CompactRISC™

Object Tools Reference Manual

Part Number: 424521772-005

August 1998

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
0.6	August 1995	First beta release.
0.7	January 1996	Minor changes and corrections.
1.0	August 1996	CR16A Product Version. CR16A Beta Version.
1.1	February 1997	Minor modifications and corrections.
2.a	September 1997	Alpha release for CR16B.
2.0	January 1998	Beta release.
2.1	August 1998	Product release.

PREFACE

Welcome to the CompactRISC Object Tools. This manual is divided into two parts. Part 1 describes the CompactRISC Linker, used to create executable files. Part 2 describes a number of utilities which help you organize and manipulate object files.

The information contained in this manual is for reference only and is subject to change without notice.

CONTENTS

Chapter 1
OVERVIEW

1.1 INTRODUCTION

This guide describes the CompactRISC Object Tools, which include the following:

crlink The CompactRISC Linker combines a number of object files, created by the CompactRISC Assembler and Compiler, into one executable object file which can be executed on a target board.

The linker combines object files by resolving symbolic references, allocating output sections, and relocating memory addresses.

The linker is controlled by the invocation line and a directive file. You can use the default directive file, supplied with the CompactRISC package, or your own directive file. The ability to write your own file is especially useful for embedded applications.

crlib The CompactRISC Archiver, **`crlib`**, maintains groups of object files in a single archive.

crprom The CompactRISC **`crprom`** utility converts executable object files to a format which can be used to burn EPROMs.

crview he CompactRISC **`crview`** utility displays, in a formatted manner, all the information that exists in a CompactRISC object file.

1.2 THE COMMON OBJECT FILE FORMAT

Many software development tools, including the CompactRISC development tools, use the Common Object File Format (COFF) as the standard object file format.

Object files are made up of sections. A section is a contiguous block of code, or data, having common attributes, and is the smallest unit of relocation. A section can have any name.

Traditionally, both the compiler and the assembler created the following default section names:

- **.text**.
  The **.text** section contains executable code.
- **.data**.
  The **.data** section contains initialized data. This data is available at run-time without any explicit assignment statement from the program.
- .**bss**.
  The **.bss** section contains uninitialized data. Since this data is uninitialized, the **.bss** section does not occupy space in the object file. When program execution starts, the **.bss** values found in memory are environmentally dependent. Most environments initialize the **.bss** section to zeroes.

The assembler still creates these default names.

To optimize program space consumption, the compiler creates the following sections, instead of **.bss** and **.data**:

- **.rdata_4**, .**rdata_2**, .**rdata_1**.
  for ROM data of sizes 4, 2, and 1.
- **.data_4**,.**data_2**, .**data_1**.
  for initialized data of sizes 4, 2, and 1.
- **.bss_4**, **.bss_2**, .**bss_1.**
  for uninitialized data of sizes 4, 2, 1.

1.3 LINKER INPUT AND OUTPUT FILES

Input

The linker input consists of simple object files produced by the assembler or compiler, partially linked object files produced by a previous linking operation, and library files. These input files are combined to produce an output file. The linker directive file can also be considered as linker input. Figure 1-1 shows the linker input, and output file options.

Simple object file

The simple object file is the most common type of linker input. It is created when the assembler or compiler translates source-language programs into COFF. Simple object files may contain unresolved external references. A simple object file is specified either on the invocation line (see Section 2.2) or in a directive file (see Section 3.5).

Partially linked object file

The partially linked object file is produced by a previous linking operation. It contains unresolved external references, and is therefore not executable. Partially linked object files must be included as input in a subsequent linking operation that produces an executable file.

The partially linked object file is used for many programming needs:

- A complex linking task is made easier by creating small groups of simple object files. Each group is then linked together to create a partially linked object file. Finally the partially linked object files can be linked to produce an executable object file.
- A large program can be modified without relinking the entire program.
- A set of routines can be produced for use in different application programs.

Refer to Section 2.3.7 for an explanation of the Retain Option used to create partially linked files.

Library file

The library file is a collection of simple object files, each representing a useful function. Several library files are supplied as part of the CompactRISC software package. You can also create your own library file using the CompactRISC archiver (see Chapter 5).

Library members that are referenced (by an external symbolic reference) are selected by linker and included in the linking process. Library members that are not referenced are not included in the linking process.

See Section 2.3.3 for the invocation line option to specify a library file.

Directive file

The directive file controls certain actions of the linker (specifically memory allocation). You can exercise considerable control over the linking operation by creating your own directive file. This feature is especially useful for embedded applications.

Chapter 3 offers a detailed description of the directive file.

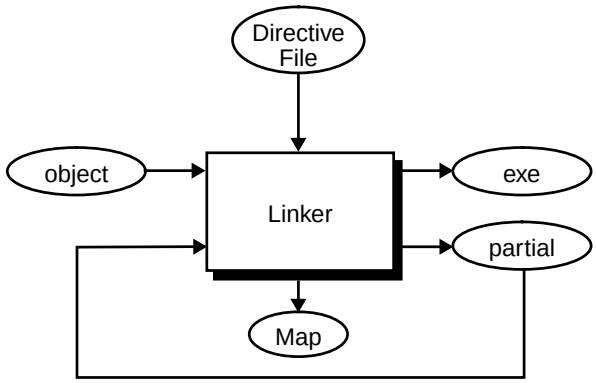


Figure 1-1. Linker Input and Output File Option

Output

Output of the linker consists of executable object files and partially linked object files. A memory map can also be considered as linker output.

"Output section" is defined as a section of a linker output object file.

Executable object file The executable object file is the final output of the linker. In an executable object file all external symbolic references have been resolved. The executable object file is therefore in a form that can be executed on the CompactRISC-based target system.

Memory map The memory map illustrates the allocation of memory after the linking process. It also illustrates the composition of the output sections from input sections. Section 4.2.6 provides a complete explanation of the memory map.

1.4 LINKER FUNCTIONS

The linker performs three basic functions: resolution of symbolic references, allocation of output sections, and relocation of memory addresses.

Resolution of symbolic references A symbol is used either to mark a program location, or to represent a data element. Object files contain a symbol table which holds information about symbols defined or referenced in the source program.

An external symbol is a symbol that can be referenced from any object file.

The linker resolves references to external symbols. The resolution of symbolic references is the process by which the linker matches an external symbolic reference with its definition (see Chapter 4).

A symbol can also be defined in a directive file. Section 3.8 explains the use of the assignment statement for this purpose.

Allocation of output sections The linker determines which part of memory is available for the allocation of output sections. The linker then assembles output sections from input sections and binds each output section to a starting memory address.

The directive file allows you to specify memory configuration. The directive file informs the linker which parts of memory are available for allocation (see Section 3.6), how to construct output sections from input sections, and how to allocate memory for these output sections (see Section 3.7). Control over the allocation of memory allows you to create a memory layout for various hardware requirements.

See Chapter 5 for a complete description of allocating output sections.

Relocation of memory addresses Once external symbolic references have been resolved, and output sections allocated to memory, the linker assigns each symbolic reference its actual memory address.

Chapter 6 explains the relocation of memory address.

Chapter 2
INVOKING THE LINKER

2.1 INTRODUCTION

This chapter explains the CompactRISC Linker invocation line. The linker is directly invoked by specifying the appropriate invocation name followed by invocation line arguments. These arguments specify a list of object and library files to link and linker options.

The linker is often invoked by the C compiler. A compiler driver invokes the linker with a predetermined set of linker options. If these options are not suitable for your needs, you can either force the compiler driver to pass specific options to the linker, (see Section 2.3.3 in the *CompactRISC Toolset - C Compiler Reference Manual*) or terminate the compilation process after object file creation and invoke the linker directly.

2.2 INVOCATION LINE

crlink [**{** ***option*** **|** ***filename*** **| @argfile}]...**

crlink is the linker name.

filename is any valid object or library file. Object and library filenames must be separated by a space. ***filename*** must include a complete or relative pathname if the specified object or library file is not in your current directory.

option is any valid linker invocation line option. Each option is preceded by a dash (**-**). Options must be separated by a space.

@argfile a file containing arguments for the linker invocation line. The linker replaces ***@argfile*** with these arguments, and handles them as if written in the invocation line.

filename and ***option*** can be placed in any order within the invocation syntax. However, object files which contain symbolic references to a library must precede that library filename on the invocation line. Libraries specified explicitly or through the **-l** invocation line option are processed as they are encountered.

The linker invocation syntax is case-sensitive.

Invoking the linker without options, or filenames, prints the usage line and version number of the currently used linker.

2.2.1 Libraries

Libraries can be specified through the **-l** invocation-line option (Section 2.3.3). The linker searches for these libraries by directory, according to a default directory list. The default library location is ***crdir*/lib** (where ***crdir*** is the top-level directory of the installed CompactRISC tools). For the CR16B large programming model, the default library location is ***crdir*/libl**.

The **LIBPATH** environment variable can be set to override the default directory search.

set LIBPATH=*directory-search-list*

directory-search-list
is a list of library directories separated by colons.

At link-time, the linker searches the listed directories for the libraries specified with the **-l** invocation line option.

2.3 INVOCATION OPTIONS

This section describes the linker invocation options that control linking operations such as specifying a directive file, requesting an output memory map, naming the output file, and suppressing error and warning messages.

Table 2-1 provides an abbreviated syntax guide to the invocation options.

2.3.1 Specify Output Filename

Use this invocation option to specify a name for the output file that is produced by the linker. This option overrides the default output filename.

-o *filename*

filename is any valid filename.

The default output filename is **cr.x**

2.3.2 Specify Directive File

Use the Specify Directive File invocation option to designate a directive file to be used by the linker in creating the executable object file. This invocation option overrides the default directive file used by the linker.

-d *filename*

filename is any valid directive file. ***filename*** must include a complete pathname if the directive file is not in your current directory.

LINKERFILE When a directive file is not specified in the invocation line, the linker uses the directive file specified by **LINKERFILE**.

If the **LINKERFILE** parameter is missing, the linker uses the default directive file **linker.def**, located in the CompactRISC root directory.

If **linker.def** does not exist, the linker issues a warning message and then follows a predefined trivial link process. You should not depend on the trivial link process to produce meaningful results.

See Chapter 3 for a full explanation of the linker directive file.

2.3.3 Specify Standard Library Filename

The Specify Library Filename invocation option can be used to specify standard libraries to be used by the linker. This option is useful for specifying system libraries.

-lx

x is a sequence, of up to nine characters, that defines a system library name in the filename **libx.a**.

On the invocation line the **-lx** option must follow the list of object files with external references that are resolved in the specified library. The linker first searches for this library in the directories specified with the **-L** invocation line option (Section 2.3.4) and then in the default library locations (Section 2.2.1.)

2.3.4 Specify Library Directory

Use this option to define the directory in which the linker first searches for a library specified with the **-l** invocation option (Section 2.3.3).

-L*dir*

dir is any valid directory pathname containing user libraries.

The **-L** option must precede any **-l** option. The linker searches for libraries specified through the **-l** invocation line option first in ***dir*** and then in the default library locations (Section 2.2.1).

2.3.5 Request Memory Map

Request Memory Map invocation option generates a memory map of the executable object file. The format and contents of the memory map are detailed in Section 4.2.6.

-m

The memory map is sent to standard output. You must redirect standard output to save the memory map to a file.

2.3.6 Specify Program Entry Point

Use this option to indicate to the linker the entry point of your program. The operating system, or the debugger, uses this entry point as the starting point for program execution. Entry point information is part of the COFF file and does not necessarily represent the actual beginning of the .text section.

By default, the linker looks for the external symbol **start** as indicating the entry point. If **start** is not found, the linker issues an error message.

-e *symbol*

symbol is an external symbol that marks the program entry point.

2.3.7 Retain Relocation Information

The Retain Relocation Information option must be used if the product of the linking operation is to be a partially linked file (i.e., not all symbolic references have been resolved) that may be used as input in a subsequent link.

-r

Use of the Retain Relocation Information option ensures that the linker retains relocation information and does not issue a linking error for unresolved external references.

2.3.8 Keep Relocation Information

This option allows you to instruct the linker to keep relocation information in your executable object file.

