CompactRISC™

C Compiler
Reference Manual

Part Number: 424521772-002

August 1998

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
0.6	August 1995	First release.
0.7	January 1996	Minor changes and corrections.
1.0	August 1996	CR16A Product Version. CR32A 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 C Compiler. The CompactRISC C Compiler generates high-quality code for processors using the CompactRISC architecture. It is derived from the well-known GNU C Compiler from the Free Software Foundation. It includes enhancements, such as intrinsic functions, source code register control, and full structure layout control, specifically for the development of embedded code.

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 manual describes National Semiconductor's CompactRISC C Optimizing Compiler for its family of CompactRISC processors.

The GNU C based CompactRISC compiler implements the C language as described by the ANSI C standard.

In addition, the CompactRISC C Optimizing Compiler includes important extensions for programming embedded applications like interrupt/trap handling in C, intrinsic functions and an extended **`asm`** statement. The compiler is available as a cross compiler running on Windows 95, and Windows NT operating systems.

This manual is organized as follows:

Chapter 1 *Overview* (this chapter), briefly describes the contents of each chapter, and defines the intended audience.

Chapter 2 *Invoking the Compiler*, describes the compiler structure, its command line options, and the environment variables that you can use to control its functionality.

Chapter 3 *Optimizations*, provides an overview of optimization techniques used by the optimizer. This chapter provides further guidelines to help you avoid problems that can occur when using the optimizer.

Chapter 4 *Extensions to the C Language*, describes several language features, provided by the compiler, which are not found in ANSI standard C. These features include pragma directives, using variables in specified registers, assembler instructions, and intrinsic functions.

Chapter 5 *Libraries*, describes the various libraries that provide run-time support for the CompactRISC C Compiler.

Appendix A *Standard Calling Conventions*, describes standard routine-calling conventions. These conventions enable routines in one module to communicate with routines in other modules, even if they are written in different programming languages.

Appendix B *Compatibilities with GNX C Compiler*, details the compatibilities, and incompatibilities, with GNX when the CompactRISC C compiler is run in GNX-compatible mode.

Appendix C *Guidelines for Using the Compiler*, provides some guidelines for using the compiler to port programs, using optimization options to maximum effect, and how to avoid some common programming errors.

Appendix D *Implementation-Defined Behavior*, defines the behavior of the CompactRISC C Compiler for cases which, according to the ANSI C standard, are implementation-defined.

Appendix E *Compiler Limitations*, lists some compiler limitations, of which you should be aware.

1.2 INTENDED AUDIENCE

This manual is for experienced C programmers. The information provided covers compiler options, extensions to the standard C programming language, and implementation issues. A knowledge of optimization techniques is useful, but not essential.

Less experienced programmers should use this manual in conjunction with a standard C compiler manual, such as those listed below:

- ANSI C standard (ANSI X3.159-1989).
- Harbison, Samuel and Steele, Guy. *C, A Reference Manual*, 2nd. ed., Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.
- Kernighan, Brian and Ritchie, Dennis. *The C Programming Language*, 2nd ed., Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1989.

1.3 FEATURES

The following are the main features of the C Optimizing Compiler:

- Fully compatible with ANSI C industry standard.
- Compatible with National Semiconductor's GNX C compiler for the Series 32000 architecture.
- Based on stable, proven GNU technology.
- Allows for programming of interrupt/trap handlers in C.
- Optimizations can be tuned either to improve speed or save space.
- Global variables may be placed in registers for the entire program.
- Allows the use of inline functions.
- User-controlled alignment of variables and structure members.
- Assembly output can be annotated with source lines.
- Fast compilation mode to check for errors.
- Dump error and warning messages to file.

Chapter 2
INVOCATION AND OPERATION

2.1 INVOKING THE COMPILER

This chapter describes the compiler structure, its command line options, and the environment variables that you can use to control its functionality.

2.2 COMPILER STRUCTURE

The CompactRISC Compiler consists of three programs:

- Driver
- Macro preprocessor
- C Language processor

The driver, **crcc**, is a program that parses and interprets the command line, and then calls each of the other programs in sequence, depending on its input programs and the command line options.

The macro preprocessor is the C preprocessor, **cpp**. Its input is a program file optionally containing preprocessing commands.

The C language processor program parses and optimizes its input C program, and generates an assembly program. The program's output must be assembled by the CompactRISC Assembler to produce an object code program.

The assembler is automatically called by the driver program (unless you use the **-S** option).

You produce an executable program by using the CompactRISC Linker to link one or more object files and, optionally, run-time libraries. The linker is automatically called by the driver program (unless you specify the **-c** option).

2.3 COMMAND-LINE OPTIONS

2.3.1 The Invocation Syntax

The invocation syntax of the C Compiler is:

```
crcc [ {option | filename | @argfile} ]...
```

The compiler accepts one, or more, file arguments and compilation options. Compilation options start with a dash (-). In addition, the compiler can take all, or parts, of the arguments from an argument file. An arguments file is denoted by an at-sign (@) followed by a file name.

It produces an executable file, object file(s), or assembly file(s), according to the options specified. Normally, the compiler accepts C program sources, although other types of files (e.g., assembly and object files) are also accepted. A file type is recognized by its suffix.

2.3.2 Filename Conventions

Files are identified by the compiler according to their suffix. Files with names ending with **.c** or **.i** are C source programs.

Files ending with **.c** pass through the macro preprocessor (**cpp**) before compilation. Files ending with **.i** compile directly, and assemble to produce object programs.

The compiler also accepts files with a **.s** suffix as assembly source programs. These files are assembled (to produce **.o** files) and linked.

All other files (normally **.o** or **.a** files) are assumed to be compatible object programs or archives of object programs, typically produced by previous runs of the compiler and/or archiver, and are passed directly to the linker. The object files link into one executable file with the default name **cr.x**.

Table 2-1. Filename Conventions

File Name Suffix	File Type
.c	C source file
.i	Preprocessed C source file
.s	Assembly source file
other (**.o, .a**, etc.)	Object code or library-archive file

2.3.3 Compiler Options

You can specify the following compilation options in the invocation line. Note that some options have both a short-form name and a full name. If you use the short form, `crcc` expands it into the full name.

Code Generation Options

`-O` (OPTIMIZATION)
Performs optimizations. Specify **`-O`** on the command line for the fastest possible code, without an undue increase in code size, or a significantly longer compilation time.

`-Os` (SPACE OVER SPEED)
Performs optimizations, and prefers space-over-speed in the optimization process.

or

`-O -mspace`

`-ON` (LOOP UNROLLING)
Performs loop unrolling optimization, in addition to default optimizations.

or

`-O -funroll-loops`

`-Oi` (VOLATILE)
Performs optimizations, and treats global static variables, and all pointer dereferences, as volatile.

`-g` (COMPILE FOR DEBUGGING)
Produces symbolic debugging information. This option makes it possible to debug the code at source level.

`-c` (COMPILE BUT DO NOT LINK)
Directs the compiler to perform the compilation process up to, but not including, linking. Output is one, or more, object files whose names, by default, end with **`.o`**. This option is useful when you want to invoke the linker independently at a later stage. For example:

```
crcc -c sample.c utils.c
```

creates the files **`sample.o`** and **`utils.o`**. No executable file is created.

-S (COMPILE BUT DO NOT ASSEMBLE)

Directs the compiler to terminate the compilation process before assembly. The assembly output is one, or more, files whose names are those of the source, with **.s** substituted for the original suffix. For example:

```
crcc -S sample.c utils.c
```

creates the files **sample.s** and **utils.s**. No executable or object file is created.

-n (EMBED C SOURCE LINES AS COMMENTS IN ASSEMBLY)

or

-mannotate

Puts the C source lines into the assembly output file as comments.

The **-n** option is only useful together with the **-S** option.

-o *out*(RENAME THE OUTPUT FILE)

Redirects the output file from the compilation process to a file named ***out***. For example:

```
crcc sample.c utils.c -o sample
```

generates the executable file **sample** from the two source files, and:

```
crcc -S sample.c -o new_sample.s
```

generates the assembly file **new_sample.s**.

-J*width-in-bytes*

or

-mbiggest-struct-alignment-*width-in-bits*

(ALIGNMENT WITHIN STRUCTURES)

Sets the structure-member alignment to bytes (***width*** = 1), words (***width*** = 2) or double-words (***width*** = 4).

A structure member is aligned to a boundary, whose value is the smallest of ***width*** and the member alignment requirement.

In addition, the **whole** structure is padded to make its size a multiple of the maximum alignment of the structure members (calculated as above).

The default value for ***width*** is the width of the internal data bus of the architecture e.g., 2 for **CR16** and 4 for **CR32A**.

Example:

```
crcc -J1 -c prog.c
```

instructs the compiler to use one-byte alignment or, in other words, not to generate gaps between structure members. This command line is equivalent to:

```
crcc -mbiggest-struct-alignment-8 -c prog.c
```

Notes

1. If you use **-mbiggest-struct-alignment**, you must specify ***width-in-bits*** in bits (***width*** = 8, 16 or 32).
2. For correct execution of your program, all its compilation units (i.e., all the **C** files) must be compiled with the same ***width***.

-KB*width*

or

-mbiggest-alignment-*width-in-bits*

(SET TARGET BUSWIDTH)
Allows you to tune the compiler to the bus width of the target system. The bus width specification guides the compiler to allocate memory for local variables, which reside on the program stack, in the most efficient manner. A local variable is aligned to a boundary whose value is the smallest of ***width*** and the variable alignment requirement. The ***width*** parameter is specified in bytes (***width*** = 1, 2 or 4).
The default value for ***width*** is the width of the internal data bus of the architecture e.g., 2 for **CR16** and 4 for **CR32A**.
Note, if you use **-mbiggest-alignment**, you must specify ***width-in-bits*** in bits (***width*** = 8, 16 or 32).

-fshort-enums(OPTIMIZE SIZE OF ENUMERATION TYPES)
Selects the shortest possible size for each enumeration type. By default the size of an enumeration type is equal to the size of an integer.

-finline-functions (INLINE FUNCTIONS)
Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared **static**, then the function is not normally output as assembler code in its own right. This option works only if the **-O** option is also used.

-fkeep-inline-functions
(KEEP BODY OF INLINE FUNCTIONS)
Outputs a separate, run-time callable, version of the function, even if all calls to a given function are integrated, and the function is declared **static**. This option works only if the **-O** option is also used.

