (TEXT) : libfoo.a["bar*"]
                            (".my*")
```

[COMMON]

For explicit placement of **COMMON** sections. See COMMON Sections on page 7 for additional information.

[SCOMMON]

For explicit placement of **SCOMMON** sections. See COMMON Sections on page 7 for additional information.

assignment-command

Define a symbol or change the program counter to create a "hole" (which may be filled by a *fill-value*). See Assignment Commands on page 40.

ASSERT (*expression* [, *text*])

Evaluate *expression* and display an error message if *expression* is zero. Optional *text* is included in error message.

STORE (*expression* , *size-in-bytes*)

Reserve and initialize storage (see STORE Statement on page 38).

The order of the sections listed in the *section-contents* is undefined as is the order of output sections in a **SECTIONS** command. A **GROUP** definition may be used to ensure the order of a set of output sections. (See GROUP Definition on page 39.)

VECTABLE (*pattern* , *entrysize* , *maxentries*)

Build an array of uniformly named and uniformly sized sections like an interrupt vector table.

pattern format is either *name* .%d or *name* .%x. If *name* .%d, the *name* suffix is a decimal number. If *name* .%x, the *name* is a hexadecimal number.

entrysize is the size of one section in the table. A section *name n* is placed at an offset of *n* * *entrysize* from the table start address. An error appears if the input section exceeds the *entrysize* .

maxentries defines the number of sections in the table. For any input section *name n* , *n* must be smaller than the value of *maxentries* , otherwise an error appears.

Note: A section-contents specification must have at least one non-**COMMENT** input section, e.g., a **BSS**, , **DATA**, or **TEXT** section, or the type of the output section will default to **COMMENT**, and it will not be allocated any memory. See below regarding section types.

Parent topic: SECTIONS Command on page 21

3.10.3. Type Specification and KEEP Directive 指令

The *type-spec* clause sets the type of the output section.

If absent, the type will be determined by the types of the input sections. If all input sections in a given output section are of the same type, the type of the output section will be that of the input sections and no *type-spec* clause is necessary. Mixing input sections of different types in a single output section is not recommended. If input sections do have different types, the linker will choose a type from the input sections in the following order from highest priority to least: **TEXT**, **CONST**, **DATA**, **BSS**, and **COMMENT**.

To force the linker to choose the specified type regardless of the types of the input sections, use the "**=**" form. For example, **(=DATA)** will force the output section to have the DATA type.

type-spec can also be used when linking files produced by third-party tools which do not tag each section with its type.

Note: Although **SECTIONS** commands will sometimes work without explicit types being assigned to the sections, it is a good coding practice to always state the type. Failing to do so may produce unexpected results.

The alternative type specifications indicate the expected contents of the section:

(BSS)

Section contains uninitialized data space.

(COMMENT)

Section debug or other information not part of the program memory space.

(CONST)

Section contains initialized data space.

(DATA)

Section contains initialized variables.

(TEXT)

Section contains code and/or constants.

(BTEXT)

Blank text section.

告诉linker可以重叠其他段

OVERLAY tells the linker that the section can overlap other sections. The section should have **BIND** specification; memory is not allocated for it. Example:

```
    .text1  (TEXT OVERLAY) BIND(ADDR(.text)) : { .... }
```

NOLOAD tells the linker not to mark the section as loadable. 不加载

OPTIONAL tells the linker that the section should be discarded if it is empty. 若为空丢弃

For example, specifying

```
    .text1  (TEXT)  ( OPTIONAL ) :
```

means that the **.text1** section will be created only if it will not be empty, and, if it is created, it inherits its type from its contents.

Note: The COFF specifications require that a DATA and a TEXT section be defined in the linker command file.

KEEP Directive

The **KEEP** directive identifies sections that should not be removed by the -Xremove-unused-sections option. It is typically used to prevent removal of interrupt/exception vector tables and boot code.

The **KEEP** directive can be applied to either an output section pattern or an input file/section pattern.

The following example shows **KEEP** used within a **GROUP**.

```
    GROUP : {
    ...
    /* Keep intvtab1K even if it is not referenced */
    .intvtab1 KEEP (DATA) : {      *(.intvtab1K)   }
    /* Keep intvtab2K but not intvtab2R */
    .intvtab2       (DATA) : {KEEP(*(.intvtab2K)) *(.intvtab2R) }
    /* Don't keep intvtab3R (unless explicitly referenced) */
    .intvtab3       (DATA) : {        *(.intvtab3R)   }
    } > ram
```

The following example shows KEEP used outside a GROUP.

```
    /* Don't keep intvtab4R (unless explicitly referenced) */
    .intvtab4R  BIND(0x40000000) : {       *(.intvtab4R)   }
    /* Keep intvtab4K */
    .intvtab4K  BIND(0x60000000) KEEP : {  *(.intvtab4K)   }
```

Parent topic: SECTIONS Command on page 21

3.10.4. Address Specification

The *address-spec* clause specifies the address for the first byte of the output section.

The form of the *address-spec* is:

address-value | **BIND** (*expression*)

It is either an absolute address, *address-value* , or the word **BIND** followed by an expression that can contain the functions **SIZEOF**, **ADDR**, and **NEXT** (see Expressions on page 19). An *address-spec* is not allowed inside a **GROUP** (see GROUP Definition on page 39).

Note: A section with an address specification (address-spec) does not need a memory-area specification (area-spec), since the linker automatically marks the corresponding address range as reserved. If both an address-spec and an area-spec are provided, the linker checks that the address range is completely inside the memory area and displays a warning if it is not.

Parent topic: SECTIONS Command on page 21

3.10.5. ALIAS Specification

The alias-specification instructs the linker to bind all references to the symbol *reference* in the specified input section(s) to the symbol *definition* .

The form of *alias-spec* is

ALIAS (*reference* , *definition*)

Note the following requirements and behaviors:

- **ALIAS** must appear as an attribute in an output section and it applies to all the input sections that are added to the output section.
- If the input sections contain no references to *reference* , the **ALIAS** is ignored. The linker issues a warning like this:

  ```
  dld: warning: Ignoring ALIAS(reference,definition) - no references to
  reference found
  ```

  The link itself proceeds.
- If the input sections contain references to *reference* , then the symbol *definition* must be available to the linker. In other words the linker will generate a reference to *definition* that must be satisfied. The definition for *definition* may even be one that the

linker has generated (as would happen with **CLONE**). If no definition exists, the linker issues an error like the following and the link aborts unless -Xdont-die is also specified (see Linker -X Options on page 67):

```
    dld: error: Unknown symbol definition is required by ALIAS
```

- The linker does not look up libraries for *definition* .
- It is not necessary for other references to *definition* to exist in addition to **ALIAS**.
- A definition for *reference* may exist in the link.

Parent topic: SECTIONS Command on page 21

3.10.6. ALIGN Specification

An *align-spec* clause causes the linker to align the section on the byte boundary given by the value of *expression* .

The form of the *align-spec* is:

ALIGN (*expression*)

Parent topic: SECTIONS Command on page 21

3.10.7. COPYTABLE Specification

Copy tables are linker-generated tables that direct overlays. They are generated wherever content is copied from a load-time/physical address (such as ROM) to a run-time/logical address (such as RAM), or whenever an area needs to be cleared at runtime (e.g., **BSS**).

The form of the *copytable-spec* is:

COPYTABLE [*name*]

To use copy tables:

1. Add -Xgenerate-copytables to your linker command line (see Linker -X Options on page 67).
2. Define **GROUPS** that use a **COPYTABLE** specification.
3. Create an output section to hold the generated copy table.

Copy table entries are generated for each output section in the **GROUP**. Loadable sections (such as **.data**) get entries that include the physical address, the logical address, and the section size. **BSS** sections get similar entries but with a physical address of 0. Two marker symbols, **__COPY_TABLE_BEGIN__** and **__COPY_TABLE_END__** , are generated to delineate the bounds of the generated copy table. Note that the copy table can contain entries from multiple groups.

The generated copy table must also be put in an output section by the linker command file. For example:

```
GROUP : {
    .text {} /*some code*/
    ... /*other stuff*/
    .copytable : {} /*the copytable */
} > rom
```

At runtime the startup code makes a call to

```
__copy_multiple(__COPY_TABLE_BEGIN__, __COPY_TABLE_END__);
```

which carries out the actions defined in the copy table. For each entry with a non-zero physical address the contents are copied from the physical address ("load") to the logical ("run") address. For entries with a zero physical address (**BSS**), the corresponding **BSS** region starting at the given logical address is initialized with zeroes.

Using copy tables is simpler and uses less code than performing a copy "by hand"; for example, instead of

```
GROUP : {
    __DATA_RAM = .;
    .data LOAD(__DATA_ROM) : {}
    .sdata (DATA) LOAD(ADDR(.sdata) - ADDR(.data) + __DATA_ROM) : {}
    __DATA_END = .;
    __BSS_START = .;
    .sbss (BSS) : {}
    .bss (BSS) : {}
    __BSS_END = .;
} > ram
```

you can achieve the same effect with much less code:

```
GROUP LOAD(>rom_d) COPYTABLE : {
    .data : {}
    .sdata (DATA) : {}
    .sbss (BSS) : {}
    .bss (BSS) : {}
} > ram
```

The compiler startup code can handle both of the above examples, but it is recommended that new projects use copy tables.

Named Copy Tables

Named copy tables exist to support the need for multiple copy tables. For example, on a multi-core system the primary core might copy its data with a call to __copy_multiple( ), but secondary cores would copy theirs at a later time and would have completely different data to copy. Putting all the data into one copy table may not be appropriate, so you would want to create separate copy tables, one for each core.

All copy tables go into a single section. Given **COPYTABLE** [*name*], the linker defines two symbols, **NAME** and **NAME__END__**, to mark the ends of the copy table.

The default startup code contains the following:

```
__copy_multiple(__COPY_TABLE_BEGIN__, __COPY_TABLE_END__)
```

which performs a copy using all the copy tables. For this reason, if you use one named copy table, then all copy tables must be named.

Remove the line above from the startup code and add calls to

```
__copy_multiple(NAME,NAME__END__)
```

for each of your named copy tables; this code performs a copy using just that copy table, independent of the other tables in the section.

See also, *Example: Using CLONE and ALIAS for Core-Local RAM*.

You can rename the section into which the copy tables go with -Xcopytable-section-name (see Linker -X Options on page 67).

Parent topic: SECTIONS Command on page 21

3.10.8. ENDOF Specification

The *endof-spec* evaluates the address just beyond the last byte, including any padding, of section.

The form of *endof specification* is:

ENDOF (*expression*)

The **RESERVE**'d space would reflect in the value computed by **ENDOF**.

You can also write **ENDOF** (*region_2*) - **ADDR** (*region_2*) and receive the full-size of *region_2*.

Note: **ENDOF** applies to memory regions just as sections.

The statement **LOAD** (>rom) is identical to **LOAD** (**ENDOF** (rom)) by definition.

```
region_1 BIND ( 0x10010000 ) KEEP RESERVE ( 0x100 ): {
   region_1_start= .;
   *( .textvle )
   *( .rodata )
   region_1_end= .;
   } = 0x00 > ROM

   region_2 BIND( ENDOF ( region_1 ) ) KEEP : {
   region_2_start = .;
   . = region_2_start + 0x0100 ; region_2_end = .;
   } = 0x00 > ROM
```

Parent topic: SECTIONS Command on page 21

3.10.9. LOAD Specification

The load-spec controls the physical/logical addressing of the section.

The form of the *load-spec* is:

LOAD (*expression*)

In a typical embedded system, the values for all variables with explicit initialization must be stored in some type of read-only memory before the system is powered up. During execution, the variables must themselves be located in RAM so they can be set (except for **const** variables which can remain in ROM). Thus, during startup, the initial values for these variables must be copied from ROM to RAM.

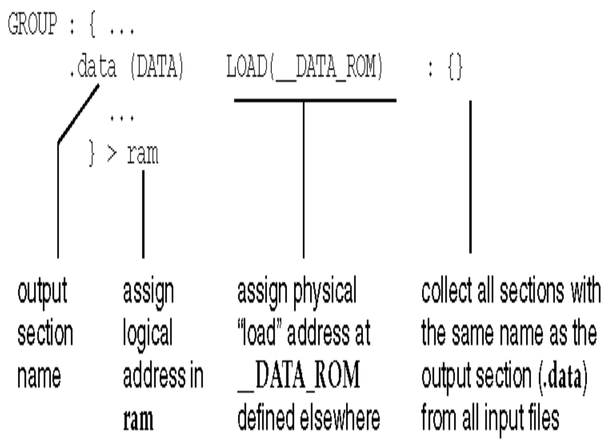
To distinguish these two locations, we refer to the *physical* and *logical* addresses of the output section.

- *physical address*: This is the address given by the *expression* in the *load-spec*. It is this address which is used in the section header when the section is written to the linked output file. Thus, if a dynamic loader loads the section, or the section data is burned into a ROM, it will be at this *physical* address.
- *logical address*: This address is set by an *address-spec* or an *area-spec* in the *-section-definition*. This will be actual address of the section during execution. Thus, when linking references to a variable in the section, the linker will use the variable's *logical* address.

Note: The load-spec only controls the physical/logical addressing of the section. Typically, assignment statements are used to define symbols for the physical and logical addresses of the section and its length. These symbols are then used by startup code to copy the physical data from ROM to its logical location in RAM. See the examples in this chapter, as well as **default.dld** in the **conf** directory and **crt0.s** in the appropriate target directory for the startup copying code.

Also, copying code in the startup module, **init.c**, copies only a single contiguous physical section. Thus, while more than one **LOAD** specification is permitted, the output sections named in the expressions must be contiguous.

The following example shows the **LOAD** specification:


GROUP : { ...
.data (DATA) LOAD(__DATA_ROM) : {}
...
} > ram
output section name
assign logical address in ram
assign physical "load" address at _DATA_ROM defined elsewhere
collect all sections with the same name as the output section (.data) from all input files

Usually, adjacent output sections need to have contiguous physical ("LOAD") addresses. One way to achieve this is by manually computing the physical address of each output section using several **LOAD** directives:

```
GROUP : {
    .data LOAD(__DATA_ROM) : {}
    .sdata (DATA) LOAD(ADDR(.sdata) - ADDR(.data) + __DATA_ROM) : {}
    .sbss (BSS) : {}
    .bss (BSS) : {}
} > ram
```

A simpler way to achieve the same effect is to use a **LOAD** directive on the **GROUP**:

```
GROUP LOAD(>rom_d) : {
    .data : {}
    .sdata : {}
    .sbss (BSS) : {}
    .bss (BSS) : {}
} > ram
```

(Here it is assumed that **rom_d** is an area defined with the **MEMORY** command to hold **DATA** loaded into **ROM**..)

This syntax automatically assigns the next available address in the specified memory area (in this case **rom_d**) as the load address of the next output section. No load address is assigned for non-loadable sections such as **BSS** sections.

Parent topic: SECTIONS Command on page 21

3.10.10. OVERFLOW Specification

The overflow specification enables you to specify the size limit of a section and to request that the linker place input sections which will not fit into the initial section into a different section, called the *overflow section*.

The form of the *overflow-spec* is:

```
OVERFLOW (size-expression , overflow-section-name )
```

The *size-expression* specifies the size of the initial section in bytes, and *-overflow-section-name* names the section that is to receive the input sections that cannot fit into the initial section.

The linker tries to gather input sections into the output section according to their order. If an input section is too large to be accommodated, all remaining sections including itself are moved to the tail of the overflow section. (See also -Xoverflow-greedy.) Other non-hole expressions are always kept.

TriCore Example

The **OVERFLOW** directive may be especially useful for programs compiled in TriCore's 18-bit near-absolute mode, since the absolute addressing range is so small. The *OVERFLOW Specification Example* below shows an excerpt from a linker command file for such compilation. Initialized data goes into **zdata** and uninitialized data goes into **zbss**, which are both 16Kb long; however, if these sections aren't big enough, additional data goes into **zdata_ovfl** and **zbss_ovfl**, also 16Kb in length, respectively.

OVERFLOW Specification Example

```
MEMORY
{
    ram:                    org = 0xa0000000, len = 0x700000
    zbss:                   org = 0xc0000000, len = 0x3fff
    zdata:                  org = 0xb0000000, len = 0x3fff
    csa:                    org = 0xd0000000, len = 0x7fff
    stack:                  org = 0xd2000000, len = 0x2fffff0
    isrsp:                  org = 0xd5000000, len = 0xffff
    zbss_ovfl:              org = 0xe0000000, len = 0x3fff
    zdata_ovfl:             org = 0xf0000000, len = 0x3fff
}

SECTIONS
{
    GROUP : {
        .zdata (ZDATA) OVERFLOW(0x3fff, zdata_ovfl) : { *(.zdata) }
    } > zdata

    GROUP : {
        .zbss (ZBSS) OVERFLOW(0x3fff, zbss_ovfl)  : { *(.zbss)  }
    } > zbss
    GROUP : {
        .text (TEXT)         : { *(.text) *(.frame_info) *(.init) *(.fini) }
    } > ram
```

Parent topic: SECTIONS Command on page 21

3.10.11. About the REGISTER Specification for SDA (PowerPC)

Refer to the following information on REGISTER specification for SDA PowerPC.

“Small” data areas can be addressed by a 16-bit offset from a base register, enabling the use of smaller code that uses a “near” 16-bit addressing mode. In general, it is best to use these areas for data that is small and frequently accessed. The compiler by default stores small variables into the “small data area” and small constants into the “small constant area”. See "Small data and small **const** areas" in Appendix A of the *Getting Started* manual for an overview of small data areas.

Using an extra SDA can speed up code, but the trade-off is that the compiler loses the use of one register for each additional SDA. This means that it may be necessary to test to see whether adding an SDA actually increases performance.

The **REGISTER** specification can be used to mark an output section as a “small data area” (SDA) or to change the properties of a standard SDA. The format of *register-spec* is:

REGISTER (*SDA-register-number* [,*offset adjustment*])

SDA-register-number is the register number. *offset adjustment* is a number that can be used to compensate for signed or unsigned offsets to the base register. The default value of *offset adjustment* is 0x7ff0.

The **REGISTER** specification can be used in conjunction with the **#pragma global_register**, **#pragma section**, and **#pragma use_section** directives to reserve a base register for an additional SDA beyond the two standard SDAs (**.sdata** / **.sbss** for small data and **.sdata2** small constants), allocate variables to that SDA, and direct the linker to patch instructions that reference those variables to use the specified base register.

For example, the following specification marks the **.sdata14** section as an SDA with base register 14:

```
    .sdata14 REGISTER(14) : { }
```

The name **.sdata14** was chosen for convenience; the name of the output section does not have to contain the register number.

In a group (see GROUP Definition on page 39), a **REGISTER** specification is in effect until the next **REGISTER** specification, or the end of a group. **REGISTER**(-1) makes a section non-SDA. Sections in a group prior to the first **REGISTER** specification implicitly use the first.

The linker uses the implicit (default) or explicit **REGISTER** specifications to resolve the address of relocatable data using the **sdarx** relocation type. (See F. Object and Executable File Formats for a discussion of relocatable data.) The linker selects the register from:

1. Any **REGISTER** directive explicitly defined for that section.
2. The default SDA registers for **.sdata** / **.sbss**(**r13**) and **.sdata2** (**r2**).

If no SDA register has been defined for a section, a 16-bit absolute address is used.

The base register must be initialized by use of a reference to the symbol_ **SDA**<*re-gister-number*> **_BASE**. For example, in **crt0.s**, use **_SDA14_BASE** to initialize the base register. If **_-SDA**<*register-number*> **_BASE** is not defined by the user, the linker will define it.

(Note: for compatibility with code written for versions of the linker before v4.3, if changing the base register for the standard **.sdata**, the linker symbol name **_SDA_BASE_** must be used instead of **_SDA13_BASE_**. Also, for the same reason, symbol names can have two leading underscore characters instead of one.)

Parent topic: SECTIONS Command on page 21

3.10.12. Using the REGISTER Specification to Create New SDA (PowerPC)

Follow these steps to mark a section as SDA and store data in it.

Procedure

1. Use the **REGISTER**(*n*) clause in a *section-definition* in the linker command file to associate a section of memory with a register.
2. In the file **crt0.s**, load the base address from **SDA***n* **_BASE** into the register, using the same register *n* as was used with the **REGISTER** clause, above.
3. In all C and C++ source files in the build, use **#pragma global_register** to reserve a register (with a dummy variable name). This cannot be a scratch register. See global_register Pragma, p. 37 for a description of this pragma.
4. In all source files where the variables are declared or used, define a special section using the **#pragma section** directive, and use **#pragma use_section** to place variables in the section.

   The **#pragma section** directive must specify that the section be accessed as **near-data** to get the compiler to generate data references using **sdarx** relocation type.

Parent topic: SECTIONS Command on page 21

3.10.13. Writing Assembly Language Routines to Use New SDA (PowerPC)

Assembly language routines may also be written to use the new SDA by writing instructions which use the specified base register.

About This Task

For example, a program will use **r14** for storing the variables **var** and **ivar**, where **ivar** is initialized.

Procedure

1. Add **REGISTER** clauses to the **SECTIONS** command in the linker command file:

```
.data14u REGISTER(14) : {} > ram /* uninitialized data */
.data14  REGISTER(14) : {} > ram /* initialized data   */
```

2. Add the following at the top of every source file to prevent the compiler from using **r14**:

```
    #pragma global_register dummy=r14
```

3. Add the following pragmas to the files that refer to the variables:

```
#pragma section DATA14 ".data14" ".data14u" near-data
#pragma use_section DATA14 ivar, var
```

4. In **crt0.s**, load the base address into the register:

```
addis      r14,r0, _SDA14_BASE_@ha
addi       r14,r14,_SDA14_BASE_@l
```

Results

After compiling and linking, **var** and **ivar** will be referenced by **r14**.

Note: Instructions which refer to small data areas (whether the default area or additional areas created with **REGISTER**) use operands of the form **@sdarx(r0)**. **r0** is a placeholder for the SDA base register — the linker will patch in the correct register at link time, for example in this case, **r14**

Parent topic: SECTIONS Command on page 21

3.10.14. Fill Specification

The *fill-spec* instructs the linker to fill any holes in an output section with a two-byte pattern.

The form of the *fill-spec* is

=*fill-value*

or

=(*fill-value* [**,** *size* [**,** *alignment*]])

A hole is created when an assignment statement is used to advance the location counter "**.**" The linker also creates holes to align input sections according to *alignment*. *size* and *alignment* are in bytes; valid values are 1, 2, and 4.

This example fills the section **.fill_sect** (which will be output into the area of memory defined by **INIT_fill**) with the hex value 0xffaa:

```
    .fill_sect (DATA) :   {
        ...
        ...
    } = 0xFFAA } > INIT_fill
```

Note that you must specify the section type (in this example, **DATA**). The **BSS** and **BTEXT** types are not allowed in this context.

Assuming a memory region named "rom," the following example shows how to fill unused bytes up to the end of the region with a specified value:

```
rom_hole_start = .; /* optional -- shows up in map file */

.dummy (=TEXT) : {    /* create a hole up through the end of rom */
. = ADDR(rom) + SIZEOF(rom);
} = 0x1234    /* hole will be filled with 0x12 0x34 0x12 0x34 .... */

rom_hole_end = .; /* optional -- shows up in map file */
```

Parent topic: SECTIONS Command on page 21

3.10.15. CRC Specification

The *crc-spec* calculates a cyclic redundancy check for a memory region and stores the value in the output section.

The form of the *crc-spec* is

= *algorithm* **(***start-addr* **,** *size* [**,** *start-value*] **)**

where

- *algorithm* is **CRC16** or **CRC32**
- *start-addr* is the starting address of the memory region
- *size* is the size in bytes of the memory region
- *start-value* is an optional initial value for the CRC calculation

CRC16 calculates a CRC-16-IBM value.

CRC32 calculates a CRC-32-IEEE 802.3 value.

Example:

```
.crc16_ram  (DATA) :
{ . += 2; } = CRC16(crc16_start, crc16_end - crc16_start, crc16_int)
> crc16_ram
```

This calculates a CRC-16-IBM value for the memory region beginning at **crc16_start**, and stores the value in the area **crc16_ram**.

Parent topic: SECTIONS Command on page 21

3.10.16. Area Specification

An *area-spec* causes the linker to locate the output section at the next available location in the given area (subject to any **ALIGN** clause).

The form of the *area-spec* is

```
> area-name
```

where *area-name* is defined by an earlier **MEMORY** command (see MEMORY Command on page 20).

For more information, see ALIGN Specification on page 30).

Parent topic: SECTIONS Command on page 21

3.10.17. STORE Statement

The **STORE** statement reserves and initializes memory space.

Its form is:

```
STORE ( expression , size-in-bytes )
```

where *expression* is the value to be stored at the current address, and *size-in-bytes* is the size of the storage area, normally 4 for 32-bit values. Example:

```
_ptr_to_main = .;
STORE(_main, 4)
```

will create a label **_ptr_to_main** that contains the 4-byte pointer to the label **_main**.

Parent topic: SECTIONS Command on page 21

3.10.18. TABLE statement

The **TABLE** statement reserves and initializes memory space.

The syntax of the **TABLE** statement is:

TABLE (*pattern* , *size*) ;

where *pattern* is a wild card string that specifies symbol names to be included in the created table. *size* is the byte size of each storage element. The wild card can contain the special token "**?**" (matching any one character), "*" (matching any string of characters, including the empty string), and "[" (matching any of the characters listed up to the next "]"). Example:

```
__CTORS = .;
table(__STI__*, 4);
```

will create a label **__CTORS** that contains a table of 4 byte pointers to all identifiers starting with **__STI__**. This example is used to declare an array containing pointers to all static initializing functions in C++.

Parent topic: SECTIONS Command on page 21

3.10.19. GROUP Definition

A **SECTIONS** command may contain *group-definitions* as well as *section-definitions*

For more information, see SECTIONS Command on page 21.

A group treats several output sections together and ensures they are located in a continuous memory block in the order given in the *group-definition* . When sections are not in a group, their order is not defined, although it may be dictated implicitly by, for example, *address-spec* clauses.

The full form of a *group-definition* is shown below. For clarity, each clause is written on a separate line and is identified to its right. Although each of the specifications is optional, their order matters and must be as shown below; for example, **COPYTABLE** cannot come before **LOAD**.

下面每个选项是可选的，但顺序不可变

GROUP [*address-value* | **BIND** (*expression*)] [**ALIGN** (*expression*)] [**LOAD** (> *area_name*)] [**COPYTABLE** | **COPYTABLE** [*name*]]	*address-spec align-spec load-spec copytable-spec area-spec*

`:`
 	
`{ section-definition ... }`
`[ >area-name ]`	

The clauses in a **GROUP** are defined above: *address-spec* in Address Specification on page 29, *align-spec* in ALIGN Specification on page 30, *load-spec* in LOAD Specification on page 32, *copytable-spec* in COPYTABLE Specification on page 30, *-section-definition* in Section-Definition on page 22, and *area-spec* in Area Specification on page 38.

Note: The address-value and **BIND** clauses may not be used on a section-definition inside a **GROUP**, only on the **GROUP** itself.

Both a section-definition and a group-definition can end with an area-spec. Usually when defining a group, an area-spec is used only on the group-definition and not on the section-definitions enclosed within it.

Parent topic: SECTIONS Command on page 21

3.11. Assignment Commands

An *assignment* command defines or redefines the value of a symbol. Assignment commands are allowed at the outer-most level of a linker command file, and as items in the *section-contents* of a *section-definition.*

For more information, see Section Contents on page 23.

An assignment command may have either of the following forms:

symbol assign-operator expression ;

create an absolute symbol and assign it the value of *expression*

symbol **@** {*section-name* |*symbol2* }*assign-operator expression* ;

create a symbol in the given section, or the same section as *symbol2*, and assign it the value of *expression*

where:

symbol and *symbol2*: an identifier following the rules of the C language with the addition of "**$**" and "**.**" as valid characters and limited to 1,000 characters.

assign-operator: one of

```
        =     +=     -=     *=     /=
```

The assign "**;**" is required.

When the assignment is inside a *section-definition*, the special symbol "**.**" is allowed on either the left or right and refers to the current location counter.

A “hole” can be created in a section by incrementing the "**.**" symbol. If the *fill-spec* is used on the *section-definition*, the reserved space is filled with the *fill-value*.

Example - create a 100 byte gap in a section:

```
    . += 100;
```

Example - define the beginning of the stack for use by initialization code:

```
    __SP_INIT = ADDR(stack) + SIZEOF(stack);
```

3.12. Examples

Refer to the following examples.

Avoiding Long Command Lines

A simple command file to avoid having to give a long command line when invoking the linker could look as follows:

```
main.o load.o read.o arch.a -m2
```

This means: load files **main.o**, **load.o** and **read.o**, search archive **arch.a**, and generate a detailed memory map.

The output sections for the above, not being defined in the command file itself, and absent -Bd and/or -Bt options on the command line, will be as described for these options (see Linker Basic Options on page 55), and using default addresses for each which are appropriate to the target.

Basic

The command file:

```

MEMORY
{
    mem1 : origin = 0x2000, length = 0x4000
    mem2 : origin = 0x8000, length = 0xa000
}
SECTIONS
{
    .text (TEXT) : {} > mem2
    .data (DATA) : {} > mem1
    .bss  (BSS)  : {} > mem1
}

_start_addr = start;
```

means that all **.text** sections are collected together and positioned in the memory area starting at 8000 hex. The sections **.data** and **.bss** are placed in order in the **mem1** area beginning at 2000 hex. The symbol **_start_addr** is defined to be the same as the address of the symbol **start** from one of the input files.

The input object files for the above linker command file are those given on the command line (and any others extracted from libraries to satisfy unresolved external symbols in those files).

Define a Symbol, Create a “Hole”

The command file

```
SECTIONS
{
    .text (TEXT) : {}
    .data ALIGN(8) (DATA) :
    {
        f1.o ( .data )
        _af1 = .;
        . = . + 2000;    a hole
        *
( .data )
    } = 0x1234
    .bss (BSS) : {}

  }
```

means first load the **.text** sections. Align on 8 and load the **.data** section from the file **f1.o**. Set the symbol **_af1** to the current address. Create a hole in the output section with a size of 2000 decimal bytes. Load the rest of the **.data** sections from the files given on the command line. Fill the hole with the value 0x1234. Load the **.bss** sections thereafter.

Groups

The command file

```
MEMORY
{
    a: org = 0x100a8, len = 0x7ffeff58
}

SECTIONS
{
    .text (TEXT) BIND((0x10000 + HEADERSZ+7) & (~7)) :
    {
        *(.init) *(.text)
    }
    GROUP BIND(NEXT(0x10000) +
        ((ADDR(.text) + SIZEOF(.text)) % 0x2000)) :
    {
        .data (DATA) : {}
         .bss (BSS)  : {}
    }
}
```

means that all input sections called **.init** or **.text** are combined into the output section **.text**. This output section is allocated at the address “0x10000 + size of all headers aligned on 8”.

If **HEADERSZ** is 0xe0, the address becomes 0x100e0.

The sections **.data** and **.bss** are grouped together and put at the next multiple of 0x10000 added to the remainder of the end address of **.text** divided by 0x2000.

If **.text** is 0x23450 bytes long, the values are defined to be:

```
NEXT(0x10000) = 0x40000
ADDR(.text) = 0x100e0
SIZEOF(.text) = 0x23450
(ADDR(.text)+SIZEOF(.text))%0x2000 = 0x01530
address of .data = 0x41530
```

This is a typical default algorithm in a paged system where it is important to align the section addresses on the file-offset in the executable file.

Document With C-Style Comments

The following command file is documented with C-style comments.

```
/*
 * The following section defines two memory areas:
 * one 1 MB RAM area starting at address 0
 * one 1 MB ROM area starting at address 0x1000000
 */
MEMORY
{
    ram: org = 0x0, len = 0x100000
    rom: org = 0x1000000, len = 0x100000
}

  /*
   * The following section defines where to put the
   * different input sections. .text contains all
   * code + optionally strings and constant data, .data
   * contains initialized data, and .bss contains  * uninitialized data.
   */

SECTIONS
{
    /* Allocate code in the ROM area. */

    .text (TEXT) : {} > rom

    /*
     * Allocate data in the RAM area.
     * Initialized data is actually put at the end of the
     * .text section with the LOAD specification.
     */
    GROUP : {
        .data (DATA) LOAD(ADDR(.text)+SIZEOF(.text)) : {}
        .bss (BSS) : {}
    } > ram
}
```

Note the use of the **LOAD** clause to allocate the **.data** section to a physical address in ROM, after the **.text** section, while the logical address (the address used during execution) is in the RAM. The initialized data in **.data** has to be moved from the physical address to the logical address during startup.

Empty Sections

It may be an error to define a section without any input sections. This extended example begins with a sample linker command file extract likely to be faulty, and then discusses some potential workarounds. Recommended solutions are at the end of the example. While some of the workarounds are not recommended, they serve to illustrate a number of principles in linker command file construction.

Consider the following example:

```
SECTIONS
{
    ...
    .stack : {
        stack_start = .;
        stack_end   = stack_start + 0x10000;
    } > ram
    ...
}
```

The above is apparently intended to reserve space for a stack and to define symbols marking its beginning and end.

There are four potential problems:

- The address of the current location, ".", and therefore of **stack_start**, is not well-defined. If there are no input sections named **.stack** in the input files, then **stack_start** will be at the "next" unfilled location in **ram**, or at the beginning of the **ram** memory area if no other commands directing output to **ram** precede the above **.stack** output section definition.
  - However, if **.stack** sections do appear in the input files, these will be automatically included in this **.stack** output section — but whether they will appear before or after the address given to **stack_start** is undefined (the rules are complex and subject to change, so no guarantee of order is made for this poorly constrained case).
  - If **.stack** sections do appear in the input files, the definition of "." and therefore of **.stack_start** can be made well defined by adding an input section specification as follows:

    ```
    .stack ALIGN(4) : {
        stack_start = .;
        *(.stack)
        stack_end   = .;
    } > ram
    ```

- **stack_start** may not be aligned as required. Lacking an *align-spec* as in the case above, the alignment will be 1, which may not be valid if the **.stack** section definition is preceded by a section with, for example, an odd length.

  This problem could be solved by providing an *align-spec*:

  ```
  .stack ALIGN(4) : { ... }
  ```

- The assignment to **stack_end** will as expected define it to be **stack_start** plus 0x10000 bytes, *but this assignment in and of itself does not allocate/reserve memory*. If other section definitions result in object bytes in what is intended to be the stack area, the linker will not warn of the conflict.
  - This problem could be solved by incrementing the current location:

    ```
    stack_start = .;
    . += 0x10000;
    stack_end = .;
    ```

  - Incrementing "." creates a “hole”. The hole will be zero-filled (absent specification of a different constant with -f option (see Linker Basic Options on page 55).
  - A reminder: the current location symbol, ".", may appear only in a **SECTIONS** command, either between section definitions, or within a *section-definition* (Section-Definition on page 22) or a *group-definition* (GROUP Definition on page 39).
- Creating a hole by incrementing "." actually uses space in the output image (which could be more of an issue with larger stack). If the area reserved for the stack is expected to be 0, this unnecessary space in the output image can be eliminated by a **BSS** *type-spec* (Type Specification and KEEP Directive on page 27):

```
.stack (BSS) ALIGN(4) : { ... }
```

Combining all of the above, the following is at least valid and likely to produce an acceptable result if there are no **.stack** sections in input files.

```
SECTIONS
{
    ...
    .stack (BSS) ALIGN(4): {
        stack_start = .;
        . += 0x10000;
        stack_end   = .;
    } > ram
    ...
}
```

However, because of its potential problems as described in this example, this approach is not recommended. A recommended way to define a stack, especially in combination with a heap, is to use **GROUP** definitions to locate sections in the desired order, and then to define a stack and heap from the end of the final **GROUP** (using assignment commands as above). Another way is to define a separate memory area for the heap or stack with the **MEMORY** command. These approaches are combined in the **default.dld** linker command file.

Right and Wrong Ways to Use SIZEOF

有对齐产生的间隙

Adding the size of a section to its address is not a reliable way to calculate the address of the next section to follow because there may be an alignment gap between the sections. For example, the following figure shows incorrect and correct ways to define the physical address in a **LOAD** specification and to define a heap symbol. Incorrect commands in the incorrect method and changes in the correct method are in bold.

Note that the lines for **.sdata2** and **.sbss** are specific to ColdFire, MIPS, PowerPC, RH850, and TriCore; the rest is generic.

Correct and Incorrect Use of SIZEOF:

```
MEMORY                          (Used by both incorrect and correct examples.)
{
    rom1: org = 0x20000, len = 0x10000 /* 3rd 64KB */
    rom2: org = 0x30000, len = 0x10000 /* 4th 64KB */
    ram: org = 0x80000, len = 0x30000 /* 512KB - 703KB */
    stack: org = 0xb0000, len = 0x10000 /* 7043B - 770KB */
}
```

Incorrect LOAD Specification and Symbol Definition Using SIZEOF:

```
SECTIONS
{
    GROUP : {
        .text : { *(.text) *(.init) *(.fini) }
        .ctors ALIGN(4):{ ctordtor.o(.ctors) *(.ctors) }
        .dtors ALIGN(4):{ ctordtor.o(.dtors) *(.dtors) }
        .sdata2 : {}
    } > rom1

    .text2 : { *(.text2) } > rom2
     GROUP : {
        .data LOAD(ADDR(.text2) + SIZEOF(.text2)) : {}
        .sdata LOAD(ADDR(.text2) + SIZEOF(.text2) + SIZEOF(.data )) : {}
        .sbss : {}
        .bss : {}
     } > ram
...
 __HEAP_START = ADDR(.bss ) + SIZEOF(.bss ); (Alignment gap after .bss could make __HEAP_START wro
ng.)
 __HEAP_END = ADDR(ram ) + SIZEOF(ram ); (Memory areas are fixed size; SIZEOF use is correct.)
```

Corrected:

```
SECTIONS
{
    GROUP : {
        .text (CONST) : { *(.text) *(.init) *(.fini) }
        .ctors ALIGN(4):{ ctordtor.o(.ctors) *(.ctors) }
        .dtors ALIGN(4):{ ctordtor.o(.dtors) *(.dtors) }
        .sdata2 (DATA) : {}
     } > rom1

     .text2 (TEXT) : { *(.text2) } > rom2

__DATA_ROM= .; (Define symbol for use in LOAD.)
     } > rom2

     GROUP : {
        .data (DATA) LOAD(__DATA_ROM) : {}
        .sdata (DATA) LOAD(ADDR(.sdata) - ADDR(.data) + __DATA_ROM) : {}
        .sbss (BSS) : {}
        .bss (BSS) : {} }
    > ram
...