-k

Relocation information is used to calculate the actual address of a data or routine reference. Normally, executable object files do not have relocatable information since final addresses have been calculated by the linker. However, the Keep Relocation Information option instructs the linker to keep relocation information in your executable object file. This may be useful on systems that implement dynamic (load-time) address relocations.

2.3.9 Strip Symbolic Information

This option instructs the linker to remove the symbol table and line number information from the executable file the linker produces, thereby reducing the size of the executable file.

-s

Since symbolic information is used for debugging and for relocation of memory addresses, do not specify the strip symbolic information option if the output file is either to be used in debugging or is a partially linked object file.

2.3.10 Strip Local Symbolic Information

The Strip Local Symbolic Information invocation option removes only local symbolic information from the linker output file. This option is useful for reducing the size of linker output files.

-x

Since local symbolic information is used for debugging, do not specify the strip local symbolic information option if the output file is to be used in debugging. However, the output file can be a partially linked object file.

2.3.11 Specify Undefined Symbol

The Specify Undefined Symbol invocation option allows you to create a "pseudo" external reference to a symbol. This may be used to force the linker to select a library member for the linking process. Unless this option is specified, the linker links library members only if they resolve an external symbolic reference from a previously specified object file.

-u *symbolname*

symbolname is any valid symbol name.

2.3.12 Specify Fill Value for Section Gaps

This invocation option is used to specify a fill value other than zero for section gaps. By default, the linker fills any gaps created in output sections with a zero value.

-f *int*

int is an 2-byte unsigned integer constant.

2.3.13 Suppress Size Warning Message for Common Data

Use this invocation option to suppress the issuance of warning messages by the linker when various references to common data are of different memory address sizes.

-t

Common data are consolidated by the linker and allocated to a linker created **.bss** section. Normally, every reference to common data is of the same size. However, if the references are of different sizes, the linker issues a warning message for each reference that is contrary to the first found common data size. In this case the linker uses the largest size.

2.3.14 Suppress Error Message

This option instructs the linker to suppress all nonfatal error messages that describe problems encountered during the linking operation.

-s

Only fatal errors cause the linking operation to abort immediately. Error messages describing fatal errors are issued by the linker.

2.3.15 Issue Warning for Defined Common Data

This option causes the linker to issue a warning whenever a common variable is defined later on in a program.

`-M`

2.3.16 Output Linker Version Information

This option produces information regarding the version and revision numbers of the linker in use.

`-V`

Version and revision number information is useful to determine whether changes and updates in the linker package apply to the linker you are using.

2.3.17 Specify Version Stamp

This option specifies a version stamp for identifying linker output files.

`-VS` ***`int`***

A version stamp is a 16-bit integer constant. It is stored in a special field in the optional header of linker output files.

2.3.18 Enable Code Overlay Allocation

This option enables binding a number of text output sections to the same memory address, i.e., to allocate an overlay code.

`-O`

This option should be accompanied by appropriate instructions in the linker definition file, See Section 4.2.5 for further details.

2.3.19 Dump Errors and Warnings Into a File

Dumps both errors and warnings into an error file.

`-z`

The error file name is ***filename*.err** where ***filename*** is the base name of the executable file. For example:

```
crlink -z test.o -o test.x
```

generates the error file **test.err**.

```
crlink -z test.o
```

generates the error file **cr.err**.

-zn*filename*

The error file name is ***filename***. For example:

```
crlink -zntest new_test.o
```

generates the error file **test**.

Note, there must be no space between **zn** and ***filename***.

2.3.20 Enable Bank Switching Mechanism

This option enables the Bank Switching mechanism. This mechanism is needed for CR16A applications with code size larger than 128 Kbytes. (For a full explanation see Section 4.2.6.)

-BS

This option should be accompanied by appropriate instructions in the linker definition file. See Section 4.2.6 for further details.

Notes

The Bank Switching mechanism requires additional chip-specific hardware. It is currently supported for CR16A-based chips only.

The CompactRISC debugger does not support Bank Switching.

Table 2-1. Environment Invocation Options

Option	Explanation	Section
-k	Keep relocation information in executable file	2.3.8
-V	Output linker version information	2.3.16
-r	Retain relocation information	2.3.7
-m	Request an output memory map	2.3.5
-d ***filename***	Specify a directive file	2.3.2
-L***dir***	Specify a directory to search for libraries	2.3.4

Option	Explanation	Section
-f ***int***	Specify a fill value	2.3.12
-l***x***	Specify a library file for linking	2.3.3
-e ***symbol***	Specify a program entry point	2.3.6
-u ***symbol***	Specify an undefined symbol	2.3.11
-o ***filename***	Specify an output filename	2.3.1
-VS ***int***	Specify a version stamp	2.3.17
-x	Strip local symbolic information	2.3.10
-s	Strip symbolic information	2.3.9
-S	Suppress error messages	2.3.14
-M	Issue warning for defined common data	2.3.15
-t	Suppress size warning for common data	2.3.13
-O	Enable code overlay allocation	2.3.18
-z[**n*filename***]	Dump errors and warnings into a file	2.3.19
-BS	Enable bank switcing mechanism (for CR16A-based chips, requires additional hardware)	2.3.20

Chapter 3
THE LINKER DIRECTIVE FILE

3.1 INTRODUCTION

The linker directive file controls the linker functions. It mainly dictates memory configuration, output section content, and allocation of output sections.

A default linker directive file, **`linker.def`**, is supplied with the CompactRISC software package. It contains predefined directive definitions. This file resides in the CompactRISC top-level directory and includes the configuration to produce an executable object file for a CompactRISC development board.

You can exercise considerable control over the linking operation by creating a directive file that contains directive definitions tailored to your unique needs. This directive file is especially useful for embedded applications. The linker is instructed to use a user-written directive file through the Specify Directive file invocation option (see Section 2.3.2).

This chapter provides an overview of the directive file. Section 3.2 explains the structure. Section 3.3 briefly describes the expressions used in the directive file. The remainder of the chapter defines the various parts of the directive file, providing a detailed description of use, syntax and examples.

3.2 STRUCTURE OF THE DIRECTIVE FILE

A directive file is made up of the following parts:

- Comments. A comment may be used for documentation purposes.
- Input file specifications. An alternative to specifying input files on the invocation line.
- **`MEMORY`** statements. A **`MEMORY`** statement is used to define which parts of the memory are available for allocation of output sections.
- **`SECTIONS`** statements. A **`SECTIONS`** statement is used to control the construction of output sections from input sections and the allocation of output sections to memory addresses.

- Assignment statements. An assignment statement both defines a symbol and assigns the symbol an absolute address.
- Output file options. Controls certain output file characteristics.

The following is an example of a Directive File:

Example

```
/* Input file specification */

a.o
b.o
c.o
/* Memory configuration */

MEMORY {
          mem1 : origin=0x10000, length=0x8000
          mem2 : origin=0x20000, length=0x8000
}

/* Output section construction and allocation */

SECTIONS {
          .text  BIND(0x10000)          : { *(.text) }
          .data INTO(mem2)              : { *(.data)
*(.data_2) }
          .bss   INTO(mem2) ALIGN(64)  : { *(.bss) *(.bss_1) }
}

/*Special symbol assignment */

end_bss = ADDR(.bss) + SIZEOF(.bss) ; /* End of .bss section */
```

3.3 DIRECTIVE FILE EXPRESSIONS

Directive file expressions are used as arguments for certain options and as right-hand-side of assignment statements. Expressions consist of integer constants, operators, special functions, and parentheses. The value of a directive file expression is always a 4-byte unsigned integer. The value generally represents a memory address.

Examples

```
0x1000
```

`0x1000` is an integer constant having the value of 1000 in hexadecimal.

```
ADDR(.text)+SIZEOF(.text)
```

This is the sum of the start address of the **`.text`** output section and its size. The result is the address following the end of the **`.text`** section.

Expressions are used in two places in the directive file:

- As arguments to the **BIND, ROMBIND** and **ALIGN** options and to the **NEXT** function.
- As the right-hand side of an assignment statement.

3.3.1 Integer Syntax

The linker accepts three radices for unsigned integer input: decimal (the default), hexadecimal, and octal. Integer input in the directive file syntax is denoted by the word ***int***. Unless otherwise noted, ***int*** represents a 4-byte unsigned integer.

Decimal A decimal value begins with a digit in the range of 1 through 9 followed by optional digits in the range of 0 through 9.

Octal An octal value begins with 0 followed by digits in the range of 0 through 7.

Hexadecimal A hexadecimal value begins with either **0x** or **0X**, followed by digits in the range of 0 through 9 and/or letters in the range of A through F (either upper- or lower-case).

The linker supports the following unary operators:

Precedence	Operator
1	! Logical negation
1	~ One's complement
1	- Two's complement

A unary operator has a higher precedence than a binary operator in expression evaluation.

3.3.2 Unary and Binary Operators

The linker supports the following unary and binary operators (listed in order of precedence):

Precedence	Operator
UNARY	
1	! Logical negation
1	~ One's complement
1	- Two's complement

Precedence	Operator
BINARY	
2	* (multiplication) / (division) % (modulus)
3	+ (addition) - (subtraction)
4	>>(right shift) <<(left shift)
5	> (greater than) < (less than) >=(greater than or equal) <=(less than or equal)
6	==(equal) != (not equal)
7	& bitwise AND | bitwise OR
8	&&logical AND || logical OR

3.3.3 Assignment Operators

The value of an expression may be assigned to a symbol in one of five ways:

symbol = *expr*; (assign the value of ***expr*** to **symbol)**

symbol += *expr*; (equivalent to: **symbol = symbol + *expr*)**

symbol -= *expr*; (equivalent to: **symbol = symbol - *expr*)**

symbol *= *expr*; (equivalent to: **symbol = symbol * *expr*)**

symbol /= *expr*; (equivalent to: **symbol = symbol / *expr*)**

The assignment syntax always requires a semicolon after the expression.

3.3.4 Special Functions

Five special functions provide useful information about output sections and memory addresses. These functions and the information they return are listed in Table 3-1.

Table 3-1. Special Functions

Function	Returned Value
SIZEOF	Size of a specified output section
ADDR	Memory address of a specified output section
FILEADDR	File address of a specified output section
NEXT	Next memory address aligned to a specified value
HIGHMEMADDR	Next address after highest allocated memory address

Output function size

The size of the output function returns the number of bytes in the specified output section. The syntax for the **SIZEOF** function is:

SIZEOF (*section_name*)

The **SIZEOF** function can return a valid value only for a section which has already been created, otherwise it returns a zero.

If more than one section exists with the same name, the information returned will be relevant only for the first section recognized.

Output function size

The memory address function returns the starting address of the specified output section. The syntax for the memory address function is:

ADDR (*section_name*)

The **ADDR** function can return a valid value only for a section which has already been allocated memory space, otherwise it returns zero.

If more than one section exists with the specified section name, the information returned will be relevant only for the first section recognized.

File address

The file address function returns the file address of a section's raw data in the output file. The syntax for the file address function is:

FILEADDR (***section_name***)

The **FILEADDR** function can return a valid value only for a section which has already been allocated file space in the output file, otherwise it returns zero.

If more than one section exists with the specified section name, the information returned will be relevant only for the first section recognized.

Next address

The next address function returns the next available memory address (i.e., after the most recently allocated output section), which is a multiple of a specified value. The syntax for the next function is:

NEXT (***int***)

int must be greater than zero.

Highest memory address The highest memory address function returns the next available memory address after the highest address that has been allocated in memory. The syntax for the highest memory address is:

HIGHMEMADDR

3.4 COMMENT

A comment can be placed anywhere in the directive file. Comments begin with a slash and asterisk (/*) followed by one or more lines of text. The comment is terminated with an asterisk and slash (*/). Comments cannot be nested.

3.5 INPUT FILE SPECIFICATION

As an alternative to specifying input files on the linker invocation line, you can specify input files in a linker directive file. An input file is any object or library file to be linked.

Input filenames may appear anywhere in the directive file, except within a **MEMORY** or **SECTIONS** statement. The placement is significant because the linker processes input files as they are encountered in the directive file. It is recommended that you place input filenames before the **SECTIONS** directive so that the **SECTIONS** directive is applicable to all input files.

filename is any valid input filename. It may include a full or partial pathname. A filename containing special characters should be enclosed in double-quotes (" ") to avoid conflict with definitions of special characters in the directive file.