-ffixed-*REG* (DO NOT USE A CERTAIN REGISTER)
Treats the register, ***REG***, as a fixed register; generated code should never refer to it. ***REG*** must not be used for passing parameters, or returning a value from a routine (e.g., **-ffixed-r8**).

-sbrel
(USE STATIC-BASE RELATIVE ACCESS MODE FOR GLOBAL AND STATIC VARIABLES)
Instructs the compiler to generate code that accesses all the global and static variables (CR32A), or part of them (CR16), using a static-base (SB) relative mode to save code. The static base is a

contiguous block of memory that contains all (CR32A), or part (CR16), of the global and static variables, and is pointed to by register **R13** (**R12** for CR16B large programming model). The SB relative mode is a register-relative addressing mode, where the base register is **R13** (**R12** for CR16B large programming model). In SB relative addressing mode, the compiler always uses medium size displacement (CR32A), or small size displacement (CR16), thus reducing the code size.
For CR16 the **-sbrel** option is used in conjunction with the **#pragma sbrel** directive.
For detailed information see Section 3.2.

-msmall (CR16B only)
(GENERATE SMALL MODEL CODE)
Instructs the compiler to generate code for the CR16B small programming model. This is the default.

In this model, code pointers are 16 bits wide.

-mlarge (CR16B only)
(GENERATE LARGE MODEL CODE)
Instructs the compiler to generate code for the CR16B large programming model.

In this model, code pointers are 20 bits wide.

-mcr16a (CR16 only)
(GENERATE CR16A COMPATIBLE CODE)
Instructs the compiler to generate code for the CR16A core. In this model the code pointers are 16 bits wide.

-mbank (CR16A-BASED CHIPS WITH BANK SWITCHING HARDWARE SUPPORT)
This option enables the Bank Switching mechanism. This mechanism is needed for CR16A applications with code size larger than 128 Kbytes.

Warning and Error Options

-ansi (STRICT ANSI)
Accepts only strict ANSI standard C programs. This option rejects non-ANSI programs.

-Wimplicit(WARN ABOUT IMPLICIT FUNCTION DECLARATIONS)
If this option is used, the compiler issues a warning message for each function which is called and its prototype declaration is missing. In this case, the function prototype is implicitly set by the compiler according to the function call. Note: the standard I/O functions (e.g., **printf**) are an exception. If you fail to declare a standard I/O function prototype (e.g., do not include **stdio.h**) the compiler implicitly sets its prototype to that of the ANSI-C standard i.e., the same prototype that appears in **stdio.h**.

-w (NO WARNING DIAGNOSTICS)
Suppresses the warning diagnostics, which the compiler normally prints if there are inconsistencies in the input program.

-Q (ERROR CHECKING ONLY)
Allows quick error-checking compilation. No code is generated.

or

-fsyntax-only

-v (VERBOSE MODE)
Lists the subprograms of the compiler as they are executed by the driver program.

-vn (SHOW BUT DO NOT ACTUALLY EXECUTE)
Lists the compiler subprograms that are called by the compiler's driver program, while compiling an existing file, without actually executing these subprograms. Can be used to verify operation of other compiler options.

-z (DUMP ERRORS AND WARNINGS INTO ERROR FILE)
Dumps errors and warnings into **filename.err** file, where **filename** is the base name of the input file. For example:

```
crcc -z test.c
```

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

Note, using this flag causes the linker to dump its errors and warnings into **filename.err**, where **filename** is the output filename. For example:

```
crcc -z test1.c test2.c test3.c -o test.x
```

generates the following error files:
- **test.err** for the linker errors
- **test1.err test2.err test3.err** for compiler and assembler errors

-zn*filename* (DUMP ERRORS AND WARNINGS INTO ERROR FILE)
Dumps errors and warnings into the specified *filename*. For example:

```
crcc -zntests.log test1.c
```

generates the error **file tests.log**. Note, there must be no space between **zn** and **filename**.

Preprocessor Options

-E (RUN **cpp** ONLY)
Terminates the compilation after preprocessing; only the **cpp** preprocessor is invoked, and its output is sent to the standard output, **stdout**.

-P (RUN **cpp** ONLY, REDIRECT OUTPUT TO .i FILE)

This option is similar to **-E**, except that the output of **cpp** is sent to a file with a **.i** extension. For example:

```
crcc -P sample.c utils.c
```

creates the files **sample.i** and **utils.i**.

-C (LEAVE COMMENTS IN)

Prevents the preprocessor from removing the comments from its output. This option can be useful when **cpp**'s output must be examined. It can only be used in conjunction with the **-E** option.

-D*symbol[=def]*(DEFINE **cpp** SYMBOL)

Defines ***symbol*** equal to ***def*** to the preprocessor. If no explicit value is given, ***symbol*** is defined as having the value 1. This option is equivalent to putting:

```
#define symbol def
```

at the beginning of each C source file. For example:

```
crcc -DMODE=DEVELOPMENT sample.c
```

acts as if the following define was at the head of **sample.c**:

```
#define MODE DEVELOPMENT
```

-U*symbol*(UNDEFINE **cpp** SYMBOL)

Using this option is equivalent to putting:

```
#undef symbol
```

at the beginning of each C source file.

-I*dir*(SPECIFY DIRECTORY FOR INCLUDED FILES)

Instructs **cpp** to use the specified directory as the default directory for included files. Include files that are called using double quotes, for example:

```
#include "filename"
```

are sought first in the directory of the compiled file, then in the directories specified by **-I**, and finally in directories on a standard list (***CRDIR*/include/**), where ***CRDIR*** is the root directory of the CompactRISC Development Toolset.

If you explicitly name the file to be included using the complete path, for example:

```
#include "/a/mydir/filename"
```

the named file is sought directly.
If angle brackets are used instead of double quotes, for example:

```
#include <filename>
```

the file is sought first in the directories specified by **-I**, and then in the directories on a standard list (***CRDIR*/include/**).

-M (RUN **cpp** ONLY, GENERATE MAKEFILE DEPENDENCIES)

Runs only the **cpp** macro preprocessor on the named C programs, requests it to generate makefile dependencies and then sends the result to the standard output, **stdout**.
For example:

```
crcc -M test1.c  test2.c > new.mak
```

runs **cpp** on two C programs in the current directory and generates all makefile dependencies for them. These dependencies are then sent to the file **new.mak**.

Linking Options

-l*name* specifies a program library. By default, the linker looks for the library **lib*name*.a** in ***CRDIR*/lib** (where ***CRDIR*** is the root directory of the CompactRISC development toolset), and links it with the program.

-KFemulation

(INCLUDE THE FLOATING-POINT EMULATION LIBRARY)
If you use the compiler driver to invoke the linker (as it is by default), this option instructs the compiler to include the floating point emulation library (**libhfp**) as one of the linker inputs. By default, the compiler does not include this library. This option is necessary if the program includes floating-point operations.

-L*dir* (SPECIFY THE LIBRARY DIRECTYORY)

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

The **-L** option must precede any **-l** option. The linker searches for libraries specified through the **-l** invocation option first in **dir**, and then in the default library locations.
(See the *CompactRISC Toolset - Object Tools Reference Manual*).

General Options

It is possible to pass any option to the various phases of the compiler.

-W*phase*,*option*

(PASS OPTIONS TO COMPILATION PHASE ***phase***)
Passes options to the C preprocessor (***phase*** = p), the compiler (***phase*** = c), the assembler (***phase*** = a), or the linker (***phase*** = l). The *options* must not contain embedded spaces, unless quoted. An option is passed as one argument whether it contains spaces or not. To pass multiple arguments, use commas.

For example, the command:

```
crcc -Wl,-d,linker.def
```

instructs the linker to use the file **linker.def** as the linker definition file.

2.4 ENVIRONMENT VARIABLES

The following environment variables are used by **crcc**:

TMPDIR This environment variable defines the location at which temporary files are created in the compilation process. If **TMPDIR** is not defined, the compiler looks for the **TMP** variable, and then for **TEMP**. If none of the above exist, the default location for temporary files is the current directory. For example, to direct temporary files to the directory **c:\temp** type the following:

```
set TMPDIR=c:\temp
```

CRDIR This variable must be set to the directory where the CompactRISC Toolset is installed. The installation procedure of the CompactRISC Toolset takes care of this.

CRINC If this variable is set, its value is used as an additional search path for header files. The value is a string which represents a list of directories, in the same format as the **PATH** environment variable. The compiler appends this directory list to the list of directories which are specified using the **-I** compilation option. In other words, for each directory in **CRINC** the compiler adds an implicit **-I** option, with this directory as parameter.

2.5 PREDEFINED CPP SYMBOLS

The following **cpp** symbols are predefined by the CompactRISC compiler as if they were specified by the **-D** option, or with the **cpp** *#define* directive:

Symbol	Comments
__CR__	Always defined.
__GNU__	Always defined.
__CR16A__	Always defined by the CR16A compiler (ver1.x). Defined by the CR16B compiler when in cr16a model.
__CR16B__	Always defined by the CR16B compiler.
__CR16BS__	Defined by the CR16B compiler in small model only.

Symbol	Comments
__CR16BL__	Defined by the CR16B compiler in large model only.
__CR32A__	Always defined by the CR32A compiler.
__OPTIMIZE__	Defined only when **-O** is specified
__STDC__	Defined only when **-ansi** is specified
__STRICT_ANSI__	Defined only when **-ansi** is specified

Chapter 3
OPTIMIZATIONS

3.1 INTRODUCTION

This chapter describes some of the advanced optimization techniques used by the CompactRISC compiler.

The most important optimization techniques are:

- Constant folding
- Arithmetic simplifications
- Function inlining
- Jump optimization
- Common sub-expression elimination
- Loop optimization
- Flow optimization
- Register allocation

3.2 OPTIMIZATION TECHNIQUES

Constant folding

If an expression or condition consists of constants only, it is evaluated by the compiler into one constant, saving computation at run-time.

Arithmetic simplifications

Arithmetic simplifications, such as the Distributive Law, are carried out on expressions.

Function inlining

Function inlining integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared static, the function is not normally output as assembler code in its own right. This optimization is enabled by a special option (**`-finline-functions`**) or by marking a function as inlined, see Section 4.5.