__HEAP_END = ADDR(ram ) + SIZEOF(ram ); Memory areas are fixed size;
__SP_INIT =  ADDR(stack ) + SIZEOF(stack ); SIZEOF use is correct.)
__SP_END = ADDR(stack );
```

Copying Code from ROM to RAM Without Copytables

In embedded systems, code and data are typically burned into a ROM-type device, and then initial values for global and static variables are copied to RAM during system startup. The startup code can automatically copy such initial values as described in

Copying Initial Values From ROM to RAM and Initializing .bss, p. 154, which makes reference to the linker **LOAD** specification. (See LOAD Specification on page 32.)

Copying code, not just initial data values, to high speed RAM can increase performance because it can be much faster to access than ROM. This example shows how to modify a simplified version the *versionDir* **/conf/sample.dld** file shipped with the compiler suite to support this. In addition, a new copy_to_ram( ) function is required, and **crt0.s** is modified to call it.

Note: For simplicity, the small data and small constant areas have been removed from this example.

This example assumes an understanding of the startup code and the **LOAD** specification referred to above. (Note that if you want to debug code like this with the Wind River Workbench debugger, you must use the -Xgenerate-paddr=0x2 linker option).

The first part of this discussion describes changes that are made to the linker command file. The following **SECTIONS** directive can be used to locate code physically in ROM but logically in RAM:

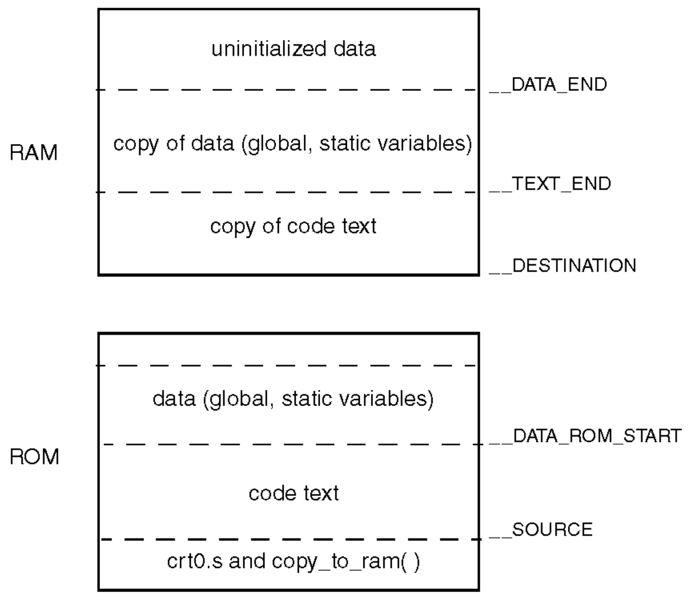
```
SECTIONS
{
    .text (TEXT) LOAD (ROM_ADDRESS) :{}
} > ram
```

The **LOAD** instruction tells the linker where code is to be loaded in ROM at load time — the *physical* address (for example, when the PROM is burned). The area specification (the **> ram** part of the statement) tells the linker where the code will be during execution — the *logical* address. Note that this **SECTIONS** directive does not copy the data from ROM to RAM; it only tells the linker where to resolve references to functions, labels, string constants located with code, and so forth. In this example a user-supplied function called copy_to_ram( ) does the actual copying of code from ROM to RAM during system startup.

If a **LOAD** directive and an area specification such as those shown above are used for the initialization code, that code will not be accessible. This is because the linker would resolve references to the initialization code in the **ram** area, and so the initialization code would never be found. One solution to this “chicken and egg” problem is to refrain from copying the initialization code, **crt0.o** and copy_to_ram( ), to RAM, leaving it in ROM.

Here are the details:

1. Locate initialization code into ROM only, in a section called **.startup**. The startup code consists of **crt0.o** and copy_to_ram( ).
2. Locate the rest of the code, and all global and static variables, physically in ROM but logically in RAM, except for uninitialized variables, which is only placed in RAM.
3. Assign symbols to keep track of important addresses in RAM and ROM. See the diagram below.
   分配符号来跟踪RAM和ROM中的重要地址



The symbols **__SOURCE** (in ROM) and **__DESTINATION** (in RAM) mark the beginning of the code areas (not including the initialization code). **__DATA_ROM_START** marks the beginning of data in ROM, and **__TEXT_END** marks the end of the **.text** section in RAM. **__DATA_END** marks the end of the code and variable sections that are to be copied.

The next two pages show the simplified **sample.dld**, before and after changes are made. Comments have been reduced to improve readability and unnecessary details have been omitted; changes appear in bold text in the second version of **sample.dld**. See Example bubble.dld on page 15 for another example of more complete linker command files.

In the "after" linker command file, note that **__DATA_ROM** and **__DATA_RAM** are made equal to each other in order to prevent **crt0.o** from redundantly copying data. (**crt0.o** copies data from ROM to RAM if those symbols are not equal; see *Copying Initial Values From ROM to RAM and Initializing .bss*.)

Table 1. sample.dld As It Is Distributed

`MEMORY { rom: org=0x0, len=0x100000 ram: org=0x100000, len=0x100000 stack: org=0x300000, len=0x100000 }`	Specify memory layout.
`SECTIONS { GROUP : { .text (TEXT) :{ *(.text) *(.rodata) *(.rdata) *(.frame_info) *(.init) *(.fini)} .ctors ALIGN(4):{ ctordtor.o(.ctors) *(.ctors) } .dtors ALIGN(4):{ ctordtor.o(.dtors) *(.dtors) } } __DATA_ROM = .; } > rom`	The first **GROUP** contains code and constant data, and is allocated in the rom memory area.

`GROUP : { __DATA_RAM = .;`	The second **GROUP** allocates space for initialized and uninitialized data in the **ram** memory area, as directed by **> ram** at the end of the **GROUP**. This is the "logical" location; references to symbols in the **GROUP** are to **ram**.

`.data (DATA) LOAD(__DATA_ROM) : {`
`*(.data) } __DATA_END = .;`
	But the **LOAD** specification on the **.data** output section causes that section to follow be at **__DATA_ROM** in the **GROUP** above in the actual image (the "physical" address).

`__BSS_START = .; .bss (BSS) : {}`
`__BSS_END = .; __HEAP_START= .`
`; } > ram }`
	Allocate uninitialized sections.

Table 2. sample.dld Highlighting Changes Made for Copying from ROM to RAM

`MEMORY { ... } SECTIONS { .startup`
`(TEXT) : { crt0.o(.text)`
`*(.startup) __SOURCE = (. + 3) & ~3; } > rom`
	Create a startup section for initialization code, **crt0.o** and **copy_to_ram( )**, that will only be placed in ROM. **__SOURCE** is the beginning address for the ROM to RAM copy. Make sure **__SOURCE** is aligned.

`GROUP : { __DESTINATION = .; .text (TEXT) LOA`
`D(__SOURCE) : { *(.text) ...`
`}`
	Combine the rest of the code and data into a group located in RAM. Use **LOAD** directives to place all of this group (except uninitialized data) in ROM. **__DESTINATION** is the address in RAM for the ROM-to-RAM copy. Some details (such as **.ctors** and **.dtors**) have been removed.

`__TEXT_EN`
`D = .; __DATA_ROM_START =`
`__SOUR`
`CE + __TEXT_END - __DESTINATION; .data`
`(DATA) LOA`
`D(__DATA_ROM_START) : { *(.data)`
`}`
	__TEXT_END marks the end of code.
	__DATA_ROM_START marks the beginning of data in ROM.

`__DATA_END`
`= .;`
	__DATA_END marks the end of data to be copied.

`__BSS_START = .; .bss (BSS) : {}`
`__BSS_END = .; __HEAP_START = .`
`; } > ram }`
	Allocate uninitialized sections.

`__DATA_ROM = 0; __DATA_RAM = __DATA_ROM ;`	Make **__DATA_ROM** and **__DATA_RAM** equal so initialization code will not copy initial values from ROM to RAM.

A Simple Copy Program from ROM to RAM, Using Linker Symbols

A simple copy program can be used to copy from ROM to RAM, using **__DATA_END** and **__DESTINATION** to calculate the number of bytes to copy.

```
/* These symbols are defined in a linker command file. */

 extern int __SOURCE[], __DESTINATION[], __DATA_END[];
 #pragma section CODE ".startup"
 void copy_to_ram(void) {
     unsigned int  i;
     unsigned int  n;
     /* Calculate length of the region in ints */
     n = __DATA_END - __DESTINATION;

     for (i = 0; i < n; i++) {
         __DESTINATION[i] = __SOURCE[i];     }
     }
```

crt0.s must call copy_to_ram( ). The following is added after the comment “insert other initialization code here,” before calling __init_main( ).

ARM	`bl copy_to_ram`
ColdFire	`jbsr _copy_to_ram`
MIPS	`jal copy_to_ramnop`
PPC	`bl copy_to_ram`
TriCore	`call copy_to_ram`
x86	`call copy_to_ram`
RH850	`jarl _copy_to_ram`

Note: An alternative to using copy_to_ram( ), which is implemented with a **for** loop, would be to call memcpy( ) from **crt0.o**, but then memcpy( ) would remain in ROM, with its slow access.

Using Copy Tables to Initialize data and bss Sections

Assume that you want to have separate copy tables for data and bss sections, so that you can control when you copy data, and when you zero out the bss. The code can run on systems that are capable of a hardware clean of all memory in case of a reset, and in this case you do not need to zero out the bss.

First, you need to define two copy tables in your linker command file, as in this abbreviated code snippet (the ellipses in brackets indicate parts of the linker command file that are not relevant to this example):

```
-Xgenerate-copytables
MEMORY
{
  rom: org = 0x1000000, len = 0x700000
  ram:   org = 0x2000000, len = 0x700000
}

SECTIONS
{
  GROUP : {
    /*Normal ROM contents like .text, and .sdata2*/
    [...]
    .copytable: {} /*The section that holds the copy tables*/
  } > rom

  /*One group for all data. Note the LOAD directive that
    defines where to store the initial values.
    Also note the name of the copy table.*/
    GROUP LOAD(>rom) COPYTABLE [data_copy_tab] : {
        .data (DATA)     : {     *(.data, ".data.*")     }
        .sdata (DATA)     : { *(.sdata, ".sdata.*") }
    } > ram

 /*One group for all bss. Note the name of the copy table.*/
 GROUP COPYTABLE [bss_copy_tab] : {
   .sbss (BSS) : { *(.sbss, ".sbss.*") }
   .bss (BSS) : { *(.bss, ".bss.*") }
   __HEAP_START = .;
 } > ram }

[...]
```

In your startup code, you must do the following to initialize the data and bss section using the copy tables: make the automatically generated symbols known to the compiler; and mark the symbols as weak (otherwise the linker will not auto-generate the symbols). For example:

```
#pragma weak data_copy_tab
#pragma weak data_copy_tab__END__
extern void* data_copy_tab[], *data_copy_tab__END__[];
#pragma weak bss_copy_tab
```

```
#pragma weak bss_copy_tab__END__
extern void* bss_copy_tab[], *bss_copy_tab__END__[];
```

Finally, call __copy_multiple( ) using the generated symbols to either initialize the data variables with their values from ROM, or to zero out the bss sections. In this example, the bss sections are only cleared if required:

```
/*initialize data variables*/
__copy_multiple (&data_copy_tab, &data_copy_tab__END__);

if (!platformZeroedOutTheMemoryAtReset)
  /*Zero out bss if running on a platform that did not do it for us*/
  __copy_multiple (&bss_copy_tab, &bss_copy_tab__END__);
```

Using CLONE and ALIAS for Core-Local RAM

In modern computer systems a CPU (or the individual core of a multi-core CPU) can have local RAM that is usually much faster with respect to access times—but also much smaller—than global RAM or ROM. Local RAM should therefore be used for the code whose execution time is most critical.

Special consideration should, however, be given to code that is loaded into local RAM if it calls library functions. The library code itself is most likely located in global RAM or ROM—due, for example to the size of the libraries, or to allow sharing the library code between multiple cores. This means any call to a library function would reduce, or in the worst case nullify, the execution speed advantages of local RAM. Therefore, whatever function is called from code that is executed from local RAM should also be located in local RAM.

Manually duplicating the code, and renaming function names of objects to be placed into local RAM is tedious and error prone. Using **CLONE** and **ALIAS** linker command language directives allows to achieve duplication and renaming programatically.

For this example, let's assume the following:

- The system has global RAM, global ROM, and core-local RAM.
- Code is executed from global ROM. This code includes library functions in a number of object files, one of which is foo( ). It also includes application code that calls foo( ).
- Application code is executed from local RAM. This code also calls the function foo( ). The object file for this code is **local_main.o**.

The following linker command file copies the function foo( ) into local RAM for use only by the code that runs in local RAM. The code in global ROM still uses the function foo( ) that resides in global ROM.

```
<LinkerCommandFile>

-Xgenerate-copytables

MEMORY
{
  rom: org = 0x100000, len = 0x00010000 /* 64k of ROM */
  ram: org = 0x200000, len = 0x00100000 /* 1M of global RAM */
  local_ram: org = 0x400000, len = 0x00001000 /* 4k of local RAM */

}

SECTIONS
{
   GROUP : {
     /*The output .text section contains ALL code except the code to be
     executed from local RAM.*/
   .text (TEXT) : {
     {!local_main.o}(.text, ".text.*") /* This section also contains foo */
     *(.rdata)
     *(.rodata)
     *(.frame_info)
     *(.eh_frame)
     *(.init)
     *(.fini)
   }
   .ctors ALIGN(4) : { ctordtor.o(.ctors) *(.ctors) }
   .dtors ALIGN(4) : { ctordtor.o(.dtors) *(.dtors) }
   .sdata2 (TEXT) : {}
   .copytable : {}
 } > rom

 /*The RAM contains data, bss, heap, and stack for the non-local
 RAM code.
 */
 GROUP LOAD(>rom) COPYTABLE: {
   .data (DATA) : { {!local_main.o}(.data)}
   .sdata (DATA) : { {!local_main.o}(.sdata) {!local_main.o} }
   .sbss (BSS) : { {!local_main.o}(.sbss)}
   .bss (BSS) : { {!local_main.o}(.bss)}
   __HEAP_START = .;
 } > ram

 /*The local RAM is filled by the startup code.
 The startup code relocates all sections of local_main.o,
 and a clone of the section that defines foo, into local RAM.

 The CLONE directive creates a clone of the section that
 contains foo and renames foo to _foo. Due to the COPYTABLE
 directive, this clone is placed in global ROM, but is
(at runtime) relocated to local RAM. Any references to
 _foo are resolved to its local RAM position.