3.6 MEMORY STATEMENT

Use the **MEMORY** statement to specify the configured and unconfigured (i.e., non-available) areas of memory. If a **MEMORY** statement is not specified, the linker assumes the maximum amount of configured memory address space available to the CompactRISC family member (e.g., in CR16B small, **0x0** through **0x1ffff**).

If one or more **MEMORY** statements are specified, the linker treats all memory areas not within these statements as unconfigured. Unconfigured memory is not used in the linker allocation process. Therefore, output sections can not be allocated within unconfigured memory.

```
MEMORY          {
                mem_name[(attributes)] : ORIGIN = int , LENGTH = int
           ...
                }
```

mem_name is any symbolic name to be associated with the specified configured memory area.

int is a valid integer constant (in decimal, hexadecimal, or octal format).

attributes are a sequence of one or more of the following attribute letters:

I - The named memory area is initializable.

R - The named memory area is readable.

W - The named memory area is writable.

X - The named memory area is executable.

Attribute letters can only be used to direct a section to a memory area with specified attributes (see Section 3.7.2). Attribute letters have no other meaning.

A configured memory area is a contiguous block of memory. It starts at the address specified by the value given to **ORIGIN** and contains the number of bytes specified as the value of **LENGTH**. **ORIGIN** may be abbreviated to **ORG**, and **LENGTH** may be abbreviated to **LEN**.

Any number of configured memory areas may be declared within one **MEMORY** statement. However, if more than one memory area is declared, no overlap should exist among the specified areas. A memory area overlap causes the linker to issue an error message and terminate the linking process.

Each memory area can be referenced from the **SECTIONS** statement either by name or by attribute.

Example

```
MEMORY {
     ROM (R)    origin = 0x1000 length = 0x40000
     RAM (RW)   origin = 0x8000 length = 0x100000
}
```

3.7 SECTIONS STATEMENT

Use the **SECTIONS** statement to specify how output sections are constructed from input sections, and to allocate memory for output sections.

```
SECTIONS {
     output_section_name [options] : {input_section_spec ...
}
     ...
}
```

output_section_name
is the name of the output section to be created. ***output_section_name*** can be any name of up to eight characters.

options are a list of allocation options (Section 3.7.2) or section type options (Section 3.7.4).

input_section_spec
is a specification of an input section. This input section is combined with the other specified input sections to produce an output section.

Example

```
SECTIONS {
     .text BIND(0x8000):{ file1.o(.text) file2.o(.text) }
     .data ALIGN(16):{ file1.o(.data) file2.o(.data) }
}
```

The use of the **SECTIONS** statement is described below.

3.7.1 Input Section Specification

Input sections are specified in the **SECTIONS** statement as follows:

1. Specify all sections of the input file

filename is any valid input filename. The filename can include a full or partial pathname. A filename containing special characters may be enclosed in double quotes (" ") to avoid conflict with the directive syntax. If ***filename*** is a library, this specification applies to all library members which have been selected for the linking process.

Example 1

```
.xxx : { abc.o }
```

Output section **.xxx** consists of all input sections of file **abc.o.**

2. Specify only certain sections of the input file.

filename (section_name ...)

Example 2 `.text : { file.o(.text .data) }`

Output section **.text** consists of sections **.text** and **.data** from **file.o.**

3. Specify only certain sections from *all* input files indicated on the invocation line or in the directive file before the **SECTIONS** statement.

(section_name ...)

Example 3 `.text : { *(.rdata_4) *(.text) }`

Output section **.text** consists of sections **.text** and **.rdata_4** sections from all input files.

4. Specify the initialization table created by the linker. The initialization table resides in a linker-created **.init** section (see Section 4.2.4 for an explanation of data initialization).

***[INIT]**

Example 4 `.text : { *(.text) *[INIT] }`

Output section **.text** consists of both the **.text** sections from all input files and the linker-created **.init** section (which contains an initialization table).

3.7.2 Allocating a Section to Memory

Output sections can be allocated to memory by using allocation options in the following ways:

1. Bind a section to a particular memory address by using the **BIND** option. This instructs the linker to assign a configured memory address to the specified output section.

BIND(*expression*)

expression is any valid linker expression. The expression value is a memory address to which the output is bound.

Example 1 `.text BIND(0x1000) : { *(.text) }`

The **.text** output section is allocated at address 0x1000.

2. Direct a section to a memory area by name using the **INTO** option. This instructs the linker to assign a memory address within the memory area to the specified output section. The output section must fit in the memory area.

INTO(*mem_name*)

mem_name is a name that has been associated with a configured memory area through use of the **MEMORY** directive (see Section 3.6).

Example 2

```
MEMORY {
     ROM : origin=0x1000 length=0x2000
     RAM : origin=0x5000 length=0x8000
}

SECTIONS {
     .text INTO(ROM) : { *(.text) }
     .data INTO(RAM) : { *(.data) *(.data_4) *(.data_2)
                       *(.data_1) }
     ...
}
```

The **.text** output section is allocated within the ROM memory area as defined with the **MEMORY** statement. The **.data** output section is allocated within the RAM memory area.

3. Direct a section to two memory areas while splitting the section, if necessary. This instructs the linker to allocate space in one, or two, of the specified memory areas for the specified output section, and to split the output section between them, if necessary.

INTO(*mem_name_1, mem_name_2*)

mem_name_1 and ***mem_name_2***
are names that have been associated with a configured memory area through use of the **MEMORY** directive (see Section 3.6).

Example 3

```
MEMORY {
     ROM1 : ORIGIN = 0 LENGTH = 0XD800
     ROM2 : ORIGIN = 0X10000 LENGTH = 0X10000
}

SECTIONS {
     .text INTO(ROM1, ROM2) : { *(.text) }
}
```

This is useful, for example, in a system with two non-consecutive ROM segments, if you do not want to split the code between them manually. Here the linker automatically splits the output section.

Note: if the linker splits an output section, it attempts to implement a best-fit approach to achieve optimal memory allocation. This may change the order of the input sections inside the output section.

4. Direct a section to memory by attributes by using the **INTO** option. The **INTO** option instructs the linker to assign a memory address, within any memory area having the listed attributes, to the specified output section.

INTO((*attributes*))

attributes is a sequence of attribute letters (I, R, W, X, meaning respectively init, read, write and execute).

Example 4

```
MEMORY {
     ROM1(R)     : origin=0x1000 length=0x2000
     ROM2(R)     : origin=0x5000 length=0x2000
     RAM1(RW)    : origin=0x8000 length=0x1000
     RAM2(RW)    : origin=0xa000 length=0x1000
}

SECTIONS {
     .text INTO((R)) : { *(.text) }
     .data INTO((RW)) : { *(.data)*(.data_2) *(.data_1)}
     ...
}
```

The **.text** output section is allocated within a memory area that has only read attributes (ROM1 or ROM2). The **.data** output section is allocated within a memory area that has read and write attributes (RAM1 or RAM2).

5. Bind or direct a ROM copy of a section to memory by using the **ROMBIND** and **ROMINTO** options. These options are equivalent to the **BIND** and **INTO** options, respectively. Use of the **ROMBIND** or **ROMINTO** option instructs the linker to allocate memory for a ROM copy of the specified output section. The output section therefore has two addresses: a ROM address and a RAM address. This can be used for data initialization (see Section 4.2.4 for details).

ROMBIND (*expression*)
ROMINTO (*mem_name*)
ROMINTO((*attributes*))

expression is any valid linker expression.

mem_name is a name that has been associated with a configured memory area through use of the **MEMORY** directive.

attributes is a sequence of attribute letters (I, R, W, X, meaning respectively init, read, write and execute).

Example 5

```
.data BIND(0x1000) ROMBIND(0x8000) : { *(.data) }
```

The **.data** output section is allocated at address 0x1000. A copy of the **.data** section is allocated at address 0x8000. This copy is used only for initialization purposes. At run-time the **.data** section is copied by an initialization routine from address 0x8000 (ROM address) to its actual (RAM) address 0x1000.

Example 6

```
MEMORY {
     ROM : origin=0x1000 length=0x2000
     RAM : origin=0x3000 length=0x3000
}

SECTIONS {
     .text INTO(ROM) : { *(.text) }
     .data INTO(RAM) ROMINTO(ROM) :
               { *(.data) *(.data_4) *(.data_2) *(.data_1)
                         ...
}
```

The **.text** output section is allocated within the ROM memory area as defined by the **MEMORY** statement. The **.data** output section is allocated within the RAM memory area, and a copy of the **.data** section is allocated within the ROM memory area. This copy is used only for initialization purposes.

3.7.3 Aligning a Section

Aligning an output section to an alignment value ensures that the output section is assigned a memory address that is a multiple of the value. The **ALIGN** option is used for this purpose.

ALIGN(*expression*)

expression is any valid linker expression (Appendix A).

Note The **ALIGN** option is ignored when it appears in conjunction with the **BIND** or **ROMBIND** allocation options, because the allocation options are specified with a particular address.

Example

```
.text ALIGN(16) : { *(.text) }
```

The **.text** output section is allocated anywhere within available configured memory but its address must be a multiple of 16.

Example

```
.text INTO(RAM) ALIGN(16) : { *(.text) }
```

The **.text** output section is allocated within the memory area RAM and its address is a multiple of 16.

3.7.4 Setting the Section Type

The COFF section headers contain information that indicate how the sections are to be handled by the debugger and linker, and what category of data is contained within the section. By default the linker determines the type of an output section based on the input sections comprising it. You may control the section type using the following options:

- **type_contents_option**

  Type options that specify the contents of output sections are:

  – **TYP_TEXT** section contains executable text.

  – **TYP_DATA** section contains initialized data.

  – **TYP_BSS** section contains only uninitialized data.

Example

```
.xxx (TYP_TEXT) BIND(0x2000) : {a.o(.yyy)}
```

The **.xxx** output section contains executable text from section **.yyy** of the file **a.o**.

- **type_control_option**

  The only type control option that is currently supported is:

  – **NOLOAD** this option specifies that the output section is not loaded.

Example

```
.data (NOLOAD) BIND(0x1000) : {*(.data_2)}
```

3.7.5 Grouping Output Sections

Several output sections may be grouped to create a contiguous block of memory by using the **GROUP** option. This allows you to allocate consecutive sections without having to specify an allocation option for each section. Although the output sections are grouped together in memory, they remain separate.

GROUP [***group_options*** **] : {**

output_section_spec [type_option] : {input_section_spec ...} }

group_options
are the allocation options **BIND**, **ROMBIND**, **INTO** and **ROMINTO**, and the option **ALIGN**.

output_section_spec
is a specification of the output section to be created.

type_option is any section type option (see Section 3.7.4 for the list of these options).

input_section_spec
is a specification of an input section.

Example

```
.text BIND(0x8000) : { *(.text) }
GROUP BIND(0xa000) : {
     .data : { *(.data) *(.data_4) *(.data_2) *(.data_1)}
     .bss  : { *(.bss) *(.bss_4) *(.bss_2) *(.bss_1) }
}
```

The **.data** and **.bss** output sections are grouped to a contiguous block of memory starting at address 0xa000.

Note Output sections that are specified within a **GROUP** option may be qualified only by section options, since all other options are specified in **GROUP** level.

3.8 ASSIGNMENT STATEMENT

Symbols can be defined and assigned a memory address at link-time through use of the assignment statement. This is useful for two reasons:

- To use link-time computed information at run-time. You can assign a symbol an expression that is calculated by the linker, and then use it in your program.
- To bind a symbol to an address in a flexible way. If you define a symbol in a directive file and later want to change its address, you simply change the assignment in the directive file and re-link. Therefore there is no need to recompile your program.

symbol = expression ;

symbol is any valid symbol name.

expression is any valid linker expression.

The syntax supports other assignment operators in addition to “=”, e.g., "+=" operator. (For a complete list of assignment operators see Appendix A.)

Example

```
abc = 0x1000 ;
```

A symbol named **abc** is defined, with address 0x1000.