Jump optimization

Jump optimization simplifies:

- jumps to following instructions
- jumps across jumps
- jumps to jumps

In addition, jump optimization performs jump threading. It converts code, originally written with jumps, into sequences of instructions that directly set values as a result of comparisons.

Example

```
if (j>k) b=0 else b=1;
```

becomes:

```
b=j<=k
```

The jump threading optimization deduces information from the conditional jump test, and uses this information to optimize the taken flow path.

Example The code:

```
if (i>5) {
      if (i<=5) i = 0;
      i++;
}
```

is transformed into the equivalent code:

```
if (i>5) {
      i++;
}
```

Common subexpression elimination Searches for a common subexpression, and saves the result of the first computation in a temporary location (usually a register). This saved result is employed in any further use of the expression.

Example

```
a = b + c;
d = b + c;
```

b + c is a common subexpression. If the values of **b** and **c** do not change between the two expressions, the compiler calculates the values of **b + c** once, assigns the result of the calculation to a temporary location, and uses that location instead of recalculating the expression.

Loop optimization Loop optimizations consist of moving constant expressions out of loops, strength-reduction, and optionally, loop unrolling.

Strength reduction Strength reduction optimization replaces complex operations with simpler ones. This is primarily useful for reducing complex array-subscript computations, typically by turning multiplications into additions inside a loop.

Example The loop:

```
int a[15];
for (i=0; i<15; i+=1) {
      a[i] = 1;
}
```

is transformed into the equivalent of:

```
int a[15],*ptr;
for (ptr=a; ptr < &a[15]; ) {
     *ptr++ = 1;
}
```

which is smaller and faster.

Loop unrolling

Loop unrolling duplicates the body of a loop. This reduces the number of times that the loop control code is executed. Loop unrolling improves performance, but it increases code size.

Loop unrolling is performed only for loops whose number of iterations can be determined at compile time, or before execution of the loop.

Loop unrolling is not performed by default. (It is enabled by the option **`-O`***n* or **`-funroll-loops`**.)

Flow optimization

Flow optimization deletes unreachable code, and computations whose results are never used.

Register allocation

Allocates registers to frequently used variables and subexpressions. Only non-address-taken automatic variables (automatic variables whose addresses were taken using the **&** operator), or constants, or a subexpression which is a combination of both, are allocated to a register.

Space optimization

These optimizations are intended to reduce code size and memory usage.

1. Prefer space over speed
   Usually there is no contradiction between improving program speed and reducing its code size. If there is such a contradiction, by default, the CompactRISC C Compiler optimizes for speed. However, when you use the **`-Os`** option, the compiler prefers space over speed and changes its optimizations accordingly. The following optimization is performed when you use **`-Os`**:

   a. Save-emul optimization (CR16A/CR32A only)
      The compiler replaces a routine entry/exit sequence by a call to a special routine which performs a save/restore of the registers. The save/store emulation routines are located in the **`libc`** library.
      Note: the save-emul optimization is not necessary for the CR16B because **`PUSH`** and **`POP`** instructions, which can save/restore several registers within a single instruction are available on this CPU core.

Example

A save-emul optimization replaces an entry sequence:

```
addw      $-8,sp
storw     r10,6(sp)
storw     r9,4(sp)
storw     r8,2(sp)
storw     r7,0(sp)
```

By:

```
addw      $-8,sp
bal       r1,___save_r7_to_r10
```

b. Register allocation
When the **-Os** compiler option is used, the compiler changes its criteria for allocating registers to local variables. It gives relatively less weight to loop-nest considerations (e.g., variables which are used inside loops) and more weight to code-size related considerations (e.g., variables which are used several times along the function regardless of loops).

c. Prefer multiply instruction
When a variable is multiplied by a constant, the compiler can generate a combination of shifts and adds which is faster than the multiply instruction. However, if you specify the **-Os** option the compiler uses the multiply instruction to save code.

d. Loop code size optimization
By default, the compiler generates code which tests a for loop condition, once before the loop, and then at the end of each loop. This saves one branch. With the **-Os** option, the compiler checks for the loop condition only at the end of the loop, in order to save code.

e. Switch optimization
By default, code the compiler generates for a switch statement may implement a binary search. This improves the speed when searching for the case in the statement to which control should be transferred. With **-Os**, the compiler prefers a sequantial check of the different case values. This generates slower, but more compact, code.

2. Static-Base (SB) relative access mode optimization
This optimization can significantly reduce the code size, and in most cases does not adversly affect the speed.

The **-sbrel** option instructs the compiler to generate code which accesses global and static variables, using a static-base (SB) relative mode, to save code.

The static base register is defined as follows:
- **R13** for the CR16A, CR32A and the CR16B, small model
- **R12** in the CR16B large model.

The static base memory is a contiguous block, and is limited to 32 bytes for the CR16, due to the limitation of small size displacements, or ±32 kbytes for the CR32A due to the limitation of medium size displacements. The compiler always uses medium size displacement (CR32A), or small size displacement (CR16), thus reducing the code size significantly: only 4 bytes per variables access (CR32A), and only 2 bytes per variables access (CR16). The compiler directs static base variables to a special section, **.sb**.

The linker must be given the address of the static base by defining the symbol **__STATIC_BASE_START** in the linker directives file. In addition, the SB register must be initialized to the address of the static base as part of the start-up routine as in the following example:

```
movw    $__STATIC_BASE_START,r13
```

Warning

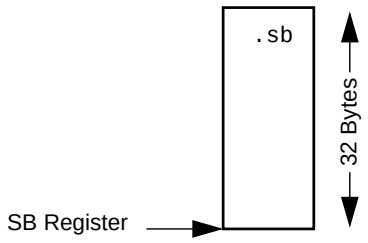
To avoid using the static base register for other purposes when compiling with **-sbrel**, we advise you to compile all the program C source files with this option.
If some of the source files are intentionally compiled without this option, we recommend using the **-ffixed-r13** (**-ffixed-r12** in CR16B large programming model) compiler option for these files, to prevent any usage of the SB register which changes its contents.

CR16

The static base can contain up to 17 1-byte variables (e.g., **char**) and any number of 2-byte variables (e.g., **int short**) as long as the total size of the static base is not more than 32 bytes. All these variables must be global/static uninitialized variables.
You can specify the variables to be part of the static base using the **#pragma sbrel** directive (see Section 4.3.4). You should prefer the most frequently accessed global or static variables of your application to achieve maximum code saving. For the CR16 the symbol **__STATIC_BASE_START** points to the start address of the static base.

The following line (from the CR16A default linker directives file) shows how to allocate the static base:

```
__STATIC_BASE_START = ADDR(.sb);
```

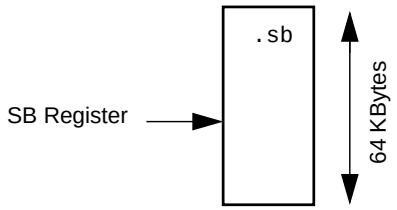


CR32A

The symbol **__STATIC_BASE_START** should point to the middle point of the static base. The following line (from the CR32A default linker directives file) shows how to allocate the static base:

```
__STATIC_BASE_START =
ADDR(.data) + (SIZEOF(.data)/2)+ (SIZEOF(.bss)/2);
```

This example assumes that the program data resides in two sections which are consecutive in memory: the **.data** section (initialized data) and the **.bss** section (uninitialized data). It locates the Static Base in the middle of the memory region that is formed by these two sections.



Note: all non-constant global and static variables are included in the static base. It can be hazardous to use variables in your program that are defined in the linker directives file, and are bound to specific addresses (e.g., to memory-mapped I/O registers). Consider the following variable:

```
extern volatile unsigned char io_reg;
```

which is defined in the linker directives file, and bound to a specific address as follows:

```
_io_reg = 0xfc00;
```

The compiler attempts to use the SB relative mode when accessing this variable. However, this variable is most probably located outside the contiguous memory block of the **.data** and **.bss** sections, which form the static base, and hence cannot be accessed using the SB relative mode.

To avoid this probelm, you can use the following method to access memory-mapped I/O registers in your C code:

```
#define IO_REG *((volatile int *)0xfc00)
```

Now, instead of a variable we have a constant address, IO_REG. The compiler does not attempt to use the SB relative mode to access this address. See also Section 7.10.4 .

Another problem which may occur is that the total size of the **.data** and **.bss** sections in your aplication is greater than 64K. In this case, you must decide which variables *not* to put in the sb area. You should define the less frequently accessed variables as part of a user section (see Section 4.3.3). Since only the variables that are in the default sections are referenced through **sbrel**, variables in the user section are referenced normally.

The following example shows how to define a variable in this manner:

```
#pragma section (".ex_zone", var1,var2)

int var1[100];

char var2;
```

The variables var1 and var2 are put in the **.ex_zone** section, and are not referenced through sbrel. The **.ex_zone** section must be defined in your linker definiton file.

Chapter 4
EXTENSIONS TO THE C LANGUAGE

4.1 INTRODUCTION

The CompactRISC C compiler provides several language features not found in ANSI standard C. Some of these extensions originate from the GNU C compiler on which the CompactRISC compiler is based. The goals of these extensions are:

- To enable better programming in an embedded environment.
- To enable programmers to take advantage of specific features of the CompactRISC architecture.
- To allow more control over the compilation process.

4.2 FAR VARIABLES (CR16 ONLY)

The **__far** qualifier, recognized by the CR16 compiler, is used to denote variables whose memory address can be 65536 (64K), or above.

The CR16 architecture supports a data address-space in the range 0-256K (18-bit addressing). However, data in the range 64K - 256K (which we call "far data") has a unique aspect. If a pointer to data is below 64K, the compiler generates code which accesses this data using a register-relative addressing mode. The pointer value is copied to the register prior to accessing the data. Since a CR16 register has 16 bits, it can point to any address in the range 0-64K. For a pointer to far data, the compiler must use the far-register relative addressing mode. Two consecutive registers are required to hold the value of the pointer, since it contains more than 16 bits. Therefore, if you have a pointer that may point to far data you must use the **__far** qualifier to inform the compiler. For example:

```
__far int *p;
```

Clearly, using a far pointer is slower than using a normal pointer, since a far pointer must be loaded into two registers rather than one. You should, therefore, declare a pointer as a far pointer only if you are sure that it is going to point to far data at some stage.