 The ALIAS directive ensures that any references to foo in
 local RAM are replaced by references to _foo, so that any
 call to foo stay in local RAM.

 NOTE: The code written for the local RAM should, of course,
 change the stack pointer to operate in local RAM. For example
  by using a symbol like __FREE_LOCAL_RAM_START
 */

 GROUP LOAD(>rom) COPYTABLE : {
   .text (TEXT) ALIAS(foo:_foo) : {
       local_main.o(.text)
       CLONE(foo:_foo)
   }
   .data (DATA): { local_main.o(.data) }
   .bss (BSS) : { local_main.o(.bss) }
    __FREE_LOCAL_RAM_START = .;
  } > local_ram


}

__SP_INIT = ADDR(ram)+SIZEOF(ram);
__SP_END = __SP_INIT-0x800; /* Stack is 2KB. */
__HEAP_END = __SP_END; /* Heap contiguous with stack. */
```

4. LINKER OPTIONS

4.1. Linker Basic Options

This section describes the basic options to the Diab linker.

-?

-?X

Show options summary.

Synopsis

-?

-h

--help

Description

Show a synopsis of command-line options.

-?X

-hX

Show a synopsis of -X options.

-@

-@@

Read options from an environment variable or file.

Synopsis

-@*name*

Description

Read command-line options from the environment variable *name* if it exists, else from the file *name*. With an environment variable, separate options with a space. With a file, place one or more options per line, separated by a space.

-@@*name*

Same as -@*name*; also prints all command-line options on standard output.

-@E

-@O

Redirect output.

Synopsis

-@E=*file*

-@E+*file*

Description

Redirect any output to standard error to the given file.

-@O=*file*

-@O+*file*

Redirect any output to standard output to the given file. In both cases, use of + instead of = appends the output to the file.

-a

Allocate memory for common variables when using -r.

Synopsis

-a

Description

Common variables are not normally allocated when an incremental link is requested by the -r option. The -a option forces allocation in this case.

For more information, see COMMON Sections on page 7.

-A

Link files from an archive.

Synopsis

-A *filename*

-A -l*name*

-A -l:*filename*

Description

Link all files from the specified archive. The -A option affects only the argument immediately following it, which can be a filename or -l option. If *filename* or *name* is not an archive, -A has no effect.

Sections can still be dropped with the -Xremove-unused-sections option.

-A1

Same as -A.

-A2

Same as -A, but overrides -Xremove-unused-sections for the specified archives.

-A3

Same as -A2, but also overrides -s and -ss for the specified archives.

The -A commands generate warnings for any duplicate symbols in the archives.

-Bd

-Bt

Set addresses for data and text.

Synopsis

-Bd=*address*

-Bt=*address*

Description

Allocate **.text** and **.data** sections to the given address.

The -Bd and -Bt options provide a simple way to either:

- define where to allocate the sections without having to write a linker command file, or
- change the address of sections specified in a linker command file, dynamically, from a command line

If either -Bd or -Bt is specified, the linker will use the following command specification:

```
SECTIONS {
     GROUP BIND
TEXTBASE : {
     .text (TEXT) : {
          *(.text) *(.rdata) *(.rodata)
          *(.init) *(.fini)
     }
     .sdata2 (TEXT) : {}
  }
  GROUP
BIND DATABASEnvarname{}: {
     .data (DATA) : {}
     .sdata (DATA) : {}
     .sbss (BSS) : {}
     .bss (BSS) : {}
}
```

where **DATABASE** and **TEXTBASE** are replaced by the values given by -Bd=*address* and -Bt=*address*, respectively.

Note: If you use -Bt or -Bd with a linker command file, you must include **TEXTBASE** and/or **DATABASE** in that file; otherwise the results are unpredictable. The default linker file (**default.dld**) does not specify **TEXTBASE** or **DATABASE**. To use -Bt or -Bd without any linker command file, suppress the use of the default linker command file by specifying the -W m option with no name on the dcc or dplus command line.

If the -N option is given, the **.data** section is placed immediately after the **.text** section.

-Bsymbolic

Bind function calls to a shared library.

Synopsis

-Bsymbolic

Description

When creating a shared library, bind function calls, if possible, to functions defined within the shared library. For VxWorks RTP application development.

-Dsymbol=*address*

Define a symbol at an address.

Synopsis

Dsymbol=*address*

Description

Define the specified symbol at the specified address.

-e

Define a default entry point address.

Synopsis

-e *symbol*

Description

symbol is made the default entry address and entered as an undefined symbol in the symbol table. It should be defined by some module.

-f

Specify "fill" value.

Synopsis

-f *value*

-f *value, size*

-f *value, size, alignment*

Description

Fill all "holes" in any output section with a 16-bit value rather than the default value of zero. Optional size and alignment are specified in bytes; the default is 2, 1.

-I *path*

Add *path* to the preprocessor include search path.

Synopsis

-I *path*

Description

Use -I to add paths to the preprocessor's include search path. This search path is used when the preprocessor encounters an **#include** directive. See also:

- -MD, -MU
- -Xpreprocess-lecl

-L

Specify a directory for the -l option search list.

Synopsis

-L *dir*

Description

Add *dir* to the list of directories searched by the linker for libraries or files specified with the -l option. More than one -L option can be given on the command line. Must occur prior to a -l option to be effective for that option. See Also -l.

-l*name*

Specify a library or file to process.

Synopsis

-l*name*

-l:*filename*

Description

The -l option specifies a library with the constructed name **libname.a** to be searched for object modules defining missing symbols.

The -l: option processes the given *filename* (without modification, no path prefix allowed). An object file is linked, an archive is searched as necessary, a text file is taken as a linker command file.

For both forms, the search for the file is performed in the following order:

1. The directories given by -L *dir* options in the order these options are encountered.
2. The directories as given by any -Y L, -Y P, or -Y U options.

Any -L or -Y option must occur prior to all -l options to which it applies.

If no -L or -Y option is present, search a set of directories based on the selected target and environment.

Duplicate symbols:

In certain circumstances the linker will generate a warning if the same symbol name appears in two different libraries.

The linker searches the libraries in the order they are listed. When a symbol is found in a given library the linker stops searching. When resolving additional symbols, the linker may need to search other libraries. If these other libraries contain duplicates of any symbols that have already been resolved, then the linker will issue an error.

For example, assume that object file **main.o** contains references to external symbols **foo** and **bar**. If **libfoo.a** contains the symbol **foo**, while **libfoo_dup.a** contains both **foo** and **bar**, the following will produce a warning:

```
% dld -lfoo -lfoo_dup main.o
dld: warning: Redeclaration of foo
Defined in foo.o(.nlibfoo.a) and bar.o(.nlibfoodup.a)
```

However, if the search order is reversed, then the linker can resolve both **foo** and **bar** from **libfoodup.a**, so it has no need to search **libfoo.a** and does not produce a warning:

```
% dld -lfoo_dup -lfoo main.o
```

If it is necessary to find duplicate symbols, run the dld command twice, reversing the library order the second time.

See also:

- -L
- -Y L, *dir*
- -Y P, *dir*
- -Y U, *dir*

-m

Generate a link map.

Synopsis

-m**[***n* **]**

Description

Generate a link map of the input and output sections, and display on standard output.

The value *n* following the -m in the option name defines the type of map to produce. It is converted to hexadecimal and used as a mask. For example, **-m6** is equivalent to **-m2** plus **-m4**. Undefined bits in the mask are ignored.

-m

List all output sections with their virtual addresses and sizes. Same as -m1.

For each of the output sections that are listed, the map lists the input sections that are part of the output section with their virtual addresses, sizes, and origins. The origin is either an object file, an object file within a library, or a COMMON section (for information about COMMON sections, see see COMMON Sections on page 7). In the case of a file, it shows the path to the object file (as specified on the command line). In the case of an object file within a library, it shows the path to the library (as found by the linker), with the name of the container object file appended to the library file name (in square brackets). In case of a COMMON section, it shows the origin with the label **[COMMON]**. Note that all numbers (sizes and addresses) are in hexadecimal. For example:

```
                output   input
                virtual  section  section
                address  size     file.
text            00020000 00003f34
          .text 00020000 00000038 objects/crt0.o
          .text 00020038 00000000 objects/swap.o
          .text 00020038 00000112 objects/bubble.o
          .text 0002014c 00000074 /foo/diab/5.9.3.0/PPCE/libi.a[exit.o]
          .text 000201c0 000000f4 /foo/diab/5.9.3.0/PPCE/libi.a[init.o]
          .text 000202b4 00000006 /foo/diab/5.9.3.0/PPCE/libi.a[memfile.o]
          .text 000202bc 0000000c /foo/diab/5.9.3.0/PPCE/libi.a[xexit.o]
          .text 000202c8 0000005c /foo/diab/5.9.3.0/PPCE/libi.a[puts.o] [...]
          .bss  00080208 0000072c
          .bss  00080208 00000140 /foo/diab/5.9.3.0/PPCE/libi.a[exit.o]
          .bss  00080348 00000050 /foo/diab/5.9.3.0/PPCE/libi.a[xfiles.o]
          .bss  00080398 00000010 /foo/diab/5.9.3.0/PPCE/libi.a[xsyslock.o]
          .bss  000803a8 00000414 /foo/diab/5.9.3.0/PPCE/libimpl.a[cxa_atexit.o]
          .bss  000807bc 00000178 [COMMON]
```

-m1

Same as -m.

-m2

Generate a more detailed link map of the input and output sections, including symbols and addresses. The -m2 option is a superset of -m1. The symbols and their addresses are listed below the input section that contains them. For legibility, the individual input section are separated by a blank line. For COMMON sections, each module that defines the same symbol is listed. For example:

```
           output input    virtual
          section section  address size file
           .text  00020000 00003f34
           .text  00020000 00000038 objects/crt0.o
          _start  00020004 00000000
            .text 00020038 00000000 objects/swap.o
            .text 00020038 00000112 objects/bubble.o
        get_short 00020038 00000024
             main 0002005c 000000ac
            .text 0002014c 00000074 /foo/diab/5.9.3.0/PPCE/libi.a[exit.o]
    _register_fini 0002014c 00000014
        _Atrealloc 00020160 00000008
             exit 00020168 00000058
            .text 000201c0 000000f4 /foo/diab/5.9.3.0/PPCE/libi.a[init.o]
    init_main_guts 000201c0 000000c4
       __init_main 00020284 00000030
            .text 000202b4 00000006 /foo/diab/5.9.3.0/PPCE/libi.a[memfile.o]
            .text 000202bc 0000000c /foo/diab/5.9.3.0/PPCE/libi.a[xexit.o]
            _Exit 000202bc 0000000c
            .text 000202c8 0000005c /foo/diab/5.9.3.0/PPCE/libi.a[puts.o]
             puts 000202c8 0000005c
             .bss 00080208 0000072c
             .bss 00080208 00000140 /foo/diab/5.9.3.0/PPCE/libi.a[exit.o]
          _Atdata 00080208 00000140
             .bss 00080348 00000050 /foo/diab/5.9.3.0/PPCE/libi.a[xfiles.o]
             ebuf 00080348 00000050
             .bss 00080398 00000010 /foo/diab/5.9.3.0/PPCE/libi.a[xsyslock.o]
              mtx 00080398 00000010
             .bss 000803a8 00000414 /foo/diab/5.9.3.0/PPCE/libimpl.a[cxa_atexit.o]
            stack 000803a8 00000414
             .bss 000807bc 00000178 [COMMON]
          __fname 000807c0 0000001c memfile.o(/foo/diab/5.9.3.0/PPCE/libi.a)
  __std_file_table 000807dc 00000054 stdfn.o(/foo/diab/5.9.3.0/PPCE/libiold.a)
         __sig_ar 00080830 00000104 signal.o(/foo/diab/5.9.3.0/PPCE/libram.a)
```

-m4

Generate a link map with a cross reference table.

File names for a symbol in a cross-reference table are listed in reverse linking order; that is, the last file in the list is pulled in first. (However, if a file defines the symbol, but does not reference the symbol, it will be listed first, before all references.) This is useful when a symbol is referenced by multiple files, and you want to know which symbol reference caused which **.o** file to be pulled in by the linker. For each symbol, the output and input sections are listed, as well as the object files that reference the symbol. The object file that defines the symbol is identified by an asterisk. For example:

```
Symbol    Output   Input      Referenced
          Section  Section    (* - Defined)
strlen    .text    .text      com_fl_pr.o(/foo/diab/5.9.3.0/PPCEN/libcfpold.a)
                              com_print.o(/foo/diab/5.9.3.0/PPCE/libiold.a)
                              * strlen.o(/foo/diab/5.9.3.0/PPCE/libi.a)
                              fputs.o(/foo/diab/5.9.3.0/PPCE/libi.a)
```

-m8

Print library dependency information. The dependencies are listed with the paths to the libraries. For example:

```
Library Dependencies
/foo/diab/5.9.3.0/PPCE/libi.a
/foo/diab/5.9.3.0/PPCEN/libcfpold.a
/foo/diab/5.9.3.0/PPCE/libimpl.a
/foo/diab/5.9.3.0/PPCE/windiss/libwindiss.a
/foo/diab/5.9.3.0/PPCE/libram.a
/foo/diab/5.9.3.0/PPCE/libiold.a
```

-m16

Generate a link map showing the RAM usage for functions and symbols.

The RAM usage information consists of three main parts:

- An overview, showing the memory regions as defined by the MEMORY section of the Linker Command File, their usage (absolute and relative), and the accumulated hole bytes (caused, for example, by alignment requirements). In addition, the linker reports the number of bytes located outside of MEMORY regions (caused, for example, by fixed address sections).
- A list of the amount of memory region bytes used by each object.
- A list of all symbols and their respective memory usage.

For example:

```
Address Size    Used       Holes    Name
==================================================
0x20000 0x10000 16203(24%) 0x11     rom1
0x30000 0x10000 168( 0%)   0        rom2
0x80000 0x30000 2356( 1%)  0x178    ram
0xb0000 0x10000 0( 0%)     0        stack
2 bytes used external to MEMORY regions
objects/bubble.o:
0x112 rom1
0x7c  rom2
0x2c  ram
0x2 external to MEMORY regions
objects/crt0.o:
0x38 rom1
[...]
Memory usage by Symbol
=====================
input_count 0x000002
__init_main 0x000030
get_short   0x000024
main        0x0000ac
array       0x000028
```

The memory analysis provided by **-m16** only takes sections that are allocated into account. In order to ensure that the analysis correctly accounts for all sections that contribute to memory consumption, make sure that they are of an allocatable type (TEXT, TEXT_VLE, DATA, BSS, CONST or BTEXT). Example: If you have a section for your stack region that would create a “hole” in RAM, then you must define it as an allocatable type to ensure that it is counted. Otherwise, it will not be included in the memory usage summary.

Here the **.stack** section is defined as DATA to ensure that it is counted:

```
GROUP :
{
      ...
      .stack (DATA):
      {
            __SP_END = .;
            . +=0x1000;
            __SP_INIT = .;
      }
      ...
} > RAM;
```

-m32

Use with -m2 (as -m34) to display the same information as -m2, but without the origins for symbols within a COMMON section. For example:

```
.bss                000807bc 00000178 [COMMON]
__fname             000807c0 0000001c
__std_file_table    000807dc 00000054
__sig_ar 00080830 00000104
```

-MD*macro*

-MU*macro*

Define or undefine a preprocessor macro.

Synopsis

-MD *macro* **[**=*value* **]**

-MU*macro*

Description

Use -MD to define a macro (with an optional *value*) at the command line that can be evaluated in preprocessed linker command files. Use -MU to undefine a macro at the command line.

Note: In order to allow -MD and -MU to take effect, they must appear before the linker command file on the command line.

See also:

- -I *dir*
- -Xpreprocess-lecl

-N

Allocate a **.data** section immediately after a **.text** section.

Synopsis

-N

Description

This option is used in conjunction with options -Bd and -Bt. See -Bd, -Bt.

-o *file*

Change the default output file.

Synopsis

-o *file*

Description

Use *file* as the name of the linked object file instead of the default filename **a.out**.

-r

Perform an incremental link.

Synopsis

-r**[***n* **]**

Description

The -r options are required only for incremental linking, not when producing an ordinary absolute executable. For ARM, MIPS, and PowerPC, some -r options will also generate branch islands. See Branch Islands on page 9.

-r

-r1

The linked output file will still contain relocation entries so that the file can be re-input to the linker. The output file will not be executable, and no unresolved reference complaints will be reported.

-r2

Link the program as usual, but create relocation tables to make it possible for an intelligent loader to relocate the program to another address. Absent other options, a reference to an unresolved symbol is an error.

-r3

Equivalent to the -r2 option except that unresolved symbols are not treated as errors.

-r4

Link for the VxWorks loader. COMDAT sections are merged and converted to normal sections.

-r5

Equivalent to the -r option except that branch islands are generated and COMDAT sections are merged and converted to normal sections.

-rpath

Search for shared libraries on the specified path.

Synopsis

-rpath *path*

Description

Search for shared libraries on the specified *path* , a semicolon-separated list of directories. (If no search path is specified, the linker looks in the directory where the executable resides.) For VxWorks RTP application development.

-R

Rename symbols.

Synopsis

-R *symbol1* = *symbol2*

Description

Rename symbols in the linker output file symbol table. The order of the symbol names is not significant;

```
$ dld -R <symbol1>=<symbol2>
```

does the same thing as

```
$ dld -R <symbol2>=<symbol1>
```

If both symbols exist, both are renamed: *symbol1* becomes *symbol2* and *symbol2* becomes *symbol1* .

-s

-ss

Do not output symbol table and line number entries.

Synopsis

-s

Description

Do not output symbol table and line number entries to the output file.

-ss

Same as -s, plus also suppresses all **.comment** sections in the output file.

-soname

Specify a name for a shared library.

Synopsis

-soname= *libraryName*

Description

Use *libraryName* as the name of the shared object containing compiled library code. For VxWorks RTP application development.

-t

Select the target processor.

Synopsis

-t*tof*:*environ*

Description

Select the target processor with *t*, the object format with *o*, the floating point support with *f*, and libraries suitable for the target environment with *environ*. To determine the proper values, run dctrl -t to interactively display all valid combinations. See About Target Configuration.

-u *symbol*

Define a symbol.

Synopsis

-u *symbol*

Description

Add *symbol* to the symbol table as an undefined symbol. This can be a way to force loading of modules from an archive.

-V

Print version number.

Synopsis

-V

Description

Print the version of the linker.

-X

Do not output some symbols.

Synopsis

-X

Description

Do not output symbols starting with **@L** and **.L** in the generated symbol table. These symbols are temporaries generated by the compiler.

-Y

Specify search directories for the -l option.

Synopsis

-Y L, *dir*

-Y P, *dir*

-Y U, *dir*

Description

-Y L, *dir*

Use *dir* as the first default directory to search for libraries or files specified with the -l option.

-Y P, *dir*

dir is a colon-separated list of directories. Search each of the directories in the list for libraries or files specified with the -l option.

-Y U, *dir*

Use *dir* as the second default directory to search for libraries or files specified with the -l option.

Notes:

- These options must occur prior to all -l options to which they are to apply.
- The dcc and dplus programs (but not dld itself) generate a -Y P option suitable for the selected target and environment. Unless you are replacing the libraries, you should not normally use this option. Use the -L option to specify libraries to be searched before the Wind River libraries. (See -Xl .)
- If no -Y or -l options are present on the dld command line, the linker will automatically search the directories associated with the default target. See About Target Configuration.
- If a -Y option is used, -Y P is recommended. The older -Y L and -Y U options are provided for compatibility. Use of -Y P together with -Y L or -Y U is undefined.

4.2. Linker -X Options

This section describes the -X options to the Diab linker.

-Xadjust-lma

Adjust a physical address (LMA) by an offset from a virtual address (VMA).

Synopsis

-Xadjust-lma=*value*

Description

To support building VxWorks images where the physical address (LMA) should be a fixed offset from the virtual address (VMA), the Diab linker (dld) supports -Xadjust-lma.

When -Xadjust-lma is set, it implies the use of -Xemulate-gnu-vma-lma, which causes:

- the section header address to be the virtual address rather than the load or physical address;
- the program header physical address (**paddr**) to be output;
- the program header physical address to be the load address (and not the virtual address) if the physical address is not explicitly set using AT( ).

-Xadjust-lma adjusts the physical address that appears in the program headers by the specified *value* ; for example:

```
-Xadjust-lma=0x1000000
```

See also -Xemulate-gnu-vma-lma.

-Xarm-be8

Create an ARM BE-8 executable.

Synopsis

-Xarm-be8

Description

Generate a BE-8 format executable. There is no need to make any changes to compiler or assembler options.

-Xassociate-headers

Generate program headers.

Synopsis

-Xassociate-headers

Description

Generates special symbols to map sections to program headers in ELF files. (For compressed output format only.)

-Xbind-lazy

Use late binding for shared libraries.

Synopsis

-Xbind-lazy

Description

Bind each shared-library function the first time it is called. (By default, binding occurs when the module is loaded.) For VxWorks RTP application development.

-Xbranch-islands

Enable/disable branch island generation.

Synopsis

-Xbranch-islands

-Xbranch-islands-off

Description

(ARM, MIPS, PowerPC, TriCore) Enable or disable generation of branch islands. The default is to generate branch islands where necessary (-Xbranch-islands). See Branch Islands on page 9.

-Xcheck-input-patterns

Check input patterns.

Synopsis

-Xcheck-input-patterns

Description

Check that every input section pattern in the linker command file matches at least one input section. Emit a warning if an unmatched pattern is found.

-Xcheck-input-patterns=2

Same as -Xcheck-input-patterns, but emit a message of severity level “information” instead of “warning." (For use with -Xstop-on-warning.)

-Xcheck-overlapping

Check for overlapping output sections.

Synopsis

-Xcheck-overlapping

Description

Check for overlapping output sections and sections that wrap around the 32-bit address boundary.

-Xcode-factor-diagnostics

Generate diagnostics for -Xcode-factor.

Synopsis

-Xcode-factor-diagnostics

Description

Generate diagnostic output for modules compiled with the -Xcode-factor option. See Also -Xcode-factor.

-Xcoff

Use COFF format for output file.

Synopsis

-Xcoff

Description

This is the default if all input files are in COFF format.

See also -Xelf.

-Xcombine-readonly-sections

Reduce code footprint by combining identical read-only sections.

Synopsis

-Xcombine-readonly-sections**[=***n* **]**

Description

This option combines identical sections to reduce code size. Identical code is determined by size, checksum, relocations, and sections (**.text**, **.data**, **.rdata**, etc.). This option should be used on the dcc/dplus command line with -W1:

```
% dcc <other flags> -Wl,-Xcombine-readonly-sections <file> ...
```

Two combined sections should not be modified or patched during runtime unless the patch applies to the sections before they are combined.

For C code, -Xcombine-readonly-sections works best when the application is compiled with -Xsection-split=3, because this provides more opportunities for sharing.

For C++, the convenience alias -Xcombine-comdats (for -Xcombine-readonly-sections=0x12) is provided. It simply looks for identical COMDAT sections. COMDAT sections are used to hold code and data associated with templates and inline functions.

The mask bits are as follows:

0x01

Merge normal C sections (not C++ COMDAT sections).

0x02

Merge C++ COMDAT sections (used for templates, inline functions, and so on).

0x04

Allow merging with sections that appear in libraries.

0x08

Print cross reference table when used with map option -m4.

0x10

Check section contents. Without this option, duplicate sections are detected based on matching CRC and size. It is recommended that this option always be used for production builds. For development builds, if link time is a factor, disabling this bit may speed up linking.

0x20

Print a short summary of what the optimization has accomplished. For example:

```
Combine Sections: 336 duplicate sections found, 7.59K saved **
```

If specified without a value, -Xcombine-readonly-sections defaults to 0x19 (merge normal C sections, check section contents, print cross reference table).

-Xcombine-comdats

Merge duplicate C++ code or constant data, including string literals, duplicate template functions and so on. This option is a convenience alias for -Xcombine-readonly-sections=0x12.

See also -Xsection-split.

-Xcommon-align

Align common symbols.

Synopsis

-Xcommon-align=*n*

Description

Align each common symbol on an *n*-byte boundary if and only if, no alignment is specified for the symbol in the object file. The default value is 8. COFF only (ELF aligns each symbol individually).

-Xcompress-debug-info

Compress debugging information.

Synopsis

-Xcompress-debug-info

Description

Compress debugging information sections in executable files by finding and eliminating redundant debugging information and then generating new debugging information sections. Compression of debugging information requires extra memory and CPU resources at link time. By default, -Xcompress-debug-info is turned off.

-Xcompress-symbols

Remove multiple structure definitions.

Synopsis

-Xcompress-symbols

-Xcompress-symbols-off

Description

Remove multiple definitions of structures in a COFF symbol table. This can dramatically reduce the symbol table size when many object files containing the same structure definitions are linked together.

-Xcopytable-section-name

Specify the name of a section containing copy tables.

Synopsis

-Xcopytable-section-name=*name*

Description

Synopsis Description By default, all copy tables (created with the COPYTABLE linker directive) are put in a section called "**.copytable**." This option allows you to use name as the name of that section instead. See -Xgenerate-copytables and COPYTABLE Specification on page 30.

-Xdisable-SDA

Disable SDA optimization for a given list of SDA classes.

Synopsis

-Xdisable-SDA=section-class-list section-class-list=*section-class* **[, section-class-list]**

Description

As part of whole-program optimization, the linker performs SDA optimization on all available Small Data Areas (SDAs). This option disables SDA optimization for all the SDA classes listed as the argument of this option. Available classes are **ZDATA**, **ZCONST**, **SDATA**, and **SCONST**. For information about section classes and about SDA optimization, see the *Wind River Diab Compiler User's Guide*.

-Xdont-die

Force the linker to continue after errors.

Synopsis

-Xdont-die

Description

Force the linker to continue after errors which would normally halt the link. For example, issue warnings rather than errors for undefined symbols and out-of-range symbols. When the linker is forced to continue it produces reasonable output and returns error code 2 to the parent process. By default, the make utility stops on such errors; if you want it to continue you must handle this error code in the makefile explicitly.

-Xdont-link

Do not create output file.

Synopsis

-Xdont-link

Description

Do not create a linker output file. Useful when the linker is started only to create a memory map file.

-Xdump-metadata

Display metadata found in any linked object files.

Synopsis

-Xdump-metadata=*metadata-report-output-spec*

Description

By default, output is directed to standard output. To direct output to a file, use -Xdump-metadata-output-file.

The *metadata-report-output-spec* string uses the following syntax:

metadata-report-output-spec:=*format* [*key-list* **]**

where *format* has the syntax

format:=c | v | x

where

- **c** represents a comma-separated table with a header line, one line per object file.
- **v** represents vertical output. That means each key-value pair appears on a line of its own. Object files are separated by a blank line.
- **x** represents XML output.

key-list has the syntax:

key-list **:=***SEPkey-spec* [*key-list* **]**

where *SEP* is a separator, one of a colon (:), semicolon (;), or comma (,).

key-spec is either a *key* or a *renamed-key*, where

renamed-key **:=***new-name* **=***key*

where

- *new-name* is a name that should appear in the report instead of the real key name.

- Valid *key*s may be taken from this list of predefined key-list keys. (The report contains only the keys from the key-list. An empty key list results in an empty report.)

  sourceFile

  Input file as provided to ctoa, etoa, or das.

  ObjectFile

  Output file generated by the respective tool.

  ctoaVersion

  The tool version.

  etoaVersion

  The tool version.

  lloptVersion

  The tool version.

  reorderVersion

  The tool version.

  dasVersion

  The tool version.

  ctoaBuildLabel

  The tool build label.

  etoaBuildLabel

  The tool build label.

  lloptBuildLabel

  The tool build label.

  reorderBuildLabel

  The tool build label.

  dasBuildLabel

  The tool build label.

  ctoaOptions

  The options passed to the tool.

  etoaOptions

  The options passed to the tool.

  dasOptions

  The options passed to the tool.

  compilerVersion

  Convenience “link” to either the ctoa or etoa fields (whichever tool was used).

  compilerBuildLabel

  Convenience “link” to either the ctoa or etoa fields (whichever tool was used).

  compilerOptions

  Convenience “link” to either the ctoa or etoa fields (whichever tool was used).

  targetFlag

  Complete target flag (for example, **-tARM64EN:cross**).

targetEnv

The used environment (for example, **cross**).

target

The target (for example, **ARM64**).

See also -Xdump-metadata-output-file.

-Xdump-metadata-output-file

Specify output file for -Xdump-metadata reports.

Synopsis

-Xdump-metadata-output-file=*filename*

Description

Direct the report generated by -Xdump-metadata to the file *filename* instead of standard output.

See also -Xdump-metadata.

-Xdynamic

Use shared libraries.

Synopsis

-Xdynamic

Description

Link against shared libraries (**.so** files). For VxWorks RTP application development.

-Xelf

Use ELF format for output file.

Synopsis

-Xelf Description

Description

This is the default if any input file is in ELF format. See also -Xcoff.

-Xelf-rela

ELF format relocation information.

Synopsis

-Xelf-rela

Description

Use RELA relocation information format for ELF output. This is the default.

-Xelf-rela-off

-Xelf-rela=0

Use REL relocation information format for ELF output.

-Xemulate-gnu-vma-lma

Generate VMA and LMA fields in ELF files like the GNU tool chain.

Synopsis

-Xemulate-gnu-vma-lma

Description

Generate ELF section address and program address fields using the same semantics as the GNU tools. The section address and program header **p_vaddr** fields will be set to the "VMA" (link/runtime address) and the program header **p_paddr** field will be set to the "LMA" (load address, or address specified with the **AT()** linker command file keyword). Use this option when downstream tools such as FLASH loaders, or simulators, expect GNU style input.

See also -Xadjust-lma and -Xgenerate-paddr.

-Xexclude-libs

Do not export symbols from specified libraries.

Synopsis

-Xexclude-libs=list

Description

Do not automatically export symbols from the libraries specified in the comma-delimited list. (Use the same library names, prefixed with “l”, that you would use with the -l option.)

Example: -Xexclude-libs=lc,lm

For VxWorks RTP application development.

-Xexclude-symbols

Do not export specified symbols.

Synopsis

-Xexclude-symbols=*list*

Description

Do not export the symbols specified in the comma-delimited list when creating a shared library. For VxWorks RTP application development. Example:

```
% dld -Xexclude-symbols=function1,function2
```

-Xexpl-instantiations

Write explicit instantiations file.

Synopsis

-Xexpl-instantiations

Description

This option is deprecated.

Cause the linker to write the source lines of an explicit instantiations file to **stdout**. To minimize space taken by template classes, the output from –Xexpl-instantiations can be used to create an explicit instantiations file (necessary header files must still be added). See the discussion of templates in the *Wind River Diab Compiler User’s Guide*.

-Xextern-in-place

Generate executable for conversion to IEEE-695.

Synopsis

-Xextern-in-place

Description

Generate a modified COFF file suitable for conversion to IEEE-695 format using the ddump -I command. This linker option is required if the ddump -I is to be used. Further, when this option is used, the resulting file contains modified COFF, and is not suitable for any use other than input to ddump -I. If standard COFF is also required, link twice with and without this option.

-Xgenerate-copytables

Enable the use of copy tables.

Synopsis

-Xgenerate-copytables[=0]

Description

This option directs the linker to evaluate and use COPYTABLE directives found in linker command files. The mere presence of a COPYTABLE directive in a linker command file does not cause a copy table to be generated; this option must also be specified. This option is off by default.

If a linker command file contains a COPYTABLE directive but this option has not been specified, the COPYTABLE directive will be ignored and linking will proceed normally. To turn this option off, set it to zero.

See also -Xcopytable-section-name and the discussion of the COPYTABLE specification in COPYTABLE Specification on page 30.

-Xgenerate-paddr

Store segment address in program header.

Synopsis

-Xgenerate-paddr=*n*

Description

Store the address of each segment in the **p_paddr** field of the corresponding entry in the program header table. Without this option, the **p_paddr** value will be 0.

The number *n* provided as an argument to this option is a bit mask that can be used for the following values:

0x1

Generate the physical address in the program header table entry. 0x1 is the default for -Xgenerate-paddr and does not have to be set.

0x2

Set the section header's **sh_addr** field to the RAM address.

The 0x2 value makes it possible to debug an application with the Wind River Workbench debugger when it is copied from ROM to RAM during execution. Note, however, that you cannot use ddump with an application that is built using the 0x2 value. For more information in this regard, see the Wind River Workbench documentation.

If you need the **p_paddr** field of the program header to be set to the load address rather than the link/runtime address, use -Xemulate-gnu-vma-lma.

See also -Xemulate-gnu-vma-lma.

-Xignore-empty-sections

Disable warning about empty sections.

Synopsis

-Xignore-empty-sections

Description

Disable the warning message that is given when there is no mapping rule for assigning an input section to an output section. This option disables the warning only when the section is empty (that is, its size is zero).

-Xignore-extensible-sda-warnings

Ignore warnings related to mismatch in extensible SDA.

Synopsis

-Xignore-extensible-sda-warnings

Description

This linker switch is related to the compiler options -Xsmall-const-registers and -Xsmall-data-registers. Normally, when -Xsmall-[data/const]-registers is specified, the linker does sanity checking to ensure that the entire configuration is self-consistent and gives warnings if it is not. This option disables such warnings.

See also -Xsmall-const-registers and -Xsmall-data-registers.

-Xignore-small-data-reg-warnings

Ignore warnings related to option -Xsmall-data-registers.

Synopsis

-Xignore-small-data-reg-warnings

Description

When option -Xsmall-data-registers is set, warnings related to this option will be suppressed.

See also -Xsmall-data-registers.

-Xlinker-show-progress

Show status diagnostics while linking.

Synopsis

-Xlinker-show-progress

Description

When the linker option -Xlinker-show-progress is set, linker prints status information on **stdout**.

-Xmax-long-branch

Limit long branch island generation.

Synopsis

-Xmax-long-branch=*n*

-Xmax-long-branch=off

Description

(PowerPC, TriCore) Set the maximum branch delta for a 26-bit branch; that is, the maximum branch distance permitted before a branch island is used . If this option is not specified (equivalent to -Xmax-long-branch-off or -Xmax-long-branch=0), a default maximum branch delta of 225 -1 (that is, +/- 32MB - 1) is assumed; the branch offset is limited by instruction format only. See Branch Islands on page 9.

-Xmax-short-branch

Limit short branch island generation.

Synopsis

-Xmax-short-branch=*n*

-Xmax-short-branch=0

Description

(ARM, MIPS, PowerPC, TriCore) Set the maximum branch delta for a 16-bit branch, that is, the maximum branch distance permitted before a branch island is used. If this option is not specified (equivalent to -Xmax-short-branch-off or -Xmax-short-branch=0), a default maximum branch delta of 215 -1 (that is, +/- 32KB - 1) is assumed; the branch offset is limited by instruction format only. See Branch Islands on page 9.

-Xmixed-compression

Allow mixed compression.

Synopsis

-Xmixed-compression

Description

Allow compressed and uncompressed code to be mixed. (For compressed output format only.)

-Xmixed-compression-off

-Xmixed-compression=0

Disable generation of compression switches. This is the default.

-Xold-align

Do not align output section.

Synopsis

-Xold-align

Description

Do not align output sections. Without this option (the default), each output section is given the alignment of the input section having the largest alignment. For MIPS, PowerPC, and x86, output sections must be aligned to support position-independent code. With this option, output sections are not aligned, and each output section begins immediately after the previous output section. (In this later case, input sections will still be aligned per their requirements, potentially leaving a gap from the start of the output section to the start of the first input section within it.)

-Xoptimized-load

Pad input sections to match an existing executable file.

Synopsis

-Xoptimized-load=*n*

-Xoptimized-load

Description

Minimize the difference between the already existing executable file (if any) and the new file by padding input sections. *n* specifies how much relative space the linker can use for padding, where 0 means no padding and 100 is the default. The larger the value of *n* , the more similar the images are likely to be. The linker saves the old executable file with the **.old** extension and generate a diff file with the **.blk** extension.

-Xoverflow-greedy

Gather input sections greedily for OVERFLOW specification.

Synopsis

-Xoverflow-greedy

Description

The linker tries to gather input sections into the output section according to their order. If an input section is too large to be accommodated, it’s moved to the tail of the overflow section and then the linker continues to try to gather the rest of input sections greedily. See the OVERFLOW Specification on page 34.

-Xpic-only

Make branch islands position-independent.

Synopsis

-Xpic-only

Description

(MIPS, PowerPC, TriCore) Generate only position-independent branch islands. The default is off, generating branch islands which are not position-independent. See Branch Islands on page 9.

-Xppce200z4-erratum-010385

Disables conversion of switch to look-up array.

Synopsis

-Xppce200z4-erratum-010385

Description

When this option is enabled, the linker will produce an error if code patterns are detected that might trigger NXP erratum e10385.

If code that might trigger the erratum is detected, the linker will produce an error like the following:

```
dld: Checking for e200z4 branch displacement issue:
dld: 0x00004002: e_bc 0,2,0x4002+0x3ffe
dld: error: 1 suspicious branch instruction
```

The error message gives the address of a potentially dangerous branch instruction that is 2 bytes from the start of a 16k page boundary and that has a branch displacement value equal to 0x3ffe. To fix the condition, insert padding code (for example a **nop**) into the affected function, or slightly reorder the code. To just give a warning but continue to link, use -Xppce200z4-erratum-010385=2.

-Xprefix-underscore...

Add leading underscore ("_") to all symbols.

Synopsis

-Xprefix-underscore

Description

Add a leading underscore ("_") to all symbols in the files specified after this command. Use -Xprefix-underscore=0 to turn off this feature. The default is off.

For PowerPC:

-Xprefix-underscore-coff

-Xprefix-underscore-elf

These options add underscores to symbols coming from COFF or ELF input files respectively. This can be helpful with third-party tools that use different naming schemes.

-Xpreprocess-lecl

Perform pre-processing on linker scripts.

Synopsis

-Xpreprocess-lecl

Description

This option allows you to use C pre-processor features in linker scripts. To define/undefined command line macros use -MD and -MU. Include paths may be specified using -I.

See also:

- -MD -MU
- -I *dir*

-Xreloc-bug (For ARM, PowerPC, RH850 only)

Use workaround for ELF relocation bug.

Synopsis

-Xreloc-bug

Description

Enables a workaround for a bug in the ELF relocation information generated by some compilers. Do not use with object files created by the Wind River Diab Compiler.

-Xremove-unused-sections

Remove unused sections.

Synopsis

-Xremove-unused-sections

-Xremove-unused-sections-off

Description

Remove all unused sections. By default the linker keeps unused sections. The KEEP directive, can be used in a linker command file to mark a specific section as used. This may be necessary for sections containing only interrupt/exception vector tables or boot code. (For information about the KEEP directive, see Type Specification and KEEP Directive on page 27.)

A section is used if it:

- Is associated with a KEEP directive in the linker command file.
- Is referred to by another used section.
- Has a program entry symbol—that is, a symbol defined with the -e option or one of **__start, _start, start, __START, _START, _main**, or **main** (order reflects priority).
- Is not referenced by any section and has a name that starts with **.fini, .frame_info, .init, .j_class_- table**, or **.line**.
- Defines a symbol used in an expression in the linker command file.
- Defines a symbol specified with the -u option.

Note: This option is especially useful in combination with -Xsection-split to reduce code and/or data size. When -Xremove-unused-sections is used and code splitting is enabled with -Xsection-split bit 1, each function in a module will generate a separate CODE section, and unused functions will be removed. Similarly when data splitting is enabled with -Xsection-split bit 2, each variable will generate a separate DATA section, and unused variables/constants will be removed.

See also:

- -e
- -u *symbol*
- -Xsection-split
- -Xunused-sections...

-Xrescan-libraries

Re-scan libraries.

Synopsis

-Xrescan-libraries

-Xrescan-libraries-off

Description

Request that the linker re-scan libraries to satisfy undefined externals. This is the default. It solves the ordering problem which occurs when one library uses symbols in another and vice-versa. Use -Xrescan-libraries-off to force the linker to scan libraries and object files in precisely the order given on the command line.

-Xrescan-restart

Re-scan libraries restart.

Synopsis

-Xrescan-restart

-Xrescan-restart-off

Description

If -Xrescan-libraries is on, when more than one library is presented to the linker, force the linker to rescan the libraries from first to last in order for each undefined symbol. This is the default. Use -Xrescan-restart-off with -Xrescan-libraries to cause the linker, after finding symbols in one library, to continue with the next library for the rest of the undefined symbols.

-Xsda-base-offset

Specifies the offset between the small data/const area base and the start of the area.

Synopsis

-Xsda-base-offset

Description

If not specified, the offset depends on the -t flag and defaults to 0x7ff0 for targets that imply 16-bit addressing, and 0x3ffff0 for targets that imply 23-bit addressing. Normally it is not necessary to manually override this value. Note that this option affects all small data-like areas, not just **.sdata**.

-Xsection-align

Align sections.

Synopsis

-Xsection-align=*n*

Description

Force COFF input sections to have an alignment of *n* instead of the default 8. (Ignored for ELF output.)

-Xshared

Build shared libraries.

Synopsis

-Xshared

Description

Build shared libraries (rather than stand-alone executables). For VxWorks RTP application development.

-Xsort-frame-info

Sort the **.frame_info** section.

Synopsis

-Xsort-frame-info

-Xsort-frame-info-off

Description

To enable sorting of the **.frame_info** section, use -Xsort-frame-info. By default, sorting is disabled (-Xsort-frame-info-off).

-Xsort-section-by-alignment

Sort input sections by alignment size within output sections.

Synopsis

-Xsort-section-by-alignment=*pattern*,...

Description

Sort input sections that match at least one of the *pattern*s in the list of patterns (*pattern*,. . .), in ascending order.

-Xsort-section-by-alignment-desc

Sort input sections by alignment within output sections.

Synopsis

-Xsort-section-by-alignment-desc=*pattern*,...

Description

Sort input sections that match at least one of the *pattern*s in the list of patterns (*pattern*,. . .), in descending order.

-Xsort-section-by-name

Sort input sections by name within output sections.

Synopsis

-Xsort-section-by-name=*pattern*, ...

Description

Sort input sections that match at least one of the *pattern*s in the list of patterns (*pattern*,. . .), in alphanumeric order.

-Xstack-usage

Gather and display stack usage at link time.

Synopsis

-Xstack-usage**[=***n***]**

Description

This option instructs the linker to inspect the stack usage of all functions of all objects and libraries that are linked. The linker will then produce a report to the standard output. Stack usage is determined by static code analysis, which (by definition) cannot correctly calculate dynamically determined stack usages, such as recursions or dynamic stack allocations. Whenever the stack analysis encounters unpredictable stack usages it will report that fact. In that case the calculated numbers represent lower bounds of the stack usage. The number provided as an argument to this option is a bit mask. The bitwise OR of the following masks can be used to enable the following options:

0x1

Enable default output.

0x2

Enable optional fifth output column (automatically adds 0x1 mask).

0x4

Show functions that do not have stack usage information (automatically adds 0x1 mask).

0x8

Show functions that are known to use no stack (local=nested=0) (automatically adds 0x1 mask).

0x20

Do not show information for static (local) functions.

0x40

Show the index of the file in which a function is defined (helps disambiguate multiple functions with the same name).

If *n* is not provided, *n* defaults to 0x1.

The report is a table with four columns that are shown by default and one optional column. The meaning of the columns are:

Function

The name of the function.

(not labeled)

Information flags concerning the stack usage of the function. The flags are:

?

No stack usage information is available for the given function.

A

Function uses dynamic stack allocation. Both local and nested stack usage is lower-bound.

I

Function uses indirect calls (i.e. function pointers). Nested stack usage is lower-bound.

L

Function is static.

R

There is a recursive call chain that originates in this function. Nested stack usage is-lower bound.

T

Function uses tail calls (i.e., it removes its own stack frame before making the call). Both local and nested stack usage remain exact (unless turned into lower bounds by other effects) but the local frame does not add to the nested size.

Local

The size in bytes of the function's own (i.e., local) stack frame.

Nested

The maximum accumulated stack size when calling this function.

This size includes the local stack frame (unless in case of a tail call) and all the local stack frames of functions on a contiguous call chain (unless they use tail calls) originating in the current function. The shown size is the maximum number the linker was able to accumulate. This is not necessarily the stack usage of the longest call chain.

[Optional]

(not labeled) The call chain that led to the Nested stack usage sum.

-Xstop-on-redeclaration

Stop on redeclaration.

Synopsis

-Xstop-on-redeclaration

Description

By default, the linker issues a warning each time it encounters a redeclaration. If -Xstop-on-redeclaration is specified, which it is by default, the linker halts with warning on the first redeclaration. To disable this option, use -Xstop-on-redeclaration=0.

-Xsuppress-dot...

Suppress leading dots (“.”).

Synopsis

-Xsuppress-dot

Description

Suppress leading dots (“.”) in the object files following this option.

For PowerPC:

-Xsuppress-dot-coff

-Xsuppress-dot-elf

With these options, suppression occurs only for COFF and ELF files respectively.

-Xsuppress-path

Suppress paths in symbol table.

Synopsis

-Xsuppress-path

Description

In the symbol table, suppress any path name in “file” symbols (type **STT_FILE**). See the discussion of ELF symbol table fields in the *Wind River Diab Compiler User’s Guide*.

ELF object file format only.

-Xsuppress-section-names

Suppress section names.

Synopsis

-Xsuppress-section-names

Description

Do not output section names to the symbol table. This option is for other tools which cannot process these names.

-Xsuppress-underscore...

Suppress leading underscores (“_”).

Synopsis

-Xsuppress-underscore

Description

Suppress leading underscores (“_”) in the object files following this option. Note that for symbols with more than one leading underscore, only the first will be removed.

For PowerPC:

-Xsuppress-underscore-coff

-Xsuppress-underscore-elf

With these options, suppression occurs only for COFF and ELF files respectively.

-Xtricore-no-relax-ldata

Do not resolve references between modules compiled with and without -Xconst-in-data.

Synopsis

-Xtricore-no-relax-ldata

Description

See -Xtricore-relax-ldata.

-Xtricore-relax-eabi-elf

Allow mixing of TriCore object files with Wind River Diab Compiler object files.

Synopsis

-Xtricore-relax-eabi-elf

Description

Mark object files produced by when compiling for TC 1.6 ISA with the same ELF flags as the ones used to mark TC 1.3 ISA objects. The option is required if objects will be used along with objects produced by the Tasking toolchain for TriCore.

-Xtricore-relax-ldata

Resolve references between modules compiled with and without -Xconst-in-data.

Synopsis

-Xtricore-relax-ldata

Description

When -Xtricore-relax-ldata is set, the linker can resolve references properly between modules compiled with and without -Xconst-in-data. This is the default. For example, assume module **m1.o** has been compiled with -Xconst-in-data. If **m1.o** contains a reference to a small **const** symbol, the reference will be to a symbol in the **.sdata** section. However, if the symbol definition is in a module that has not been compiled with -Xconst-in-data, it will be in the **.ldata** section. -Xtricore-relax-ldata allows the linker to resolve the symbol properly.

See also:

-Xtricore-no-relax-ldata

-Xconst-in-...

-Xunused-sections...

Remove/keep unused sections.

Synopsis

-Xunused-sections-remove

Description

Same as -Xremove-unused-sections.

-Xunused-sections-keep

Same as -Xremove-unused-sections-off; see -Xremove-unused-sections.

-Xunused-sections-list

Print a list of removed sections.

Example

The following simple example uses a main function, functions for getting and setting memory mapped registers, an interrupt handler, and a linker command file to simply illustrate how to remove unused sections.

main( ) function in **main.c**:

The main( ) function sets one memory-mapped register and reads another one.

```
int main(int argc, char** argv)
{
     if (argc>1)
          set_reg2(atoi(argv[1]));
     return get_reg1();
}
```

Register-management functions in **mm_regs.c**:

The get( ) and set( ) functions operate on three different memory-mapped registers.

```
#define GEN_REG(name, offset)
static volatile unsigned int*
mm_##name##=(volatile unsigned int*) offset;
void set_##name##(int val) {*mm_##name##=val;}
int get_##name##(){return *mm_##name##;}
GEN_REG(reg1, 0xed800000)
GEN_REG(reg2, 0xed801000)
GEN_REG(reg3, 0xed802000)
```

Interrupt handler in **isr.c**:

The interrupt handler acknowledges and handles interrupt zero.

```
extern void ack_irq();
extern void handle_irq();
#pragma section ISR0 ".isr0" "" far-absolute X
#pragma use_section ISR0 isr0
#pragma interrupt
void isr0()
{
     ack_irq();
     handle_irq();
}
```

Original linker command file:

The original linker command file—which does not removed unused sections— includes the following element:

```
SECTIONS
{
     [...]
     /* Place isr0 at its respective offset in the IVT */
     .isr0 BIND(0xff00f000) : { *(.isr0) }
     [...]
}
```

Removing sections:

In this example, the get function for register 1, the set function for register 2, and both the set and get functions for register 3 are unused. To get rid of them, split the sections into one section per function, and then remove the unused sections. To see what is happening, also list the removed sections. To do so, compile all sources with -

Xsection-split=5, and link with -Xremove-unused-sections and - Xunused-sections-list. The output from the linker is as follows:

```
Removed unused sections:
[ 1] '.text' '' in 'objects/isr.o', 0 bytes
[ 3] '.isr0' 'isr0' in 'objects/isr.o', 32 bytes
[ 1] '.text' '' in 'objects/main.o', 0 bytes
[ 1] '.text' '' in 'objects/mm_regs.o', 0 bytes
[ 3] '.text.set_reg1' 'set_reg1' in 'objects/mm_regs.o', 36 bytes
[ 9] '.text.get_reg2' 'get_reg2' in 'objects/mm_regs.o', 28 bytes
[11] '.text.set_reg3' 'set_reg3' in 'objects/mm_regs.o', 36 bytes
[13] '.text.get_reg3' 'get_reg3' in 'objects/mm_regs.o', 28 bytes
[...]
```

(Note that the **.text** sections of the objects are empty because they have been fully split, and the output omits some removed sections from libraries.) As you can see, the unused get and set routines have been removed. However, the interrupt handler has been removed as well, which is obviously undesirable. To ensure that the interrupt handler is not removed, the linker command file must be changed. To keep the whole output section—which is generally a reasonable approach—make the following change:

```
SECTIONS
{
     [...]
     .isr0 BIND(0xff00f000) KEEP : { *(.isr0) }
     [...]
}
```

However, to keep all input sections that match the input section specification—which makes most sense if the output section consists of multiple input sections specs—alternatively make the following change to the linker command file:

```
SECTIONS
{
     [...]
     .isr0 BIND(0xff00f000) : { KEEP( *(.isr0)) }
     [...]
}
```

Either approach changes the compilation output to the following:

```
Removed unused sections:
[ 1] '.text' '' in 'objects/functions3.o', 0 bytes
[ 1] '.text' '' in 'objects/isr.o', 0 bytes
[ 1] '.text' '' in 'objects/main.o', 0 bytes
[ 1] '.text' '' in 'objects/mm_regs.o', 0 bytes
[ 3] '.text.set_reg1' 'set_reg1' in 'objects/mm_regs.o', 36 bytes
[ 9] '.text.get_reg2' 'get_reg2' in 'objects/mm_regs.o', 28 bytes
[11] '.text.set_reg3' 'set_reg3' in 'objects/mm_regs.o', 36 bytes
[13] '.text.get_reg3' 'get_reg3' in 'objects/mm_regs.o', 28 bytes
[...]
```

With either of the changes to the linker command file, only the functions that are really unused have been discarded.

See also -Xremove-unused-sections.

-Xweak-syms-policy

Cause a weak reference to load the linker an object from a library, per the ELF spec.

Synopsis

-Xweak-syms-policy

Description

A weak reference from an (ELF) object file is an undefined **WEAK** symbol in its symbol table. A weak reference must not cause the linker to load an object from a library, per the ELF spec. However the linker has always done just that prior to Diab compiler version 5.9.4.6. To retrieve this linker behavior back, please set linker option -Xweak-syms-policy=0x1.

-Xwhole-program-jobs

Specify the number of files for the linker to compile simultaneously.

Synopsis

-xwhole-program-jobs=*n*

Description

By default, -Xwhole-program-optim recompiles files at link time sequentially. If -Xwhole-program-jobs=*n* is used, however, the link-time optimization process is parallelized by recompiling *n* files at a time. This may improve overall build time on machines with multiple cores. See also -Xwhole-program-optim.

-Xwhole-program-output-dir

Set the directory for whole-program optimization.

Synopsis

-Xwhole-program-output-dir=*dir*

Description

Specify the directory for storing whole-program optimization (WPO) object files, instead of using the default temporary directory, which is cleaned at the end of linking. It is highly recommended that you specify a directory when using WPO.

When specifying this option on the driver command line, remember to prefix it with -Wl, e.g.:

```
% dcc foo.c -XO -Wl, -Xwhole-program-output-dir=<temp_dir>
```

From the linker command line, "-Wl" is not needed. For more on whole-program optimization, see -Xwhole-program-optim and the *Wind River Diab Compiler User's Guide*.