Example

```
sdata = ADDR(.data)
```

A symbol named **sdata** is defined, and its address is the start address of the **.data** output section.

Example

C program:

```
extern int foo;
mem()
{
      int x;
      x=&foo;
}
```

linker definition file:

```
MEMORY{
     data_mem: origin = 2   length = 0xd7fe
     code_mem: origin = 1,1000 length = 0x10000
}

SECTIONS{
     .text  ALIGN(4) INTO(code_mem):{*(.text)}
     .rdata ALIGN(4) INTO(data_mem):{
               *(.rdata_4) *(dspm)*(.rdata_2) *(.rdata_1)}
     .data  ALIGN(4) INTO(data_mem):
               *(.data_4) *(.data) *(.data_2) *(.data_1)
               *(.static)}
     .bss  ALIGN(4) INTO(data_mem) {*(.bss) *(.bss_4)

*(.bss_2)*(.bss_1)}
}
_foo = 0x51;
```

The symbol **foo** is resolved by the **linker.def** symbol **_foo. x** takes the value 0x51. **x** can be changed by relinking, after changing the value of **_foo** in the **linker.def** file. It is not necessary to recompile.

Since the assignment of symbols to a memory address is made at the end of the linker allocation phase, the linker does not recognize addresses assigned previously. The assignment statement may thus be placed anywhere within the directive file.

Notes

1. The linker does not check that the memory address assigned to a symbol is within configured memory.

2. A symbol that was defined in the linker directives file with an assignment statement, does not have any specific type (e.g., integer). It can be referred to from C programs as an external variable of any type. However, the debugger does not recognize the type used in the C program, since the symbol was defined in the linker directives file, and is not associated with any particular type.

3.8.1 Symbol Assignment Within SECTIONS Statement

A symbol assignment may also be used within a **SECTIONS** statement. Such an assignment can use the symbol "." to denote the current location in memory. This assignment should appear as part of the input section specification list.

Example

```
.text : { *(.text) xxx = . ; }
```

A symbol named **xxx** is defined and assigned the end address of the .text output section (the end address is the value of the current location of the assignment).

Note This symbol assignment can also be specified outside the **SECTIONS** statement:

```
xxx = ADDR(.text) + SIZEOF(.text) ;
```

3.8.2 Creating Gaps Within An Output Section

The current location symbol itself may be assigned a value. This can be used to create a gap within an output section. The linker normally combines input sections in a contiguous fashion when creating an output section. However, by incrementing the value of the current location you can create a gap of unallocated space.

. += expression

expression is any valid linker expression.(See Section 3.3)

Example

```
.text : { a.o(.text) . += 0x1000 ; b.o(.text) }
```

The **.text** output section consists of the **.text** section of file **a.o**, a gap of 0x1000 (created by the current location assignment), and the **.text** section of file **b.o.**

By default, the linker fills the gaps created within an output section with zeros, or with a value specified with the Specify Fill Value invocation option (see Section 2.3.12). However, you can specify a fill value for a specific output section. This overrides any other specified fill value.

output_section_name [options] : {input_section_spec ... } = xfill_value

fill_value is a two-byte integer constant.

Example

```
.text : { a.o(.text) . += 0x1000 ; b.o(.text) } = 0xffff
```

Each word in the gap created inside the **.text** output section contains the value 0xffff (i.e., the gap is filled with 1's).

3.9 OUTPUT FILE OPTIONS

Three characteristics of the output file can be changed with output file options: the default filename, the default execution permission set, and the header magic number.

Chapter 4
LINKER FUNCTIONS

4.1 RESOLUTION OF SYMBOLIC REFERENCES

A symbol is used either to mark a program location or to represent a data element. In high-level languages, symbols represent variables, functions or labels.

Symbol definition

an external symbol is a symbol that can be referenced from any object file. The definition (defining point) of a symbol is a source program statement which associates the symbol with an explicit location in a section of the object file. A symbol can be defined only in one object file.

Symbol reference

A symbolic reference is the use of a symbol in a statement that is not its definition. An external symbolic reference is a reference to a symbol which is defined in another object file.

Examples

In the assembly language:

```
.globl abc
```

The **.globl** assembler directive is used to declare the symbol **abc** as external.

```
xxx:
```

This assembly language label defines the symbol **xxx** since it is associated with an explicit location within one of the sections.

```
yyy::
```

A label, followed by a double colon, defines an external symbol. This example is equivalent to:

```
yyy:
.globl yyy
```

The first statement defines the symbol **yyy**. The second statement declares it as external.

```
movd sym,r0
```

This is a reference to the symbol **sym**. If **sym** is not defined in the same program, the assembler considers this an external symbolic reference.

In the C language:

```
int i;
int j = 5;
main {
extern k;
...
}
```

The variable **i** is an external symbol since it is declared outside any function. **k** is also an external symbol because it is declared as such (by the term "extern"). And **j** is a defined external symbol since it is initialized at its point of declaration. This initialization associates **j** with a location within the **.data** section.

Symbol Resolution Using the Symbol Table

The COFF object file contains a symbol table which holds information about symbols defined or referenced in the source program. If a symbol was defined in the source program, it has a "defined" status in the object file's symbol table entry. If a symbol is only referenced in the source program (and not defined), it has an "undefined" status in the object file's symbol table entry.

The resolution of symbolic references is the process by which the linker matches an external symbolic reference with its definition. It does so by using the information in the symbol tables of the object files. If a symbol has the status "undefined" in one object file, the linker searches for the symbol's definition in the other object files. The linker checks that no symbol is defined more than once, and that there is no symbol left undefined. If such a symbol is found, the linker issues an error message and terminates the linking process.

Example

```
object_1                object_2
a : defined             a : undefined
b : defined             b : defined
c : undefined
                        d : defined
```

When linking the two object files, the symbols **a** and **d** are resolved correctly by the linker. **a** is defined in **object_1** and appears in **object_2** as undefined. **d** appears only in the object file in which it is defined. The linker issues an error message for symbols **b** and **c**, since **b** is defined twice and **c** is not defined.

4.1.1 Library Processing

The resolution of external symbolic references to a library member involves a slightly different process. A library member becomes part of the linker's input only if it contains a definition of an external symbol that has been referenced in a previous input file or library member. This is unlike regular object files, which are completely included in the linker's input.

A library file is a collection of object files typically containing useful routines. When the CompactRISC archiver builds a library from object files, it creates a symbol directory as the first member of the library. The library symbol directory is a list of all defined external symbols found in the library members. For each such symbol, there is a pointer to the library member where the symbol is defined. When the linker processes a library it scans the symbol directory, selecting the definitions that resolve currently undefined external symbols.

When a library member is selected for the linking process, it may create new external symbolic references (for example, one library member can refer to a symbol which is defined in another library member). For this reason, the linker scans the symbol directory of a library repeatedly until the definitions in the symbol directory no longer resolve external symbols (i.e. all references to library members have been resolved in previous passes). Therefore, for efficiency of the linking process, the ordering of library members should be such that a library member containing a reference to another library member should be place first in the library.

4.1.2 Symbol Definition in the Directive file

Normally, the definition of an external symbol is found in one of the input files. However, you can also define a symbol at link time through use of the assignment statement in the directive file (Section 3.8). This creates an external symbol and associates it with an absolute address.

4.1.3 Linker Defined Symbols

Certain special symbols are referenced in useful routines and have a universal use. These symbols have a default definition and value that is automatically assigned by the linker. You can override the default definition and value of special symbols by supplying your own definition (either in the source program or in the linker directive file). These symbols are:

Symbol Name	Meaning	Default Value
_etext	end address of **.text** section	end address of **.text** section
_edata	end address of **.data** section	end address of **.data** section
_end	start of heap	next address after highest allocated memory address
_HEAP$_START	start of heap	next address after highest allocated memory address
_HEAP$_MAX	heap limit address	highest address in configured memory
__STACK_START	initial top of stack	highest available address (depends on CPU architecture e.g., for cr16 - 0xffff)
__INIT_TABLE	address of initialization table	address of linker-created **.init** section
__STATIC_BASE_START	address of static data area	highest available address (depends on CPU architecture e.g. for cr16 - 0xffff)

Example In the assembly language:

```
movbd $__STACK_START, sp
```

The symbol **__STACK_START** is used to initialize the stack pointer (sp).

The symbol **__STATIC_BASE_START** is used to define a static data area. This is the pointer to the area used by the compiler or assembler to generate sb-relative data.

4.2 ALLOCATION OF OUTPUT SECTIONS

The allocation process of the linker takes place after all input files have been read and all external symbolic references have been resolved. This process includes constructing output sections from input sections and assigning memory addresses to each output section. The linker directive file can be used to exercise considerable control over the allocation of memory.

4.2.1 Creating Output Sections from Input Sections

Each output section is constructed from one or more input sections. The list of input sections to be combined to produce an output section is determined in two ways:

- Through the user-specified **SECTIONS** statement in the directive file (Section 3.7).
- By the default linker rule.

The default linker rule applies to input sections that have not been associated with an output section through use of the **SECTIONS** statement. Such input sections are associated with an output section that has the same name. If there is no output section with the same name, the linker creates a new output section with the input section name and associates the input section with the newly created output section.

Example

Consider two input files, **a.o** and **b.o**, each containing the input sections **.text**, **.data**, **.bss**. For the following **SECTIONS** statement:

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

Output sections are constructed as follows (Reason 1 indicates a user specification; Reason 2 indicates the linker default rule):

Output Section	Contents	Reason
.text	.text of a.o	1
	.text of b.o	1
.data	.data of a.o	1
	.bss of b.o	1
	.data of b.o	2
.bss	.bss of a.o	2

4.2.2 Assigning an Address to an Output Section

Before assigning an address to an output section, the linker determines which parts of memory are available for allocation. By default, the linker assumes that the maximum amount of configured memory space, (e.g., in CR16, 0x0 through 0xffff), is available for allocation. However, you can (and should) specify the areas of memory to be configured, and therefore available for allocation, through use of the **MEMORY** statement in the directive file (Section 3.6).

There are four phases in the allocation process (see Section 3.7.2):

1. Processing all the **BIND** options used in the **SECTIONS** statement. The **BIND** option has the highest priority in determining output section addresses.

2. Processing all the **INTO** options of the **SECTIONS** statement to direct output sections to memory areas by name.

3. Processing all the **INTO** options of the **SECTIONS** statement to direct output sections to memory areas by attributes.

4. Assigning memory addresses to all unallocated output sections using a find-first-fit algorithm.

If the linker cannot process any one of the above phases, it issues an error and terminates the linking process.

Example Consider the following directive file:

```
MEMORY {
     MEM1 (R) : origin = 1000 length = 1000
     MEM2 (RW): origin = 3000 length = 1000
}

SECTIONS {
     .text INTO(MEM1): { *(.text) }
     .data BIND(3500): {
               *(.data) *(.data_4) *(.data_2)*(.data_1)}
     .bss       : { *(.bss) *(.bss_3) *(.bss_2) *(.bss_1) }
}
```

Assume that the size of the output section of **.text** is 500, of **.data** is 400, and of **.bss** is 500. The steps in the allocation process are as follows:

1. Output section **.data** is allocated at address 3500, as specified in the **BIND** option.

2. Output section **.text** is allocated within memory area MEM1. Since this area is empty, the **.text** section is allocated at its starting address 1000.

3. Since there is no allocation option specified for the **.bss** output section, it is allocated in the first memory address where it fits. Though the **.bss** section does not fit in memory area MEM1, it does fit in the memory area MEM2. Therefore the **.bss** section is allocated at the starting address of MEM2, 3000.

4.2.3 Using the Linker Definition File to Overlay Input Sections

The linker supports overlays of input sections, in the sense that more than one input section can be allocated to the same output section location. This may only be done for uninitialized sections. To achieve this the location counter is changed in the middle of section allocation.

Example

```
SECTIONS{
          CCC       (NOLOAD) BIND (0x100):
          {(a.o: .bss_2) .=ADDR(CCC) (b.o:.bss_2)}
```

The **.bss_2** section of **a.o** is allocated to address 0x100. The location counter is then changed to point to the beginning of the **CCC** section, 0x100, and the **.bss_2** section of **b.o** is allocated starting at this address. As a result the **.bss_2** sections of **a.o** and **b.o** are overlayed.