Arrays in the far-data address range should also be declared as far. For example:

```
__far char a[1000];
```

In addition to far pointers, you should also declare any variable in the far-data address range, whose address may be taken, as far. For example:

```
__far int i;
void foo(__far int *);
...
foo(&i);
```

In general, it is a good practice to define any variable that may reside in the far data address space as far.

Note that declaring a variable as far does not automatically place it in a memory address above 64K. To place a far variable above 64K, use the section pragma (Section 4.3.3).

4.3 PRAGMAS

ANSI C defines a **#pragma** directive which is the universal method for extending the space of directives.

The CompactRISC C Compiler uses the **#pragma** directive in the following cases:

- **#pragma interrupt**
- **#pragma trap**
- **#pragma set_options**
- **#pragma reset_options**
- **#pragma section**
- **#pragma sbre**l

4.3.1 Interrupt/Trap Pragma

The interrupt/trap handler is written as a regular C routine, in the usual C function definition syntax.

Example

```
void hndlr_foo(void)
     {
          printf("division by zero");
          exit (1);
     }
```

The function is designated as an interrupt/trap handler by using a special **#pragma** to mark it as such.

Syntax for interrupts:

#pragma interrupt(***function_name***)

Syntax for traps:

#pragma trap (***function_name***)

function_name is the name of the function to be marked as an interrupt/trap handler.

Example

```
#pragma trap(hndlr_foo)
```

Only the registers used in the interrupt/trap routine (and the scratch registers if the interrupt/trap calls another function) are saved. For further details about the standard calling convention, see Chapter 6.

The compiler issues a warning if a function is marked as an interrupt/trap handler using the **#pragma** directive, but no definition of the function was found in the compiled module.

Multiple **#pragma** directives with the same function name are considered errors, unless they are identical.

Restriction

The **#pragma** directive must appear before any declaration or definition of the function. The placement of the **#pragma** interrupt/trap in any other location results in an error message.

Using interrupt/trap handlers

It is your responsibility to install the address of the interrupt/trap handler in the proper entry of the interrupt dispatch table (see the example presented in Appendix C for further information).

You may not call an interrupt/trap handler directly from the C code. This is because the instructions for returning from the interrupt/trap routine, and those for returning from a regular routine, are different.

Note, although trap and interupt pragma have the same effect, for future compatability we recommend that you use different pragma for trap and interrupt routines.

4.3.2 Set/reset Options Pragma

This pragma is used for setting/resetting compilation options from within the source file. The **set_options** pragma receives as a parameter a constant string which consists of a list of compiler options separated by spaces. The **set_options** pragma adds the options received as a parameter to those that are currently set.

The **reset_options** pragma has no parameters and resets the options to those set at the start of the compilation.

Notes

The pragma can appear only between function definitions.

If an option has two notations, you must use the second one (full name), and not the short form, which the **set_options** pragma does not recognize. See Section 2.3.3 for more details.

Example

Consider two functions, **f1** and **f2**, both containing loops that we want to optimize using loop unrolling. In addition, we do not want **f2** to use **r10** as it calls a function which destroys **r10**.

```
#pragma set_options("-funroll-loops")
f1(){
      <loop>
      ...
}

#pragma set_options("-ffixed-r10")
f2(){
      <loop>
      destroy-r10();
      ...
}

#pragma reset_options()
```

4.3.3 Section pragma

The section pragma allows you to place variables in specific user-defined sections. This is especially useful when there are special memory areas in which you would like to place specific variables.

```
#pragma section(section_name, var1, ..., varn)
```

The first parameter **section_name** is a constant string. It should be limited to six characters. The following parameters, *varn*, are variable names.

The variables received as parameters are placed in a section whose name is **section_name** followed by an underscore (_) and the alignment of the variable (i.e., 1, 2 or 4). The separation into different sections, based on the alignment, minimizes the amount of wasted space in the allocation of the variables.

The type of the section (bss, const, initialized data) is determined according to the first variable in its list. You can not mix different types of variables in one user-defined section. An attempt to do so, results in a warning message.

The section pragma must precede the declaration of the variables to which it refers.

Example You have an area of fast memory between addresses 0xff00 and 0xffff (inclusive). To place frequently-used global variables in that memory, you declare the variables as follows:

```
#pragma section (".fast", glob_flags, glob_counter)

unsigned char glob_flags;
short glob_counter;
```

The generated assembly file appears as follows:

```
      .globl _glob_flags
      .section     .fast_1,"b"
      .align 1
_glob_flags:
      .space 1
      .globl _glob_counter
      .section     .fast_2,"b"
      .align 2
_glob_counter:
      .space 2
```

To link this file (and any other files that were similarly compiled) so that these variables reside in the fast memory, the linker definition file should take the following form:

```
MEMORY {
      ...
      /* Define fast memory area */
      fast_mem:     origin=0xff00, length=0x100
      ...
}

SECTIONS {
      ...
      /* Direct the .fast section (a unification of all
         .fast_* input sections) to fast memory.
      */
      .fast ALIGN(4) INTO(fast_mem):
           {*(.fast_4) *(.fast_2) *(.fast_1)}
      ...
}
```

4.3.4 SB Relative Pragma (CR16 only)

With the SB relative pragma you select the variables that are directed to the static base when you use the **`-sbrel`** option (see Pages 2-10 and 3-4).

```
#pragma sbrel(var1, var2, ...)
```

If you select a variable for the static base with **#prgama sbrel**, it is accessed in SB relative mode, which saves code (only two bytes per access). Since the static base is limited in space to 32 bytes (see Page 3-4) it makes sense to select the most frequently accessed variables in the program for the static base.

The SB relative pragma must precede the declaration of the variables to which it refers.

Either uninitialized variables or constant variables may be selected for the static base. However, you can not select variables of both types, i.e., all the selected variables must be either uninitialized variables, or constant variables.

4.4 VARIABLES IN SPECIFIED REGISTERS

The CompactRISC C compiler allows you to force global and local variables into specific hardware registers.

Registers that are assigned to global variables are reserved throughout the program. This is useful in programs which have a few global variables that are frequently accessed.

When a local variable is specified to be in a specific register, the compiler does not necessarily reserve that register for the variable throughout the function. The compiler analyses the function, and if there are code segments in which the value in the variable is not used until the next write to the variable, or until the end of the function, the compiler might use the register for other purposes.

Global register variables

In the CompactRISC C Compiler, you can define a global register variable by adding an **asm** statement, whose parameter is the name of the register, to a register variable declaration. For example:

```
register int *foo __asm__("r9");
```

where **r9** is the name of the register which should be used.

Assigning a certain register to a global variable reserves that register entirely for this use, at least within the current compilation unit. The register is not allocated for any other purpose in the functions in the current compilation unit.

Always choose a non-scratch register (i.e., **r7** to **r13** and **r7** to **r12** for CR16B large programming model) for a global variable. Non-scratch registers are saved and restored by function calls, and thus library routines do not clobber them. For more details about scratch and non-scratch registers refer to Chapter 6.

Furthermore, if you intend to access a global register variable from interrupt or trap handlers, or from more than one thread of control, and you are using the CompactRISC run-time libraries, we strongly recommend that you use only registers **r11**, **r12**, **r13** for global variables (**r10**, **r11**, **r12** for CR16B large programming model). Since the CompactRISC run-time libraries do not use these registers, you can safely access a global register variable (from interrupt or trap handlers, or from more than one thread of control), only if it is located in one of those registers (assuming the program uses the run-time libraries). Remember, if you use the **-sbrel** option, **r13** (or **r12** in case of CR16B large programming model) is used as a pointer to the static-base, and therefore should not be allocated for a global variable in this case.

It is not safe for a function, **foo**, that uses a global register variable to call another such function, **bar**, by way of a third function, **lose**, that was compiled without knowledge of this variable (i.e., in a different source file in which the variable was not declared). This is because **lose** might save the register, and put some other value there.

Example

```
x.c :
     register int globreg __asm__("r11");
     foo()
     {
          globreg = something;
          lose();
     }

     bar()
     {
          use(globreg);
     }

y.c :
     lose()
     {
          ...
          bar();
          ...
     }
```

As you can see, if **y.c** is compiled without the knowledge that **r11** is used as a global register variable, **lose** might use it for other purposes and then **bar** will not see its original value.

In conclusion, we recommend that you distribute the information about the usage of any register as a global variable to all the program's source files. There are two possible ways of doing this:

- Define the global register variable in a header file, and include this header file in all the source files.
- Compile all the source files which do not use the global register variable using the **-ffixed-*REG*** option (where ***REG*** is the register allocated for the global variable) to prevent the functions in these files from using ***REG*** for any purpose.

Global register variables may not have initial values. An executable file has no means of supplying initial contents for a register. In addition, the **extern** storage class specifier may not be used for global register variables, since memory allocation is, in any case, not performed.

Local register variables

You can define a local register variable, with a specified register, in a similar manner to a global variable, except that a local variable appears within a function.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines that the variable's value is not needed. Excessive use of this feature may leave too few available registers to compile certain functions.

Note

If you use the **-Os** option, and you also allocate registers for variables, the way you select those registers may affect the compiler-generated code.
When you use **-Os**, the compiler selects non-scratch registers as follows:

> If one non-scratch register is required then **r7** must be selected, if two non-scratch registers are required then **r7** and **r8** must be selected, if three then **r7**, **r8** and **r9** must be selected and so on.

Thus a contiguous range of non-scratch registers, starting with **r7**, must be selected. If, for a variable, you allocate a non-scratch register that the compiler would otherwise use, you disturb the compiler and the resulting code may not be optimal. Therefore, when selecting registers for variables, start first with **r13** (**r12** for CR16B large programming model), then if you need another register select **r12** and so on. This disturbs the compiler as little as possible.

4.5 INLINE FUNCTIONS

By declaring a function **inline**, you can direct the compiler to integrate its code into the code for callers. This makes execution faster by eliminating the function-call overhead. In addition, if any of the actual argument values are constants, you do not need to include all of the inline function's code, which allows simplifications at compile time.

The compiler respects a declaration of function as an **inline** function only when optimizations are in effect (**-O** option is used).

To declare a function inline, use the **__inline__** keyword as a qualifier in its declaration.

Example

```
__inline_
int
inc (int *a)
{
     (*a)++;
}
```

The option **-finline-functions** makes the compiler try to inline simple functions. Note that certain usages in a function definition can make it unsuitable for inline substitution.