4.2.4 Data Initialization Support

The linker supports two kinds of data initializations for an embedded environment:

- Variables that are initialized at their point of declaration rather than by assignment at run-time.
- Variables that are uninitialized are automatically initialized to zero.

By default the linker uses the following **libadb** predefined functions to do the initializations:

- **init_bss_data**
  Initializes the default initialized data sections and initializes the default uninitialized data sections (bss) to zero.
- **init_bss**
  Initializes non-initialized data sections to zero.
- **init_data**
  Initializes initialized data sections.
- **init_bss_data_code**
  Initializes data and bss sections and also jumps to RAM if copied data sections include code.

The default linker definition file uses the **ROMBIND** and **ROMINTO** output section options to define the ROM and RAM addresses of **.data** and **.bss** sections.

To implement data initialization you must do the following:

To initialize data sections, duplicate the section that contains the initialized data (typically the **.data** section) using the **ROMBIND** and **ROMINTO** output section options (Section 3.7.2). These options instruct the linker to assign a second address to the output section (a ROM address). The output section should be burned on ROM, and copied to RAM, at run-time, for it to be writable. The output section is thus allocated twice (on ROM and RAM). References to symbols that are defined in duplicated sections are modified by the linker according to the section RAM address, where it resides at run-time. In the start routine of the application program call one of the routines above that initializes data.

To initialize non-initialized data to zero, call one of the above mentioned routines, that initializes bss, from **libadb**.

In order to pass the initialization request information to the application, the linker creates an initialization table which includes:

- One kind of entry for each duplicated section. The information in this kind of entry may be used to copy sections from ROM to RAM.
- One entry for each section of uninitialized data (typically **.bss**). The information in this kind of entry may be used to initialize sections.

The structure of an initialization table is:

Type	Size	Name	Description
unsigned long	4	`i_size`	Section size
unsigned long	4	`i_srcaddr`	Section source (ROM) address. N/A for sections of uninitialized data (contains 1 in this case).
unsigned long	4	`i_trgaddr`	Section target (RAM) address

The linker also defines the symbol **_INIT_TABLE**. This symbol is assigned the address of the initialization table.

The C structure declaration for this table may be found in the **inittab.h** header file supplied with the CompactRISC package. The initialization table generated by the linker resides in a linker-created **.init** input section (see Section 4.2.4).

Example Consider a simple C program that uses both initialized and uninitialized variables:

```
int i = 5;
int j;
main()
{
     if ( i == 5 && j == 0)
     printf("PASSED \n");
     else
     printf("FAILED \n");
}
```

The initialized variables are directed to **.data_1 .data_2** and **.data_4** sections according to their size. Uninitialized variables are directed to **.bss_1 .bss_2** and **.bss_4** sections. To run this program correctly in an embedded environment, data initialization is necessary. The contents of the **.data_*n*** sections should reside on ROM and copied to RAM at the beginning of the program execution. The **.bss_*n*** sections can be filled with zeroes. Use the linker data initialization support to simplify this process. A sample directive file for linking the program is:

```
          MEMORY {
               ROM : ORG = 0x80000 LEN = 0x20000
               RAM : ORG = 0x10000 LEN = 0x40000
          }

          SECTIONS {
               .text INTO(ROM) : { }
               .rdata ALIGN(2) INTO(RAM):
                         { *(.rdata_4) *(.rdata_2)
*(.rdata_1)}
               .data ROMINTO(ROM) INTO(RAM) :
                         { *(.data_4) *(.data_2) *(.data_1)}
               .bss INTO(RAM) :
                         { *(.bss_4) *(.bss_2) *(.bss_1)}
               .init INTO(ROM) : { }
          }
```

The **.text** section that contains the program code is directed to the ROM memory area. The **.data** section (which is combined from all **.data_*n*** sections) is duplicated, and directed to the RAM and ROM memory areas.

Note that all references to the **.data** section are to the RAM copy. The ROM copy is used only for initialization purposes and is not used after initialization is completed.

The **.bss** section (which is combined from all the **.bss_*n*** sections) is directed to the RAM memory area. The **.init** section, which contains the initialization table, is directed to the ROM memory area.

The program is now ready for linking. The **start** routine, provided as part of the CompactRISC **libadb** library, can be used to perform the initialization. The invocation line for the above program is:

```
crlink main.o -ladb -lc -lvio -o prog
```

The **.data** section should now be burned to ROM using the **crprom** utility (see Chapter 6). A sample invocation of **crprom** (assuming 64K EPROMs) is:

```
crprom -x0x8000 -l64 prog -o prog.hex
```

Using the initialization routines defined in **libadb**, you can fully control the initialization process by calling the specific initialization routine needed. To eliminate all the initialization, simply comment out the call to the initialization routine in your start routine.

Normally, an input section is a part of an input file. However, for data initializations the linker creates a "dummy" input section called **.init**. The **.init** dummy input section is added to the input section list, and participates in the allocation process like any other input section. The **.init** section can be identified in the linker memory map by the term **linker_defined** that replaces the filename.

4.2.5 Code Overlay Allocation

The CompactRISC linker supports a code overlay mechanism. You can write several code sections in the same output section. This allows several time-critical code segments to run from a single, small, fast memory (e.g., SRAM), while the entire program resides in a large, slow, memory (e.g., DRAM).

The **-O** flag informs the linker to enable the code overlay mechanism. (See Section 2.3.18)

A special library function, **cp_overlay_code**, copies the code segments into the fast memory. The application must call this function before executing the code. This function is included in the Application Development Board library (**libstart** for version 2.0, or **libadb** for version 1.2).

Example

Consider an application that has two critical functions, f1 and f2 (defined in the files **c1.o** and **c2.o**, respectively), and a 0x500 bytes of fast-memory, starting at address 0x1000.

You should use the following linker definition file:

```
MEMORY {
          ...
     CODE_MEM:
          ...
```

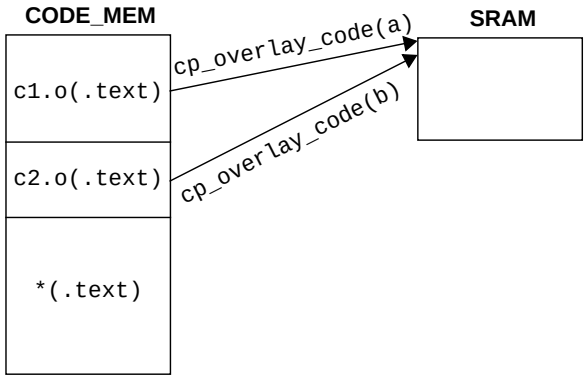
```
     SRAM:     origin = 0x1000, length = 0x500
          ...
}

SECTIONS {
          ...
     .text BIND(0x1000) ROMINTO(_a=.; , CODE_MEM) :
{c1.o(.text)}
     .text BIND(0x1000) ROMINTO(_b=.; , CODE_MEM) :
{c2.o(.text)}
     .text INTO(CODE_MEM) : {*(.text)}
          ...
}
```

The application code that calls these functions should be:

```
     #include <libstart.h> //for old tmon users <libadb.h>
     extern void a();
     extern void b();
     ...
     application()
     {
          ...
          cp_overlay_code(b);
          f2();
          ...
          cp_overlay_code(a);
          f1();
          ...
          cp_overlay_code(b);
          f2();
     }
```

In this example, the linker assigns the source address of the code section, **c1**, to the symbol **a**, and the source address of the code section, **c2**, to the symbol **b**. These symbols are used as parameters to **cp_overlay_code()**.

The size of any critical code, defined with the code overlay mechanism, must not exceed the size of the fast memory.



4.2.6 Bank Switching

The CR16A core enables accesses to 128KB code address space. The Bank Switching mechanism enables execution of applications larger than 128KB, on a CR16A-based chip.

Notes

The Bank Switching mechanism, for a CR16A-based chip, requires additional on-chip hardware.

The CompactRISC debugger does not support Bank Switching.

The Bank Switching mechanism provides virtual linear code address space of size n * 128KB (where n is the number of banks). The application resides in a sequence of 128KB external ROM segments (banks). The Bank Switching mechanism enables mapping the code address space of the CR16A core to one bank at a time. A special hardware register, on the chip, selects the number of the currently mapped bank.

While compiling the C code for a Bank Switching application, you must specify **-mbank** to the compiler. While linking a Bank Switching application, you must specify **-BS** to the linker.

Mediator function

A mediator function enables the linker to make a cross-bank function call (i.e., the caller function is located in one bank and the called function is located in another). If the linker encounters a cross-bank function call, it generates a mediator function. The linker replaces the cross-bank function call with a call to the mediator function.

The linker generates one mediator function for each cross-bank called function in the application. It allocates the code for the mediator functions to a code segment that exists in all the banks. This code segment is called common-memory, and is described below.

For example, consider a cross-bank function call: **f()** function, in one bank, that calls **g()** function, in another bank. The linker generates a mediator function, **m_g()**, for the cross-bank called function **g()**, and replaces the call to **g()** with a call to **m_g()**. The mediator function, **m_g()**, enables this cross-bank function call:

f() prepares parameters for **g()**, as usual.

f() calls to **m_g()** (the mediator function of **g()**).

m_g() loads the address of **g()** (the bank number for **g()** and its address in this bank).

m_g() saves the return address (the current bank number and the address for **f()** in this bank).

m_g() switches to the bank containing **g()** (i.e., writes this bank number to the dedicated hardware register).

m_g() calls **g()**.

and after **g()** returns:

m_g() switches back to the bank containing **f()**.

m_g() returns to **f()**.

A mediator function is composed of two parts. One part loads the called function's bank and address. This part is specific for every cross-bank called function. The second part is fixed for all the mediator functions.

Common-memory

A Bank Switching application must have a code segment that is always visible, i.e., copied to all the banks. This common-memory includes interrupt handlers, ROM data (**.rdata** sections), and the code for the mediator functions (generated by the linker).

You must define the common-memory, in the **linker.def** file, (see the example below). The length field contains the size of the common-memory.

You must also direct the linker, using the **linker.def** file, to allocate all the always visible code and data (e.g., interrupt handlers and **.rdata** section) to the common-memory (see the example below).

In the remaining part of the common-memory, the linker allocates the code for the mediator functions.

The size of the common-memory is calculated by adding the size of what you have allocated to the common-memory (interrupt handlers, **.rdata** sections) to the code size of all the mediator functions (generated by the linker).

The code size (in bytes) of all the mediator functions is (N * 12) + 36

where:

N is an estimate of the number of the cross-bank called functions in the application.

12 is the maximum size of the specific part of a mediator function, and 36 is the size of the fixed part of all the mediator functions.

Example

Consider an application with three modules (**a.o**, **b.o**, **c.o**) that we want to allocate to three banks. The MEMORY part of the **linker.def** file should have the form shown below (where the addresses are just for example):

```
MEMORY {
     common_memory:
          origin=0x10000,        length=0x2000

     bank0_code_mem:
          origin=0x12000,        length=0xC000

     bank1_code_mem:
          origin=0x32000,        length=0xC000

     bank2_code_mem:
          origin=0x52000,        length=0xC000

          .
          .
          .
}
```

Note

Define the common-memory area in the first bank (0 - 0x1FFFF). The linker copies the common-memory to the corresponding addresses in the higher numbered banks. (In the above example, to addresses 0x30000 - 0x32000 in the second bank, and to addresses 0x50000 -0x52000 in the third bank). Do not allocate anything to these addresses.

To allocate the interrupt handlers' code and **.rdata** sections to the common-memory, and the three modules to the three banks, the SECTION part of the **linker.def** file should have the form:

```
SECTIONS {


.text ALIGN(2) INTO(common_memory): { int1.o(.text) int2.o(.text) ... }
.rdata ALIGN(2) INTO(common_memory): { *(.rdata_2) *(.rdata_1) }
          .
          .
          .
.text ALIGN(2) INTO(bank0_code_mem): { a.o(.text) }
.text ALIGN(2) INTO(bank1_code_mem): { b.o(.text) }
.text ALIGN(2) INTO(bank2_code_mem): { c.o(.text) }

}
```

Note You can allocate the interrupt handlers to the common-memory *only* if they do not reside in the same modules as the rest of the application.

To define the address of the bank number register (in the **`linker.def`** file):

```
BANK_NUM_REG =0x...; (address of the special register).
```

Limitations Cross-bank function calls through a pointer are not supported.

A module can not cross bank boundaries, branches between modules are only allowed as function calls.

A cross-bank called function must not return structure as a return value.

4.2.7 Memory Map

The following information appears on the memory map (all address and size values are in hexadecimal):

Output section lists each output section in the order it appears in memory. For those output sections that have been duplicated, the ROM copy is denoted by (R) next to the output section name.

Input section lists each input section that was linked to produce the specified output section.

Memory address denotes the starting address in memory of a particular input or output section.

Size lists the total size of the output section and the individual size of each input sections.

Section contents specifies the input file from which the input section originated. This is either an object file or a library file. The term "linker_defined" indicates that the section was created by the linker.

The term “fill space” may appear in the section contents column, and indicates a gap created in the output section through use of the current location symbol assignment of the directive file (Section 3.8.2). The term “UNUSED” refers to unallocated or unconfigured memory.

Example

```
output     input      memory    size   section
section    section    address          contents
.rdata                0         2c
           .rdata_2   0         2c     libadb.a:intable.o
.data                 2c        132
           .data_2    2c        72     libc.a:xfiles.o
           .data_2    9e        c0     libvio.a:_util.o
.bss                  15e       120
           .bss_2     15e       42     libc.a:atexit.o
           .bss_2     1a0       4      libc.a:malloc.o
           .bss_2     1a4       2      libc.a:sbrk.o
           .bss_2     1a6       2      libc.a:xinit.o
           .bss_2     1a8       2      libc.a:errno.o
           .bss_2     1aa       84     libvio.a:_util.o
           .bss_1     22e       50     libc.a:xfiles.o
.init                 27e       1c
           .init      27e       1c     linker_defined
.stack                29a       2000
                      29a       2000   fill_space
.istack               229a      100
                      229a      100    fill_space
UNUSED     239a to    d800      b466
UNUSED     d800 to    db00      300
UNUSED     db00 to    e000      500
.text                 10000     8ca
           .text      10000     e      main.o
           .text      1000e     20     libadb.a:start.o
           .text      1002e     6e     libadb.a:init_all.o
           .text      1009c     12     libadb.a:bpt.o
           .text      100ae     12     libadb.a:dvz.o
           .text      100c0     12     libadb.a:flg.o
           .text      100d2     12     libadb.a:und.o
           .text      100e4     12     libadb.a:nmi.o
           .text      100f6     12     libadb.a:svc.o
           .text      10108     12     libadb.a:trc.o
           .text      1011a     12     libadb.a:ise.o
```

```
                .text       1012c     14      libadb.a:int_hnd.o
                .text       10140     14      libadb.a:int_hnd2.o
                .text       10154     4c      libc.a:exit.o
                .text       101a0     22      libc.a:_exit.o
                .text       101c2     b8      libc.a:fclose.o
                .text       1027a     e       libc.a:remove.o
                .text       10288     82      libc.a:fflush.o
                .text       1030a     86      libc.a:free.o
                .text       10390     22      libc.a:atexit.o
                .text       103b2     104     libc.a:malloc.o
                .text       104b6     1a      libc.a:xgetmem.o
                .text       104d0     7a      libc.a:sbrk.o
                .text       1054a     7c      libvio.a:close.o
                .text       105c6     ce      libvio.a:unlink.o
                .text       10694     180     libvio.a:write.o
                .text       10814     6e      libvio.a:_util.o
                .text       10882     1a      libvio.a:do_viopk.o
                .text       1089c     18      libvio.a:memcpy.o
                .text       108b4     16      libvio.a:strlen.o
.data           (R)         108ca     132
                .data_2     108ca     72      libc.a:xfiles.o
                .data_2     1093c     c0      libvio.a:_util.o
UNUSED          109fc   to      20000 f604
```

4.3 RELOCATION OF MEMORY ADDRESS

Once external symbolic references have been resolved, and output sections allocated to memory, the linker calculates the final addresses of symbolic references. It also modifies the contents of the holes within the code or data. Holes are small pieces of code, or data, that are based on symbolic references. A hole must be modified to reflect the new address of the symbolic reference on which it is based. This process includes not only external symbolic references, but also other symbolic references that must be updated (e.g., a reference that uses absolute addressing).

4.3.1 Relocation Information

Relocation information is part of the COFF file. The assembler creates a relocation table for each section of the file. Each relocation table entry provides information about a hole that should be updated as a result of link-time section allocation. This hole may be a machine instruction operand or data (typically address constants, etc.).

Each relocation entry consists of:

- The address of the hole that should be updated.
- A pointer to the symbol that is associated with the reference.
- The hole type (format, size, and semantic).

Consider the following assembly program:

Example

```
bal   r2, foo
storw r0,abc
.data
abc: .double 5
```

The first two instructions must be relocated at link-time.

The first instruction, **`bal r2, foo`**, refers to an external symbol (**`foo`**) whose address is unknown at assembly-time. In the assembler encoding of this instruction, the place that should contain **`foo`**'s address is initialized to 0. (All the encodings used throughout this section are for the CR16).

```
9e 92 00 00 00 00 (bal r2,foo)
```

The relocation entry created by the assembler contains the following information:

- A hole address that points to the instruction's second byte (the location of the instruction operand).

- The index of the symbol **foo** in the symbol table.
- The hole type (the hole size is 4 bytes, and the hole is pc-relative).

The second instruction, **storw r0, abc**, has a reference to the local symbol abc. However this reference is in absolute addressing mode, and absolute addresses are unknown at assembly time. The assembler encoding of this instruction includes the offset of **abc**'s address from the beginning of the section.

```
d9 0f 0c 00 00 00 00(storw r0,abc)
```

The relocation entry created by the assembler contains the following information:

- A hole address that points to the instruction's third byte (the location of the instruction's second operand).
- The index of the symbol **.data** in the symbol table. The symbol **abc** is already allocated in the **.data** section and its final address is updated according to the final address of the **.data** input section. The address of the symbol **.data** is also the address of the **.data** input section.
- The hole type (hole size is 4 bytes, and the hole is an absolute address).

4.3.2 The Relocation Process

During the relocation process, the linker scans the relocation table of each input section. For each relocation entry, the linker calculates the new value of the hole. The calculation is based on the referenced symbol's new address, on the new address of the hole (if the hole is a pc-relative operand), and on the type of the hole found in the relocation entry. The new value of the hole must fit the size allocated to it, otherwise the linker issues an error message. After the new value of the hole has been calculated, the linker updates the hole with this new value.

Example

Using the example in Section 4.3.1, two holes are modified. Assume that the final address of the symbol **foo** is 0x9000, and that the final address of the **bal** instruction is 0x8000. The displacement for the **bal** instruction should thus be modified to 0x9000-0x8000=0x1000. The linker modifies the operand, and the new instruction encoding includes the correct address.

Assume that the final address of the symbol **abc** is 0xf000. The second operand of the **storw** instruction is modified to this absolute address.

Chapter 5
THE ARCHIVER

5.1 INTRODUCTION

The archiver, **crlib**, maintains groups of object files combined into a single archive (library) file. Its main use is to create and update library files for use by the linker. The archive's object file are called members.

5.2 CREATING ARCHIVE FILES

When **crlib** creates an archive, it generates a symbol directory for it. The linker **crlink** uses the archive symbol directory to search object files, which define unresolved symbols, in an efficient manner. An archive symbol directory is created and maintained by **crlib** only when there is at least one object file in the archive. Whenever the **crlib** command is used to create or update the contents of such an archive, the symbol directory is rebuilt.

5.3 INVOCATION AND USAGE

crlib ***key*** **[** ***modifier*** **]** ***afile filename ...***

The minus "—" is optional to ***key***, followed by one character from the set **d**, **r**, **q**, **t**, **p**, **V**, **m**, or **x**, optionally concatenated with one or more modifiers from the set **v**, **u**, **c**, **l** and/or **s**. ***afile*** is the archive file. ***filenames*** are the names of archive members, or of object files, to be added to the archive. The meanings of the ***key*** characters are as follows:

d Deletes the named files from the archive file.

r Replaces the named files in the archive file. If the optional character **u** is used with **r**, only those files with modification dates later than the archive files are replaced. If the archive does not already exist, it is created. If the file is not already present in the archive, it is placed at the end.

q Quickly appends the named files to the end of the archive file. This command does not check whether the added members are already in the archive.

t Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

m Moves the named files to the end of the archive.

x Extracts the named files. If no names are given, all files in the archive are extracted. In neither case does ***x*** alter the archive file.

V Prints the version number of the **crlib** program in use.

The meanings of the modifier characters are as follows:

v Gives a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, gives a long listing of all information about the files. When used with **x,** precedes each file with a name.

c Suppresses the message that is produced by default when ***afile*** is created.

n Suppresses generation of symbol directory, thus shortening **crlib** processing time. This command is useful when incrementally creating large libraries by several calls to **crlib**. When used, all calls should be made with the **-n** option, except for the last call.

s Forces the regeneration of the archive symbol directory, even if **crlib** is not invoked with an option which modifies the archive contents. This command is useful to restore the archive symbol directory of an archive created using the **-n** option.

Example

```
crlib x libc.a printf.o
```

In this example the member **printf.o** is extracted from library **libc.a**.

@argfile Reads **crlib** arguments from the file ***argfile***. The **@** option directs the CompactRISC archiver to read additional command-line arguments from the named file. This option overcomes the limitation on the length of invocation lines, which exists under some environments.

Chapter 6
THE EPROM FILE GENERATOR

6.1 INTRODUCTION

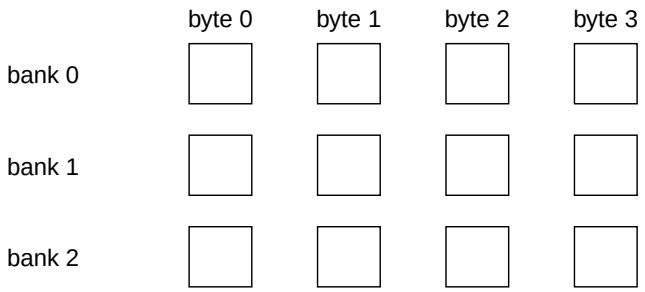
The **crprom** utility is used to convert data from CompactRISC executable object files into an EPROM programmer format. Three formats are supported: ASCII-hex (default), Intel-hex and Motorola. The output is directed into separate files, one for each EPROM. If no output file name is specified, the output is directed to the auxiliary (printer) port of a VT100 compatible terminal; the EPROM programmer must then be attached to the auxiliary port.

6.2 GENERATING EPROM FILES

Each EPROM holds part of the data bytes of a memory block, depending on the system bus-width (or word size). If the system bus-width is 2, the EPROMs come in pairs. In each pair one EPROM holds the even address data bytes and the other EPROM holds the odd address data bytes. Together they form a bank. If the system bus-width is 4, the EPROMs come in quartets: one EPROM holds the first data bytes of each word, another EPROM holds the second data byte of each word and so on. These four EPROMs again form a bank.

The **crprom** utility allows you to create output for one or more EPROMs in various ways. The ROM area is regarded by the **crprom** utility as a two-dimensional matrix of EPROMs. One dimension is the EPROM byte number (i.e., which byte in the word the EPROM holds). The other dimension is the bank number, since the **crprom** utility can produce output for several consecutive banks.

The figure below illustrates a two-dimensional matrix of EPROMs:



Using the **crprom** utility, you can specify each EPROM in the matrix, and specify each row or column of EPROMs. You can also specify the complete matrix in order to produce output for the entire ROM area in a single invocation of **crprom**. If you direct the output to a file(s), **crprom** creates one output file for each EPROM.

6.3 INVOCATION AND USAGE

crprom ***[options] [filename]***

filename is a **crprom** input file. This should be a CompactRISC executable object file produced by the CompactRISC Linker, **crlink**. The default input file name is **cr.x**.

The **crprom** invocation options are listed below. All integer constants can be specified in C syntax (i.e., decimal, hexadecimal, or octal).