If all calls to a function, that is both inline and **static**, are integrated into the caller, and the function's address is never used, the function's own assembler code is never referenced. The compiler does not actually output assembler code for the function.

For various reasons, some calls can not be integrated (in particular, calls that precede the function's definition, and recursive calls within the definition). For non-integrated calls, the function is compiled to assembler code as usual. A function must also be compiled as usual if the program refers to its address, because that can not be inlined.

When an inline function is not **static**, the compiler assumes that there are calls from other source files; since a global symbol can be defined only once in any program, and calls therein can not be integrated. A non-**static** inline function is, therefore, always compiled on its own in the usual fashion, although it may be inlined in functions that are in the file in which it is defined.

If you specify both inline and **extern** in a function definition, the definition is used only for inlining. A function is never compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

The combination of **__inline__** and **extern** can be used like a macro. Put a function definition in a header file, with these keywords, and copy the definition (without **__inline__** and **extern**) to a library file. The definition in the header file causes most calls to the function to be inlined. Any remaining uses of the function refer to the copy in the library.

4.6 DOUBLE SLASH ('//') COMMENTS

Similar to C++, a double slash (**'//'**) can be used to denote a comment. Anything following the double slash until the end of the current line, is treated as comment.

Chapter 5
RUN-TIME LIBRARIES

5.1 INTRODUCTION

The CompactRISC run-time libraries provide run-time support for the CompactRISC C Compiler. The following list summarizes the contents of these libraries according to functional groups:

5.1.1 The `libc` Library

Standard ANSI-C Library Functions The CompactRISC standard ANSI-C library is based on the Dinkum C library by Dinkumware Ltd. This library includes functions such as standard I/O functions (e.g., **`printf()`**), string processing functions (e.g., **`strcpy()`**), dynamic memory allocation functions (e.g., **`malloc()`**), and mathematical functions (e.g. **`sin()`**).

Low-level I/O functions This group includes low-level I/O functions, which implement the CompactRISC virtual I/O mechanism. This mechanism enables a program, which is executed on a development board, to read data from, or write data to, a host computer (including file access), using a communication protocol with the CompactRISC debugger which runs on the host. This group includes functions like **`read()`**, **`write()`**, or **`open()`**. These low-level functions are used by some of the standard ANSI-C functions which deal with standard I/O. You may also call them directly; see Section 5.3 for details.

32-bit emulation Applicable for the CR16 development toolset only. This group includes functions which provide long int (32-bit) emulation for CR16. The CR16 C compilers automatically calls these routines to implement certain 32-bit operations, such as multiplication and shift. See Section 5.4 for more details.

Division emulation This group includes division and modulu emulation functions. Currently, the CompactRISC architectures do not provide a hardware division-instruction, and therefore division must be emulated in software. The CompactRISC C compiler automatically calls these functions to implement division and modulu operations. See Section 5.5 for more details.

Series 32000 instruction emulation This group includes emulation functions for National Semiconductor Series 32000 microprocessor instructions. Thus they also provide backward compatibility with the GNX compiler. See Section B.4 for more details.

5.1.2 The `libhfp` and `libd` Libraries

Floating-point emulation

This group includes floating-point emulation functions. The CompactRISC C Compiler automatically calls these functions to implement floating-point operations when a floating-point unit is not available in the target hardware.

If your program does not need floating-point operations, you can use a floating-point dummy library, **`libd`**, instead of **`libhfp`**. This is necessary if parts of the program can perform floating-point operations, but are not executed (for example, although the **`printf`** function can print variables of type **`float`** and **`double`**, many programs do not require this facility).

5.1.3 The `libstart` Library

ADB support functions

This group includes functions which support the development of software for an Application Development Board (ADB). These functions include the default start-up routine, default interrupt dispatch table and default trap handlers, which are essential for debugging the application program.

5.1.4 Using the Libraries

The CompactRISC libraries are used as inputs to the CompactRISC linker. There are two ways of including the CompactRISC libraries in the link process, automatic invocation and direct invocation.

- When the linker is automatically invoked by the compiler driver (**`crcc`**), some CompactRISC libraries are specified to the linker by default, and you can add others.
- When the linker is invoked directly, you must specify each library required by your program to complete the link process.

Automatic linker invocation

The default invocation of the **`crcc`** driver tells the linker to include the following libraries in the linking process: **`libstart`**, **`libc`**, and **`libd`**.

If your program needs to perform any floating-point operations, or one of the standard mathematical functions (which themselves perform floating-point operations), invoke **`crcc`** with the **`-KFemulation`** option. In this case, the compiler tells the linker to use the real floating-point emulation library (**`libhfp`**) rather than the floating-point dummy library (**`libd`**). Alternatively, If you invoke the linker directly (i.e., using the **`cr-link`** command and not via **`crcc`**) add the **`-lhfp`** option to the invocation line. The linker then uses **`libhfp`** in the link process.

Direct linker invocation

If you invoke the linker directly, use **-l***library-name-extension* to specify each library you want to use, where *library-name-extension* is the name of the library, excluding the **lib** prefix. For example, to include the same libraries as used by the compiler by default in the linking process,

invoke the linker as follows:

```
crlink ... -lstart -lc -ld
```

To include floating-point emulation, replace **-ld** by **-lhfp**, or use the **-KF***emulation* option.

To use libraries that are compatible with TMON versions lower than 2.1, replace **-lstart** with **-ladb**.

Refer to the *CompactRISC Toolset - Object Tools Reference Manual.* for detailed explanations of the CompactRISC linker invocation.

CR16A, CR16B large and small model libraries

The CR16B toolset includes three sets of run-time libraries. Two sets are used for CR16B: one for programs that use the small programming model, and one for programs that use the large programming model. The third set is CR16A compatible. The CompactRISC linker automatically selects the appropriate set of run-time libraries according to the type of the objects being linked.

5.1.5 Re-entrant Aspects of Libraries

What is a re-entrant function?

A function is re-entrant if it can be interrupted during execution, and invoked by a call from within an interrupt or a parallel task. The new instance of the function has no effect on the interrupted instance. The two instances of the function do not interfere with each other. The interrupted instance of the function resumes processing at the same point that it was interrupted. This cycle can be performed any number of times.

If you want a function to be re-entrant, avoid writing to global and static variables.

All library functions are re-entrant, except for the standard I/O functions, dynamic memory allocation functions, and a few other functions as detailed in the lists of standard functions in Section 5.2.

I/O functions are not re-entrant since they use global opened file arrays, and each stream uses one I/O buffer that could be incorrectly maintained if both the interrupting, and the interrupted routines use the same stream.

Memory allocation functions, like **malloc()** are not re-entrant, since they use static variables to maintain the free-memory-list.

For a complete list of re-entrant and non re-entrant functions, see Section 5.2.

5.1.6 Initialization Requirements

An important issue in embedded systems software is the initialization of the program's data. The CompactRISC development tools provide a mechanism that supports the following type of initializations:

- Copying initialized data from ROM to RAM.
- Clearing the uninitialized data to binary zeros.

Refer to the *CompactRISC Toolset - Object Tools Reference Manual* for detailed explanation of data initialization.

You can choose to use, or not use, these initializations as part of your program. However you should be aware that part of the CompactRISC library functions assume that data initialization is performed at the beginning of the program. Library functions, which require any kind of data initialization, are indicated as such in the detailed descriptions below.

5.2 STANDARD ANSI C LIBRARY FUNCTIONS

The standard library conforms to the ANSI standard, with a few exceptions due to environment limitations, efficiency and re-entrancy considerations. For detailed descriptions of the functions, refer to any Standard C library documentation. The non-ANSI exceptions are:

- Time functions are not supported.
- Locale functions are not supported.
  The environment is set to the defaults defined in the ANSI standard. ASCII code is assumed. Multi-byte characters are not supported.

This section lists all the supplied standard library functions. It details any data-initialization requirements for each function, I/O low-level calls that are made by each function, and whether the function is re-entrant or not. It also contains additional information, extensions and exceptions from a standard ANSI C library reference manual. The list is organized according to the standard header-files containing each function prototype. Make sure that you use the appropriate header file when using each library function.

The standard header-files reside in the **`include`** directory, located in the CompactRISC development tools root directory.

assert.h

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
`assert()`	no	write()	no

The diagnostic message format of **`assert()`** is:

```
<file>:<source file> <failed condition> -- assertion failed
abort -- terminating
```

ctype.h

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
`isalnum()`	no	none	yes
`isgraph()`	no		yes
`isspace()`	no		yes
`isalpha()`	no		yes
`islower()`	no		yes
`isupper()`	no		yes
`iscntrl()`	no		yes
`isprint()`	no		yes
`isxdigit()`	no		yes
`isdigit()`	no		yes
`ispunct()`	no		yes
`tolower()`	no		yes
`toupper()`	no		yes

All functions have macro overrides. To make a function call instead of using the macro, use parenthesis around the function name. For example, **`(isdigit)(c)`** is not translated as a macro by the **`cpp`**.

errno.h Error codes, in addition to the standard **`EDOM`** and **`ERANGE`**, are detailed for each low-level function that uses them. Functions that use **`errno`** global variable are not re-entrant.

float.h CR32A uses 32-bit representation for float type, and 64-bit representation for double type, to conform to the IEEE 754 standard for floating point.

CR16 uses 32-bit representation for both float and double types.

math.h The CR32A compiler uses 64-bit representation for double type; the CR16 compiler uses 32-bit representation. Thus the precision of mathematical functions, from **`math.h`**. differs between cores.

For each double-precision function, there is an equivalent single-precision function. Single-precision functions have the same names as double-precision functions, with an added **f** (e.g., **acos()** becomes **acosf()**). In the CR16 libraries, all functions work in single precision, (e.g., **acos()** and **acosf()** are both the single precision version)

Mathematical functions use floating-point emulation routines. To use the mathematical library, invoke **crcc** with **-KFemulation**. This links your program with the floating-point emulation library. If you invoke **crlink** directly, specify **-lhfp** after **-lc**.