-w*size* Specifies the bus-width (word size). ***size*** can be: 1, 2, 4, 8, 16 or 32. If this option is not specified, a default bus-width of 2 bytes (16 bits) is used.

-b*bytenumber* Selects an EPROM in a bank. The EPROM is denoted by the byte number of the word. ***bytenumber*** can have any value in the range of 0 to ***buswidth***-1 (***buswidth*** is the system bus-width). By default all EPROMs in the bank are selected.

-x*address* Specifies the start address of the first bank. ***address*** is an integer constant.

-l*size* Specifies the size of the EPROMs in use. ***size*** is specified in Kbytes (i.e., the EPROM size is 1024****size***). Default is 64 Kbytes.

-k[***bank#:***]***num***
Specifies which banks should be selected. If two values are specified in the form ***bank#:num, num*** banks are selected starting from bank ***bank#***. If only one value is specified, ***num*** banks are selected starting from the first bank (bank 0). By default **crprom** selects only the first bank.

-o *filename* Specifies the output filename. Since **crprom** can produce several output files, ***filename*** is generally used as a generic name. For each output file, **crprom** adds the extension ***_banknum_bytenum*** to the file name (unless the **-n** option is specified). ***banknum*** is the EPROM bank number, ***bytenum*** is the EPROM byte number.

If this option is not specified, **crprom** directs the output(s) to an auxiliary port of a VT100-compatible terminal. Attach the EPROM programmer to this port. Before writing the output for each EPROM, **crprom** prints a message to the terminal and waits for you to plug in the EPROM.

-n Specifies that a filename extension is not required. This is useful when only one output file is to be produced. The output file name in this case is as specified by the **-o** option.

-p*offset* Specifies an EPROM offset to which the data is to be directed. By default the offset is 0. This option applies only to EPROMs of the first bank.

-i Specifies **crprom** output in Intel-hex format. This format supports EPROMs of up to 1 Mbytes. The default format is ASCII-hex, which supports EPROMs of up to 64 Kbytes.

-m{1|2|3} Specifies **crprom** output in one of the three Motorola formats:

format 1 - supports EPROMs of up to 64 Kbytes;
format 2 - supports EPROMs of up to 16 Mbytes;
format 3 - supports EPROMs of up to 4 Gbytes.

The default format is ASCII-hex, which supports EPROMs of up to 64 Kbytes.

-r Specifies that only sections that typically reside in ROM (i.e., sections of type **STYP_TEXT** and **STYP_DATA**) are extracted from the executable object file. By default, selects any loadable section that resides in the EPROM address range.

-c Specifies that a checksum byte is added for each EPROM. This option is used if the contents of the EPROM are to be verified at run-time (e.g., for diagnostics purposes). The checksum byte is located in the first unused byte of the EPROM. The checksum byte value is calculated such that the xor of the one's complement of all bytes (including the checksum byte) is 0.

Unused bytes are set by EPROM programmers to 0xFF, and therefore do not affect the checksum. If all bytes of the EPROM are occupied, **crprom** issues a warning and does not add a checksum byte.

Examples

1 **crprom -x0x10000 -o out execfile**

This command creates one bank of EPROMs located at address 0x10000. The word size is the default = 2. **crprom** produces two output files, one for each EPROM:

out_0_0 out_0_1

The EPROM size is 64K (default size). The output format is ASCII-hex (default format).

2 **crprom -w2 -b1 -x0x20000 -l32 -i -o OUT execfile**
This command creates the second EPROM in a bank located at address 0x20000. The word size is 2 (i.e., there are two EPROMs in each bank, 0 and 1). The byte number is 1, signifying that the second EPROM is required. The EPROM size is 32K and the output format is intel-hex. Only one output file is produced:

```
out_0_1
```

3 **crprom -w2 -x0x10000 -k2 -m2 -o out execfile**
This command creates two consecutive banks of EPROMs. The start address of the first bank is 0x10000. Each bank has two EPROMs because the word size is 2. Since there are four EPROMs, the following four output files are produced:

out_0_0 (first bank, first EPROM)

out_0_1 (first bank, second EPROM)

out_1_0 (second bank, first EPROM)

out_1_1 (second bank, first EPROM)

EPROM size is 64K (default). The output format is Motorola format number 2.

4 **crprom -w4 -k1:1 -i execfile**
This command creates the second bank (bank number 1) of a set of consecutive banks of EPROMs. The address of the first bank is 0 (default). However, since the second bank is required, data is taken from the start address of the second bank. This address is equal to the size of one bank (i.e., the EPROM size 64K, multiplied by four EPROMs in each bank, gives 256K or 0x40000). Because no output file is specified, the output is directed to the terminal's auxiliary port where the EPROM programmer is attached. Before writing the output for the first EPROM, **crprom** prints the following message:

```
Creating output data for bank 1 byte number 0
Press RETURN when ready...
```

A similar message is printed for each other EPROM. Plug the appropriate EPROM into the programmer before you press the RETURN key.

6.4 THE INTEL FORMAT

The Intel 16-bit Hexadecimal Object file record format, MSC-86 Hexadecimal Object code 88, is the default command used when converting data using crprom utility. It has a 9-character (4-field) prefix that defines the start of record, byte count, load address, and record type and a 2-character checksum suffix. For example:

```
     :020000020000FC
:0860000000000000001000000088
:106008009FFFFF0AEFFFFF020008F28204A55F9FCF
:10601800FFFF0A008BA0FFFF480A007D09A100E0EE

     :    :

     :1074A000000000000000000000000000000000DC
:1074B00000000000A00100010000010100000000028
:00000001FF
```

The four record types are described below:

6.4.1 00 - Data Record

This begins with the colon start character, which is followed by the byte count (in hex notation), the address of the first data byte, and the record type (equal to 00). Following these are the data bytes. The checksum follows the data bytes and is the two's compliment (in binary) of the preceding bytes in the record, including the byte count, address, record type and data.

6.4.4 03 - Start Record

This record type is not sent during output by Data I/O translator firmware.

Chapter 7
THE OBJECT FILE VIEWER

7.1 INTRODUCTION

The **crview** utility displays, in a formatted manner, all the information that exists in a CompactRISC object file.

Input files for **crview** may be object files (executables or not) and archive (library) files.

7.2 INVOCATION AND USAGE

crview [***options*** **]** ***filename ...***

If no options are specified **crview** displays all parts of the object file(s).

The options are available:

-h display file headers

-n display section headers

-r display relocation information

-l display line number information

-s[***sub-option***]
display symbol table (long format)

-S[***sub-option***]
display symbol table (short format: one line per symbol)

Possible sub options for **-s** and **-S** options:

n - sort symbols according to their names
v - sort symbols according to their values
u - display only user-defined symbols
g - display only global and static symbols
x - display symbol values in hexadecimal format

If no sub-option is specified, all the symbols in the symbol table are displayed in the same order as they appear in the symbol table, and their values are displayed in decimal format (with **-S**) and in both decimal and hexadecimal formats (with **-s**).

-a display archive symbol directory

-T disassemble **.text** section(s) contents

-D display **.data** section(s) contents in dump format

-I display **.init** section contents in an initialization table format

-O display any section contents in dump format

-e ignore disassembly errors

-f[***sub-option***]
displays function information.
Possible sub-options are:
c - Class
t - Type
n - Name
a - Arguments
b - Begin address
e - End-address
z - Size
k - Stack
s - Source file
h - Header
If no sub-options are used for the **-f** option, the whole function informations is displayed.

-z displays sections and their sizes.

Appendix A
LINKER ERROR MESSAGES

A.1 INTRODUCTION

This appendix contains a list of all linker error messages. There are five types of error messages:

- System Error - the result of an incorrect call to the operating system. A system error causes immediate termination of the linking process. An explanation of the error follows the error message.
- Warning - A warning error has no impact on the linking process.
- Severe Error - Severe errors accumulate, and eventually result in the termination of the linking process.
- Fatal Error - A fatal error causes an immediate termination of the linking process.
- Internal Error - An internal error is caused by an internal problem in the linker. If you encounter an internal error please contact National Semiconductor immediately. Internal error messages are not listed in this appendix.

The error messages are listed in alphabetical order. The error message type and an explanation is also given.

A.2 ERROR MESSAGES

Cannot close file *filename*

Type: System

Explanation: An error has been detected when closing the file.

Cannot open default directive file *filename*. Proceeding with default linker processing

Type: Warning

Explanation: The default directive file cannot be opened. The linker therefore uses a default allocation process, which assumes that the maximum amount of configured memory space is available and allocates output sections contiguously from address 0.

Cannot open specified directive file *filename*

Type: System

Explanation: The user-specified directive file cannot be opened.

Cannot open input file *filename*

Type: System

Explanation: The input object or library file cannot be opened.

Cannot open output file *filename*

Type: System

Explanation: The output file cannot be opened with write permission.

Common symbol *symbolname* has an explicit definition

Type: Warning

Explanation: The common symbol which appears in one or more input files is defined in another input file. This is not an error. All references to the symbol are resolved according to its definition. Common symbols without a definition are consolidated and allocated memory space by the linker.

Common symbol *symbolname* multiply declared with differing sizes. Larger size used

Type: Warning

Explanation: The references to the symbol in various input files have differing size specifications. The linker uses the largest size specified for allocation.

Entry point not found. To use default entry point "start" link with libadb.

Type: Fatal

Explanation: No entry point was specified in the invocation line and the default entry point "start" was not found. To use the default start routine the application must be linked with libadb.a

---

Directive file MEMORY statement error: overlapping memory areas *mem1* and *mem2*

Type: Fatal

Explanation: An error has been detected in the memory configuration as specified in the directive file **MEMORY** statement. Two memory areas overlap.

Directive file BIND/ROMBIND option error: address *addr* (specified for output section *secname*) is already allocated to another output section

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The specified argument address of the **BIND/ROMBIND** option is not available since it is already allocated to another output section.

Directive file BIND/ROMBIND option error: address *addr* (specified for output section *secname*) is not in configured memory

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The specified argument address of the **BIND/ROMBIND** option is not available since it is not within configured memory.

Directive file BIND/ROMBIND option error: output section *secname* does not fit at specified address *addr*

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The output section does not fit at the address specified by the **BIND/ROMBIND** option since there is not enough unallocated memory space at this point.

Directive file BIND/ROMBIND option is used for output section *secname* This overrides any other allocation option

Type: Warning

Explanation: Both a directive file **BIND/ROMBIND** option and another allocation option are specified for the output section. The linker ignores the other allocation option, since **BIND/ROMBIND** options have the highest priority.

Directive file INTO/ROMINTO option error: cannot direct output section *secname* to a memory area - no memory area with the specified attributes was defined

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The output section is directed by attributes to a memory area but no memory area with the requested attributes was defined in the **MEMORY** statement.

Directive file INTO/ROMINTO option error: memory area *mem*, specified for output section *secname*, is undefined

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The output section is directed to a named memory area which was not defined in the **MEMORY** statement.

Directive file INTO/ROMINTO option error: output section *sec-name* (size *size*) cannot fit in any memory area having the specified attributes

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The output section is directed by attributes to a memory area but there is not enough space left for this output section in any memory area that has the specified attributes.

Directive file INTO/ROMINTO option error: output section *sec-name* (size *size*) does not fit in memory area *mem*

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The output section is directed to a named memory but there is not enough space left for this output section in the memory area.

Directive file error: opt option illegal within GROUP

Type: Severe

Explanation: This error indicates that one of the directive file options: BIND/INTO/BLOCK was used within a GROUP. These options may only be used for the whole group.

Directive file error: input file *filename* specified inside SECTIONS statement not found in input file list

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The file name specified as part of an input section specification is not in the input file list. The input file list includes all input files specified in the invocation line and in the directive file before the **SECTIONS** statement.

Directive file error: integer constant *int* too big near line num

Type: Fatal

Explanation: The directive file contains an invalid integer constant. Integer constants must be in the range 0 - (2^{32})-1.

`Directive file error: invalid memory attribute att specified. Will be ignored`

Type: Warning

Explanation: The directive file contains an invalid memory attribute letter. This is ignored by the linker.