The mathematical functions are listed below:

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
acos()	no		yes
asin()	no		yes
atan()	no		yes
atan2()	no		yes
ceil()	no		yes
cos()	no		yes
cosh()	no		yes
exp()	no		yes
fabs()	no		yes
floor()	no		yes
fmod()	no		yes
frexp()	no		yes
ldexp()	no		yes
log()	no		yes
log10()	no		yes
modf()	no		yes
pow()	no		yes
sin()	no		yes
sinh()	no		yes
sqrt()	no		yes
tan()	no		yes
tanh()	no		yes

setjmp.h

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
setjmp()	no		no
longjmp()	no		no

signal.h

signal() and **raise()** only handle internal program-signals. External signals are not handled. **signal()** installs handlers that can be invoked by **raise()** from inside the program.

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
raise()	yes		no
signal()	yes		no

stdarg.h

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
va_arg	no		yes
va_start	no		yes
va_end	no		yes

stddef.h

Provides standard definitions.

stdio.h

File-handling functions and macros. Files are handled as streams, that hold file information, and maintain I/O buffering. In order to implement the I/O operations, these functions call some low-level I/O function (such as **read()**, **write()**, etc.), that interfaces with the host file-system through the debugger in development environment.

You can use the extension **fileno(FILE *fp)** to get the file descriptor associated with a stream. This returns the integer file-descriptor used by the stream.

A buffer is allocated for each file from the first I/O operation it involves, until you close the stream. You may use **setvbuf()** standard function to allocate your own buffer of any size. Unless you allocate the file by yourself, a buffer of size **BUFSIZ** (496 bytes) is allocated on the heap (by **malloc**). Be careful to allocate enough space to the heap in this case. If you are working simultaneously on N files (including **stdin**, **stdout** and **stderr**), you need 496 x N bytes on the heap just for the streams' buffers.

Stream functions are not re-entrant.

The **printf()** and **scanf()** **%p** conversion specification is treated as **%x**. **lp** does the same as **%lx**. You may use these qualifiers to print near or far pointers.

To print far strings, you can use the **%S** qualifier instead of the **%s** qualifier.

If you want to handle floating-point entities in formatted I/O you must invoke **crcc** with **-KFemulation**, or, if you invoke **crlink** directly, use the **-lhfp** flag. This links your program with the floating point emulation library **libhfp**. By default, programs are linked with a dummy floating point emulation library **libd**, in order to save code.

Among **stdio** functions, only **sprintf()**, **vsprintf()** and **sscanf()** are re-entrant.

Standard I/O functions require initialization of the program's data, in addition to the zero-initialization requirements detailed below.

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
clearerr()	no		no
fclose()	no	close()	no
feof()	no		no
ferror()	no		no
fflush()	no	write()	no
fgetc()	no	read()	no
fgetpos()	no	lseek()	no
fgets()	no	read()	no
fopen()	no	open()	no
fprintf()	no	write()	no
fputc()	no	write()	no
fputs()	no	write()	no
fread()	no	read()	no
freopen()	no	open(), close()	no
fscanf()	no	read()	no
fseek()	no	lseek()	no
fsetpos()	no	lseek()	no
ftell()	no	lseek()	no
fwrite()	no	write()	no
getc()	no	read()	no
getchar()	no	read()	no
gets()	no	read()	no
perror()	yes		no
printf()	no	write()	no
putc()	no	write()	no
putchar()	no	write()	no
puts()	no	write()	no
remove()	no	unlink()	no
rename()	no	rename()	no
rewind()	no	lseek()	no
scanf()	no	read()	no
setbuf()	no		no
sprintf()	no		yes
sscanf()	no		yes
tmpfile()	no		no
tmpnam()	yes		no
ungetc()	no		no
vfprintf()	no	write()	no
vprintf()	no	write()	no
vsprintf()	no		yes

The following functions have macro overrides:

fgetpos(), **fseek()**, **fsetpos()**, **ftell()**, **getc()**, **getchar()**, **putc()**, **putchar()**.

stdlib.h

The allocation functions (**malloc** and **calloc**) allocate blocks of memory space on the heap using the **sbrk()** function, as detailed below.

The macro **NULL** expands to **(void *) 0**.

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
abort()	no	close()	no
abs()	yes		yes
atexit()	yes		no
atof()	no		yes
atoi()	yes		yes
atol()	yes		yes
bserach()	no		yes
calloc()	yes		no
div()	yes		yes
exit()	yes	close()	no
free()	no		no
getenv()	no	getenv()	no
labs()	no		yes
ldiv()	no		yes
malloc()	yes		no
qsort()	no		yes
rand()	yes		no
realloc()	yes		no
srand()	yes		no
strtod()	no		yes
strtol()	no		yes
strtoul()	no		yes

sbrk() allocates memory in heap.

```
char *sbrk(int incr)
```

sbrk() allocates **incr** bytes of memory from the unallocated memory heap and returns the address of its lowest byte. The heap is defined as a continuous area which resides between two symbols - **_HEAP$_START** and **_HEAP$_MAX**, predefined in the linker directive file. For the CR16, the heap area must reside in the lowest 64 Kbytes of memory (far pointers are not handled by **sbrk**). By default, the heap area resides under the program stack, such that the stack grows down towards the heap, and the heap grows up towards the stack. In such a case, **sbrk()** en-

sures that the upper limit of the heap is at least 1024 bytes under the stack pointer, otherwise it does not allocate any more space. As there is no run-time check of the stack, you must take care to allocate enough space for both the heap and the stack, according to your specific requirements. See also the linker-directive file in the *CompactRISC Toolset - Object Tools Reference Manual*.

The **exit()** routine first calls **atexit()** to execute any functions that you selected for execution on program termination. It then calls either **_exit()** or **_eop()**. The **_exit()** function is called by **exit()** only if the program performed virtual I/O. It performs the necessary cleanup, i.e., file closing, before final exit. The **_eop()** function is mapped to the monitor, and is called for final exit.

string.h

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
memchr()	no		yes
memcmp()	no		yes
memcpy()	no		yes
memmove()	no		yes
memset()	no		yes
strcat()	no		yes
strchr()	no		yes
strcmp()	no		yes
strcpy()	no		yes
strcspn()	no		yes
strlen()	no		yes
strncat()	no		yes
strncmp()	no		yes
strncpy()	no		yes
strpbrk()	no		yes
strrchr()	no		yes
strspn()	no		yes
strstr()	no		yes
strtok()	yes		no
strerror()	no		yes

The CompactRISC CR16 library routines support only near pointers (16-bit). To handle far pointers, an additional set of functions is provided. These functions are of the form: **far_*original-function-name*** (i.e., **far_memcpy()** performs the same as **memcpy()**, but handles and returns far pointers). All these functions are listed below:

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
far_memchr()	no		yes
far_memcmp()	no		yes
far_memcpy()	no		yes
far_memmove()	no		yes

Routine	Zero-Initialization Required	I/O Low-level Calls	Re-entrant
far_memset()	no		yes
far_strcat()	no		yes
far_strchr()	no		yes
far_strcmp()	no		yes
far_strcpy()	no		yes
far_strcspn()	no		yes
far_strlen()	no		yes
far_strncat()	no		yes
far_strncmp()	no		yes
far_strncpy()	no		yes
far_strpbrk()	no		yes
far_strrchr()	no		yes
far_strspn()	no		yes
far_strstr()	no		yes
far_strtok()	yes		no

5.3 LOW-LEVEL I/O FUNCTIONS

To implement I/O and file-handling functions, the standard library calls low-level functions that have a direct interface to the monitor. These functions can be thought of as system calls; their interface is similar to, but not fully compliant with, that of the POSIX API. Thus we use the term “system calls” to refer to these low-level routines.

All I/O is performed via file descriptors, which are small integer numbers. When a program starts, the file descriptor 0 is associated with the console terminal in read mode (i.e., the keyboard) and file descriptors 1 and 2 are associated with the console terminal in write mode (i.e., the screen).

Programs open files on the host system with the **open()** function, and perform I/O and other file operations with the functions detailed below. All standard functions which handle I/O and files, call these functions.

These low-level functions are dependent on the development-board monitors. You may use them for debugging during the program development phase (e.g., writing error messages to the terminal, storing and retrieving results from files, etc.). However, a program that depends on these functions does not work in any other target system.

While these simulated “system calls”, and the libraries built on them, provide a very easy and conceptually clean interface, they may be too bulky for applications which do not require extensive I/O support. For such applications you must trim the library according to your needs.

The system calls documented here only work in conjunction with the CompactRISC Debugger. They use the debugger to perform I/O on the host file-system.

For independent programs, you can make your own I/O routines. This section provides the guide lines for making a system-dependent set of routines for any system. The rest of the library functions work correctly as long as the simulated system calls are replaced with compatible routines.

Two include files, **fcntl.h** and **unistd.h**, contain all the low-level I/O functions prototypes and related macros. Theses functions are described below:

fcntl.h Includes the **open()** function prototype, and the flags macro definition.

close closes a file

```
#include <unistd.h>

int close(int fd)
```

close() closes a file and removes its descriptor from the file descriptors reference table.

open opens a file for reading or writing or creates a new file

```
#include <fcntl.h>

open() (char *path, int flags, int mode)
```

open() opens the file **path** for reading and/or writing on the host system, as specified by the **flags** argument, and returns a descriptor for that file.

The **flags** argument may indicate that the file is to be created if it does not already exist (by specifying the **O_CREAT** flag).

path is the address of a string of ASCII characters representing a pathname, terminated by a null character.

To form the flags specified, OR the following values:

O_RDONLY	opens for reading only
O_WRONLY	opens for writing only
O_RDWR	opens for reading and writing
O_APPEND	appends to the file if exists
O_CREAT	creates file if it does not exist
O_TRUNC	truncates size to 0
O_EXCL	error if create and file exists

Opening a file, with **O_APPEND** set, appends to the end of the file, if the file exists. If **O_TRUNC** is specified, and the file exists, the file is truncated to zero length. If **O_EXCL** is set with **O_CREAT**, and the file already exists, **open()** returns an error.

This can be used to implement a simple exclusive access-locking mechanism. Upon successful completion, a non-negative integer termed a “file descriptor" is returned.

No program may have more than **_FOPEN_MAX** file descriptors open simultaneously.

In the event of an error, **errno** is set:

EPERM The pathname contains a character with the high-order bit set.

ENOTDIR A component of the path prefix is not a directory.

ENOENT **O_CREAT** is not set and the named file does not exist.

EACCES A component of the path prefix denies search permission.

EACCES The required permissions (for reading and/or writing) are denied for the named file.