`Directive file error: current location symbol (.) decremented inside output section` *`secname`* `specification`

Type: Fatal

Explanation: The directive file **SECTIONS** statement contains an error. The current location symbol (**.**) is used incorrectly in the output section specification, such that its new value is less than its old value. The current location symbol may never be decremented.

`Directive file error: current location symbol (.) used not inside SECTIONS statement, near line` *`linenum`*

Type: Fatal

Explanation: The current location symbol (**.**) is used outside the **SECTIONS** statement of the directive file. The current location symbol may be used only in an assignment statement that appears as part of an input section list in a **SECTIONS** statement.

`Directive file error: illegal character 'char' found near line linenum - ignored`

Type: Warning

Explanation: An invalid integer constant is specified as an argument for the directive file **OPTION OMAGIC** statement. The argument should not exceed 16 bits (greater than 65535).

`Directive file error: optional header magic number, specified for OPTION OMAGIC, exceeds 16 bits`

Type: Fatal

Explanation: An invalid integer constant is specified as an argument for the directive file **OPTION OMAGIC** statement. The argument should not exceed 16 bits (greater than 65535).

Directive file error: optional header magic number, specified for OPTION OMAGIC, exceeds 16 bits

Type: Fatal

Explanation: An invalid integer constant is specified as an argument for the directive file **OPTION OMAGIC** statement. The argument should not exceed 16 bits (greater than 65535).

Directive file error: specified MEMORY area mem exceeds 32 bit address range

Type: Fatal

Explanation: The directive file **MEMORY** statement contains an error. The memory area is specified with an address range that exceeds the processor address range.

Directive file expression error: output section *secname* used as an argument for SIZEOF, FILEADDR or ADDR function, is not found

Type: Fatal

Explanation: The directive file expression contains an error. The output section that was specified as an argument to the **SIZEOF, FILEADDR** or **ADDR** function is not found.

Directive file expression error: symbol *symbolname* not found

Type: Fatal

Explanation: The directive file assignment statement contains an error. The symbol that was specified as part of the right-hand-side of an assignment statement is not found.

Directive file parse error: err near line number *linenum*

Type: Fatal

Explanation: The directive file contains a parsing error. This is typically a syntax error. The parsing error type is denoted by ***err***.

`Directive file syntax error: ending quote expected near line` *`num`*

Type: Fatal

Explanation: The directive file contains a syntax error. A string is missing the end quote.

`Directive file syntax error: unrecognized keyword after OPTION near line` *`linenum`*

Type: Fatal

Explanation: The directive file contains a syntax error. An invalid keyword is specified after the **`OPTION`** keyword.

`Dynamic memory allocation failed (`*`num`*`m bytes required)`

Type: Fatal

Explanation: A system request for the dynamic allocation of ***`num`*** bytes has failed.

`Entry point is not specified and default entry point symbols (start or _main) cannot be found. Entry point will be set to 0.`

Type: Warning

Explanation: The linker failed to determine the program entry point either because it was not specified explicitly through the Specify Entry Point invocation option, or the symbols that mark the entry point by default (**`start`** or **`_main`**) were not defined in any input file. Therefore the linker will set the entry point to the address 0.

`Input file` *`filename`* `is not in proper COFF format (bad magic number)`

Type: Fatal

Explanation: The magic number is incorrect. The magic number resides in the first two bytes of any object files. If these two bytes contain an invalid value, the file is not recognized as a COFF file and is not processed by the linker.

Instruction operand or address constant cannot fit in space after relocation. Reference to symbol *symbolname* (index *num*) from section *secname*, in *filename*

Type: Severe

Explanation: A piece of code or data (hole), based on a symbolic reference, cannot be modified because its value after relocation does not fit its space. This may be the result of using byte/word displacements or absolute addresses.

Input section not found for symbol *symbolnam*e in *filename*

Type: Severe

Explanation: *symbolnam* not found in any input section of file *filenam*e

---

Instruction operand or address constant truncated after relocation. Reference to symbol *symbolname* (index *num*) from section *secname*, in *filenameddr, relocation addresss*: addr.

Type: Severe

Explanation: A piece of code or data (hole), based on a symbolic reference, was relocated, after relocation its value does not fit its hole size. The value was therefore truncated.

Integer constant int, specified as an argument to an invocation option, is too big

Type: Warning

Explanation: An argument to the Specify Fill Value or Specify Version Stamp invocation option is out of range. The argument must be in the range 0-2**16-1.

Invalid integer constant *int* specified as an argument to an invocation option

Type: Fatal

Explanation: An invalid integer constant value is used for the invocation options Specify Fill Value or Specify Version Stamp. This is either an invalid integer constant specification or the integer constant is not within the legal range (greater than 65535).

Invocation option *opt* requires an argument

Type: Fatal

Explanation: The invocation option is specified without the required argument. Note that some invocation options may require a space before the argument.

Library file *filename*, specified with -l invocation option, not found

Type: Fatal

Explanation: The library file specified with the -l invocation option is not found in any directory in the library search path.

Library file *filename* has no symbol directory. The GNX archiver utility may be used to restore it

Type: Warning

Explanation: The input library file does not contain a symbol directory. This file therefore cannot be processed. The symbol directory may not exist because the library file has been stripped. The CompactRISC archiver (n)ar may be used to rebuild the symbol directory.

More than one directive file allocation option is used for output section *secname*. Using allocation option priority rules.

Type: Warning

Explanation: The **SECTIONS** statement of the directive file has more than one allocation option specified. The linker uses the allocation option with the highest priority.

More than one directive file specified

Type: Fatal

Explanation: More than one directive file is specified. Only one directive file can be specified as linker input.

Multiply defined symbol *symname*, defined in *filename1* already defined in *filename2*

Type: Fatal

Explanation: The symbol has more than one definition. A symbol may have only one definition.

No input object files specified

Type: Fatal

Explanation: No input object files have been specified for the linking process.

Object files being linked are not all the same core code

Type: Fatal

Explanation: The object files being linked have code for different core architectures and therefore cannot be linked to create one executable.

Output section *secname* (size *size*) cannot fit in remaining unallocated configured memory

Type: Fatal

Explanation: Refers to an output section without an allocation option specified in the directive file. The linker is unable to fit the output section into the remaining unallocated configured memory since there is not enough space left for it.

Read error on file *filename*

Type: Fatal

Explanation: An error was detected when trying to read from the file.

Seek error on file *filename*

Type: System

Explanation: An error was detected when trying to perform a seek operation on the file.

Specified undefined symbol *symbolname* never resolved

Type: Severe

Explanation: The definition of the symbol is not found in any of the input object files or in the directive file.

_ _STATIC_BASE_START is not initialized properly, in relocation entry *entry* of input section *section* in *filename*

Type: Fatal

Explanation: An sb-relative symbol is used but the static base start symbol is not initialized.

Unable to recover from previous errors

Type: Fatal

Explanation: The linking process is aborted because of previously reported severe errors.

Undefined symbol *symbolname*, first referenced in file filename

Type: Severe

Explanation: The definition of the symbol that is referenced in the input object file is not found any input object file or in the directive file.

Unknown invocation option *opt*

Type: Fatal

Explanation: An unrecognized invocation option is specified.

Write error on file *filename*

Type: System

Explanation: An error was detected when trying to write on the file.

Appendix B
GLOSSARY

.bss section A COFF file section. It normally created by the assembler and contains uninitialized data.

.bss_1 section A COFF file section. It is normally created by the compiler to hold all the uninitialized data symbols which occupy 1 byte in memory.

.bss_2 section A COFF file section. It is normally created by the compiler to hold all the uninitialized data symbols which occupy 2 bytes in memory.

.bss_4 section A COFF file section. It is normally created by the compiler to hold all the uninitialized data symbols which occupy 4 bytes in memory.

.data section A COFF file section. The .data section is normally created by the assembler and contains initialized data.

.data_1 section A COFF file section. It is normally created by the compiler to hold all the initialized data symbols which occupy 1 byte in memory.

.data_2 section A COFF file section. It is normally created by the compiler to hold all the initialized data symbols which occupy 2 bytes in memory.

.data_4 section A COFF file section. It is normally created by the compiler to hold all the initialized data symbols which occupy 4 bytes in memory.

.rdata_1 section A COFF file section. It is normally created by the compiler to hold all the ROM data symbols which occupy 1 byte in memory.

.rdata_2 section A COFF file section. It is normally created by the compiler to hold all the ROM data symbols which occupy 2 bytes in memory.

.rdata_4 section A COFF file section. It is normally created by the compiler to hold all the ROM data symbols which occupy 4 bytes in memory.

.text section A COFF file section. The .text section contains executable code.

Allocation The process by which the linker constructs output sections from input sections and allocates memory for the output sections.

COFF Acronym for Common Object File Format. This is the standard object file format for many software development tools, including the Compact-RISC software development tools.
A COFF file contains machine code and data and additional information for relocation and debugging purposes.

Common data Refers to external symbols that are not defined in any input object file, but are instead consolidated and allocated by the linker. Examples of common data include symbols that are declared with the .comm assembler directive, uninitialized variables declared in C outside any function, and Fortran COMMONs.

Cross configuration When the compilation and execution of the compiled program are done on different machines (the host and target machines are different).

Directive file The directive file controls certain actions of the linker (especially memory configuration and allocation). A directive file to be used as input for the linking process may be specified on the linker invocation line.

Entry point The starting point of program execution. The entry point address is part of the information saved in an executable object file. A symbol to mark the entry point may be specified on the linker invocation line.

Executable object file An executable object file is the final product of a linking process. In an executable object file all external symbolic references have been resolved. The executable object file is therefore in a form that can be executed on the CompactRISC-based target system.

External symbol A symbol that is recognized by all files. Such a symbol can be defined in one file but referenced from any file.

Initialization table A table created by the linker to support data initialization. This table may be used by programs requiring initialized data in an embedded environment. The initialized data is copied from ROM to RAM at run-time. The initialization table provides information about memory segments to be copied from ROM to RAM or to be filled with zeros at run-time.

Input section A COFF section of a linker input object file. The linker combines input sections to create output sections. By default input sections of the same name are combined to create one output section having this name in the output file.

Library file A collection of object files that typically contains useful routines. The linker selects from a specified library file those object files which resolve external references.

Memory map A description of the memory layout after the linking process. A memory map is an optional output of the linker.

Object file A file that is the output of either the assembler or the linker. An object file contains compiled code and data and additional information for relocation and debugging purposes.

Option The term for a parameter, specified on the command line, that is used to control the utility.

Output section A COFF section of a linker output object file. The linker combines input sections to create output sections. By default input sections of the same name are combined to create one output section having this name in the output file.

Partially linked object file An object file created by the linker, which is unexecutable since it contains unresolved external references. A partially linked object file may be used as input for a subsequent linking process.

Relocation information A part of the object file that is used by the linker in the relocation process. Relocation information contains information on symbolic references that require modification of pieces of code or data (holes) at link time. The linker uses this information to calculate the final value of the holes.

Relocation process The process by which the linker modifies pieces of code or data (holes) that cannot be calculated before link-time. These holes are typically addresses or displacements that are created as a result of symbolic references. After the allocation process is completed the linker assigns final values to these holes, using relocation information (part of the COFF file) from the input object files.

Resolution of symbolic references The process by which the linker matches external symbolic references with their definition.

Section A contiguous block of code or data having common attributes. In the COFF file code and data are separated to sections. Typically there are three types of sections: the .text section, containing machine code; the .data, .data_1, .data_2, .data_4 sections, containing initialized data; and the .bss, .bss_1, .bss_2, .bss_4 sections, containing uninitialized data.

Symbol A symbol is used either to mark a program location or to represent a data element. Each symbol is associated with a memory address after the linking process.

Symbol directory Part of a library file. The symbol directory contains information on the external symbols which are defined in library members. The linker uses the symbol directory to select the correct library member for the linking process.

Symbol table Part of the object file. The symbol table contains information about symbols defined or referenced in the source program(s), and is used for various purposes such as resolution of external references (by the linker) and symbolic debugging.

Symbolic reference The use of a symbol in a statement other than its definition. An external symbolic reference is a reference to a symbol which is defined outside the module in which it resides.

INDEX

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

X