EISDIR The named file is a directory, and the arguments specify it is to be opened for writing.

EROFS The named file resides on a read-only file system, and the file is to be modified.

EMFILE Too many open files.

ENXIO The named file is a character-special or block-special file, and the device associated with this special file does not exist.

ETXTBSY The file is a pure procedure (shared text) file that is being executed, and the **open()** call requests write access.

EFAULT **path** points outside the process's allocated address space.

EEXIST **_EXCL** has been specified and the file exists.

See Also **close(), lseek(), read() and write()**

unistd.h Includes prototypes of low-level functions detailed below.

lseek moves the read/write pointer

```
off_t lseek(int fildes, off_t offset, int whence)
```

The descriptor **fildes** refers to a file on the host system or device open for reading and/or writing.
lseek() sets the file pointer of **fildes** as follows:
If **whence** is **L_SET**, the pointer is set to **offset** bytes.
If **whence** is **L_INCR**, the pointer is set to its current location plus **offset**.
If **whence** is **L_XTND**, the pointer is set to the size of the file plus **offset**.
L_SET, L_INCR and **L_XTND** are defined in **sys/type-h**, automatically included in **unistd.h**.
Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

Return Value Upon successful completion, a non-negative integer, the current file pointer value, is returned. Otherwise, the file pointer remains unchanged, a value of −1 is returned and **errno** is set to indicate the error:

EBADF **fildes** is not an open file descriptor.

EINVAL **whence** is not a proper value.

EINVAL The resulting file pointer is negative.

See Also **open()**

read reads input

int read(int fildes, char *buf, int nbytes)

read() attempts to read **nbytes** of data from the object referenced by the descriptor **fildes** into the buffer pointed to by **buf**.

The **read()** starts at a position given by the pointer associated with **fildes**, see **lseek()**.

Upon return from **read()**, the pointer is incremented by the number of bytes actually read.

Upon successful completion, **read()** returns the number of bytes actually read and placed in the buffer.

The system guarantees to read the number of bytes requested if the descriptor references a file which has that many bytes remaining before the end-of-file, but in no other cases.

If the returned value is 0, then end-of-file has been reached.

Return Value If successful, the number of bytes actually read is returned. Otherwise, −1 is returned and the global variable **errno** is set to indicate the error:

EBADF **fildes** is not a valid file descriptor open for reading.

EFAULT **buf** points outside the allocated address space.

See Also **open()**

rename changes the name of a file

```
int rename(char *old, char *new)
```

rename() changes the file named **old** to a file named **new**. The filename **old** is effectively removed. If **new** already exists, **rename()**'s behavior is undefined.

Return Value **rename()** returns zero if the operation succeeds, nonzero if it fails. The global variable **errno** indicates the reason for the failure. See the list of Return Values below.

unlink removes directory entry of a file

```
unlink(char *path)
```

unlink() removes the file on the host system whose name is given by **path**.

Return Value Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and **errno** is set to indicate the error:

- **EPERM** The path contains a character with the high-order bit set.
- **ENOENT** The pathname is too long.
- **ENOTDIR** A component of the path prefix is not a directory.
- **ENOENT** The named file does not exist.
- **EACCES** Search permission is denied for a component of the path prefix.
- **EACCE** Write permission is denied on the directory containing the link to be removed.
- **EPERM** The named file is a directory and the effective user ID of the process is not the superuser.
- **EBUSY** The entry to be unlinked is the mount point for a mounted file system.
- **EROFS** The named file resides on a read-only file system.
- **EFAULT** **path** points outside the process's allocated address space.
- **ELOOP** Too many symbolic links have been encountered in translating the pathname.

See Also **close()**

write writes to a file

```
write(int fildes, char *buf, int nbytes)
```

write() attempts to write **nbytes** of data to the object referenced by the descriptor **fildes** from the buffer pointed to by **buf**.

write starts at a position given by the pointer associated with **fildes**, see **lseek()**.

Upon return from **write()**, the pointer is incremented by the number of bytes actually written.

Return Value Upon successful completion, the number of bytes actually written is returned. Otherwise, the file pointer remains unchanged, −1 is returned and **errno** is set to indicate the error:

EBADF **fildes** is not a valid descriptor open for writing.

EFBIG An attempt is made to write a file that exceeds the process file size limit or the maximum file size.

See Also **lseek()** and **open()**

5.4 32-BIT EMULATION

The CR16 compiler issues these function calls to emulate each long int operation.

Unless it is essential, avoid using long variables in CR16 programs, because this involves extra emulation code.

The library includes the following functions:

```
long __lshlsi3(long, int) /* logical left shift */

long __lshrsi3(long, int) /* logical right shift */

long __ashlsi3(long, int) /* arithmetic left shift */

long __ashrsi3(long, int) /* arithmetic right shift */

long __mulsi3(long, long) /* multiplication */

long __divsi3(long, long) /* signed division */

unsigned long __udivsi3(unsigned long, unsigned long)
                                    /* unsigned division */

long _ _modsi3(long, long) /* signed remainder */

unsigned long __umodsi3(unsigned long, unsigned long)
                                    /* unsigned remainder */
```

5.5 DIVISION EMULATION

Division and remainder operations are performed by software emulation routines. The compiler replaces each division, or remainder, operation with these function calls:

```
int _fast_quo(int, int) /* division of two integers */

int _fast_uquo(unsigned int, unsigned int)
                        /* division of two unsigned integers */

int _fast_rem(int, int) /* remainder from two integers division */

int _fast_urem(unsigned int, unsigned int)
                        /* remainder from two unsigned integers
                        division*/
```

5.6 FLOATING-POINT EMULATION

The Floating-Point Emulation Library (**libhfp**) is used to create floating-point programs for those CompactRISC micro-processors that lack Floating-Point Unit hardware (FPU). **libhfp** is a library of floating-point arithmetic emulation routines. It provides an efficient, low-cost, floating-point solution for systems without an FPU, by emulating floating-point unit instructions in software.

The CompactRISC **libhfp** is partly based on floating-point emulation code distributed among the source files of **gcc** (GNU C Compiler).

The **libhfp** floating-point emulation library supports virtually all the arithmetic operations (see Table 5-1) of the IEEE 754 format floating-point operands. However, **libhfp** does not support the full flexibility advocated by the ANSI *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985), such as a selection of four rounding modes. Issues of compatibility and conformity to IEEE/754 standards are discussed in Section 5.6.1.

Using **libhfp** has an adverse effect on the size of your program code. Floating-point operation is translated to an emulation function call, whose code is included in the final executable object file.

5.6.1 libhfp Technical Specifications

The **libhfp** library consists of fast emulation routines for the CR floating-point instructions.

When a program is compiled with the **-KFemulation** option, the CompactRISC compiler puts floating-point entities into an integer register (or register pair if needed) and generates an emulation call whenever a floating-point operation is required. The floating-point operands are passed to the emulation routine in integer registers, using the standard calling convention (see Appendix A). You can examine the generated code by using the **-S** compiler option.

The emulation routines are re-entrant; floating-point code may therefore be used in signal and interrupt handlers.

Conformity to IEEE/754

The **libhfp** library is arithmetically compatible to the round-to-nearest rounding mode of the IEEE 754 standard. It implements single and double-precision floating-point numbers, using the IEEE 754 standard formats.

In particular, the following ANSI/IEEE 754 features are emulated:

- Basic single-precision format (float)
- Basic double-precision format (long)
- Signed zero
- Round-to-nearest

The following IEEE/754 features are not supported by **libhfp**:

- Special (NaN, denormalized, infinity) arithmetic
- Round towards + ∞, round towards − ∞, round towards zero
- Unordered compare
- Inexact exception
- Reserved operand exception by the **cmpf** and **cmpl** instructions
- Division-by-zero exception
- Invalid operation exception
- Overflow/underflow exceptions

In addition, the **libhfp** emulation routines have several other attributes which may impact the way your code works:

- Floating-point emulation is re-entrant and interruptible. Thus interrupt handlers may also use floating-point instructions. However, an access to double-precision floating-point variables is not always an atomic operation. This means that some code (such as an interrupt handler) may cease to work, if it relies on the fact that the high-order four bytes and the low-order four bytes of a global double precision variable are *atomically* consistent (Floating-point data cannot have the *volatile* property).

- Most floating-point instructions are emulated by **libhfp** routine calls. This means that the stack *above* the stack-pointer is corrupted. This is normally not a problem.

The `libhfp` interface

The CompactRISC C Compiler issues **`libhfp`** function calls as follows:

The operands of the floating-point operation are passed to the emulation routine in a group of consecutive integer registers, according to the standard calling convention (see Appendix A). Double-precision operands are passed in consecutive pairs of registers, lower-order bits (mantissa lower part) are passed in the lower numbered register, and higher order bits (exponent + sign + high-order bits of mantissa) are passed in the higher numbered register.

In the CR32A architecture, the result of the floating-point operation performed by the routine is returned in **`r0`** if it is a single-precision operand, or in **`r0`** - **`r1`** if it is a double-precision operand.

In the CR16 architecture, the result is always returned in **`r0`** - **`r1`**.

Example 1

The following code:

```
float     i,j,k;
i = j*k;
```

is transformed in CR32A to:

```
loadd     _j,r2
loadd     _k,r3
bal       ra,__mulsf3
stord     r0, _i
```

Example 2

The following code:

```
double    i,j,k;

i = j*k;
```

is transformed in CR32A to:

```
loadd     _j,r2
loadd     _j+4,r3
loadd     _k,r4
loadd     _k+4,r5
bal       ra,__adddf3
stord     r0,_i
stord     r1,_i+4
```

Table 5-1. Instructions Emulated By Calls to `libhfp` Routines

Operation	Emulation Routine
Add two floats	`float __addsf3 (float a1, float a2)`
Add two doubles *	`double __adddf3 (double a1, double a2)`
Subtract two floats	`float __subsf3 (float a1, float a2)`
Subtract two doubles *	`double __subdf3 (double a1, double a2)`
Multiply two floats	`float __mulsf3 (float a1, float a2)`
Multiply two doubles *	`double __muldf3 (double a1, double a2)`
Divide two floats	`float __divsf3 (float a1, float a2)`
Divide two doubles *	`double __divdf3 (double a1, double a2)`
Negate a float	`float __negsf2 (float a1)`
Negate a double *	`double __negsf2 (double a1)`
Convert double to float *	`float __truncdfsf2 (double a1)`
Convert float to double *	`float __extendsfdf2 (double a1)`
Convert long to float	`float __floatsisf (register long a1)`
Convert long to double *	`float __floatsidf (register long a1)`
Convert float to long	`long __fixsfsi (float a1)`
Convert double to long *	`long __fixdfsi (double a1)`
Convert float to unsigned long	`unsigned long __fixunssfsi (float a1)`
Convert double to unsigned long *	`unsigned long __fixunsdfsi (double a1)`
Return 0 if a1 = a2	`int __eqsf2 (float a1, float a2)`
Return non-zero if a1 ≠ a2	`int __nesf2 (float a1, float a2)`
Return value > 0 if a1 > a2	`int __gtsf2 (float a1, float a2)`
Return value ≥ 0 if a1 = a2	`int __gesf2 (float a1, float a2)`
Return value < 0 if a1 < a2	`int __ltsf2 (float a1, float a2)`
Return value ≤ 0 if a1 = a2	`int __lesf2 (float a1, float a2)`
Return 0 if a1 = a2 *	`int __eqdf2 (double a1, double a2)`
Return non-zero if a1 ≠ a2 *	`int __nedf2 (double a1, double a2)`
Return value > 0 if a1 > a2 *	`int __gtdf2 (double a1, double a2)`
Return value ≥ 0 if a1 = a2 *	`int __gedf2 (double a1, double a2)`
Return value < 0 if a1 < a2 *	`int __ltdf2 (double a1, double a2)`
Return value ≤ 0 if a1 = a2 *	`int __ledf2 (double a1, double a2)`

* only in CR32A

Exception handling

The following exception handling is performed by the **libhfp** routines:

- An overflow in the result of a floating-point operation returns infinity (either positive or negative, according to the sign of the result).
- An underflow (denormalized number) in the result of a floating-point operation returns zero (positive or negative, according to the sign of the result).
- A zero divisor in a division routine (_ **_divsf3**, _ **_divdf3**) returns quiet NAN.

5.6.2 Floating-point Emulation Examples

The following examples illustrate compiling and linking with floating-point emulation:

Example 1 Compile and link the **whetstone.c** benchmark program with **libhfp**.

```
crcc whetstone.c -O -KFemulation -o whetstone
```

Example 2 Assemble and link the **w.s** program with **libhfp**.

```
crasm w.s
crlink w.o -lc -lhfp
```

Chapter 6
STANDARD CALLING CONVENTIONS

6.1 CALLING CONVENTION

The calling convention is defined as part of the CompactRISC architecture, and is supported by the CompactRISC Development Tools. The calling convention consists of a set of rules which form a handshake between different pieces of code (subroutines), and define how control is transferred from one to another. It thus defines a general mechanism for calling subroutines and returning from subroutines.

If you develop your entire application in the C language, you may not be aware of the calling convention since it is handled by the CompactRISC C Compiler. To ensure compatibility between a calling subroutine (C function) and a called subroutine use ANSI C function prototypes. This ensures that a call to a function is compatible with its definition. In other words, it ensures that all arguments are correctly transferred from the calling function to the called function.

6.1.1 Calling a Subroutine

The CompactRISC architecture usually uses the **bal** or **jal** instruction to call a subroutine. Each of these instructions performs two operations:

- Saves the address of the following instruction in a specified general-purpose register (pair of general-purpose register in CR16B large model). This address is used as a return address for the called subroutine. According to the calling convention, the return address is always saved in the **ra** register (**era-ra** register pair in CR16B large model).
- Transfers control to a specified location in the program (the subroutine address).

Example 1

```
bal   ra, s      # call the subroutine "s"
```

or

```
jal   ra, r7     # call the subroutine whose address is stored in r7
```

Example 2 For CR16B large model only:

```
bal   (era,ra), s# call the subroutine "s"
```

or

```
jal   (era,ra), (r8,r7)# call the subroutine whose address is
                       # stored in the register pair r7-r8
```

6.1.2 Returning from a Subroutine

The **jump** instruction is used to return from a subroutine. The program jumps to the return address, which is stored in the **ra** register (**era-ra** register pair in CR16B large model).

Example 1

```
jump ra    # return to caller
```

Example 2 For CR16B large model only:

```
jump (era,ra)# return to caller
```

In the CR16B, the popret instruction might also be used to return from a function:

```
popret $1,ra
```

6.2 CALLING CONVENTION ELEMENTS

The elements of the calling conventions are as follows:

6.2.1 Passing Parameters to a Subroutine

The calling sequence loads some of the arguments to registers according to a predefined convention. The registers used are the integer registers **r2**, **r3**, **r4**, **r5**.

The arguments list is comprised of arguments which can be passed in registers (qualified arguments) and arguments which can not be passed in registers (non qualified arguments).

Qualified arguments Qualified arguments are of the following types:

- Integer types, pointer types.
- Structures with size less than, or equal to, the size of two registers (e.g., 4 bytes in the CR16 compiler and 8 bytes in the CR32A compiler), and aligned to the size of one register.
  If a structure member must be split between two registers, the structure is disqualified.

- In the CR16 compiler, long integers arguments and far pointers arguments are passed in a pair of registers.

Non-qualified arguments

Non-qualified arguments are of the following types:

- C structures or unions, other than those specified above.
- Arguments for which all the parameter registers are already allocated to other arguments.

6.2.2 Parameter Passing Algorithm

The following algorithm determines how parameters are passed to a given routine:

- The parameter list is scanned from left to right.
- Registers are allocated in ascending order (i.e., **`r2`** is allocated before **`r3`** etc.).
- A parameter, which is qualified to be passed in a register, is allocated the next free register (in the range **`r2`**-**`r5`**).
- If a structure is to be passed in registers, its layout within the registers is as in memory. Fields are allocated in the order they are declared, starting at the lower register. If two members reside in the same register, the first is put in the least significant bits. The alignment of members within the structure is kept by also passing the padding bytes.
- If a parameter cannot be passed in a register (either because it is not qualified, or because the registers have been entirely allocated to previous parameters), it is passed on the stack. Each parameter on the stack is aligned to a size which is fixed per architecture (2 for CR16, 4 for CR32A). This alignment is relative to the address of the first parameter on the stack, and not absolute.

Note

A type mismatch between formal and actual parameters may produce incorrect results. Use function prototypes to check parameter consistency at compile time.

There is one exception to the rule stated above:

For C routines with a variable number of arguments, that are declared as such using the ANSI C VARARGS declaration, all parameters are put on the stack by the calling routine. None of the arguments are passed in a register.

The compiler recognizes some library routines as variable-number-of-arguments routines (e.g., **`printf`**).

Note Failure to declare a routine with a variable number of arguments as such, may result in erroneous results due to a mismatch between caller and callee.

6.2.3 Returning a Value

A subroutine can return one value to its caller. The calling convention uses the **r0** register for passing the return value to the caller.

For example, consider the following C code:

```
return 5;
```

The assembly code generated from this line is:

Example 1

```
movw $5, r0     # pass return value
jump ra         # return to caller
```

Example 2 For CR16B large model only:

```
movw $5, r0     # pass return value
jump (era,ra)   # return to caller
```

When the returned value is of size greater than that of a single register, the **r0** - **r1** pair is used (**r0** having the lsb). This is the case for CR16 longs, single-precision fp values, far pointers, and for CR32A double-precision fp values.

The only exception to this rule is a function that returns a structure. In this case, the calling function puts the address of a structure in which to store the result in **r0**, and is responsible for the allocation needed. The called function then uses **r0** as a pointer to the resulting structure

6.2.4 Scratch and Non-scratch Registers

The calling convention divides the registers into two groups:

Scratch registers. Any of these registers can be freely modified by any subroutine, without first saving a backup of their previous value. The caller cannot assume that their value will remain the same after a subroutine has returned. If, for any reason, the caller needs to keep this value, it is responsible for saving the scratch register, either on the stack or in a non-scratched register, before calling the subroutine, and to restore it after the subroutine has returned.

Non-scratch registers. Before using any of these registers, a subroutine must first store its previous value on the stack. On returning to the caller the subroutine must restore this value. The caller can always assume that these registers are clobbered by any subroutine that it has called.

The calling convention defines **r0** - **r6** as scratch registers. All the other general-purpose registers are defined as non-scratch, except **sp** which is used as a stack pointer and thus does not belong to any of these categories.

Note

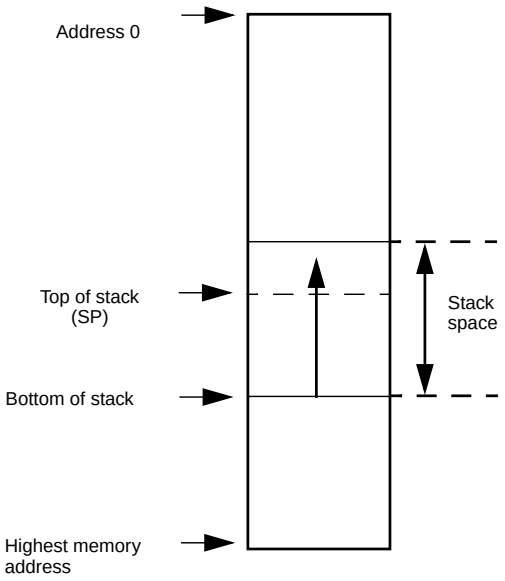
An exception to the rule for using scratch registers is an interrupt/trap subroutine. This kind of subroutine must always save and restore any scratch register that can be used during the interrupt trap. This is because there is no real caller. The interrupt, or trap, suspends another subroutine which is not aware of, or prepared for, this interception. To protect it, its scratch registers must be saved and restored so that the interrupt or trap is transparent.

6.2.5 Program Stack

The program stack is a contiguous memory space that can be used by your program for:

- Allocating memory for local variables which are not in registers
- Passing arguments in special cases (see passing arguments to a subroutine)
- Saving registers before calling a subroutine, or after being called (see scratch registers)

The stack is a dynamic memory space which begins at a fixed location (stack bottom) and grows towards lower memory addresses. Its lowest address (also called top of stack) is changed dynamically and is pointed to by the Stack Pointer register (**SP**). The stack pointer is kept aligned, at all times, to the internal bus width, e.g., 2 for CR16, 4 for CR32A.



Any subroutine can allocate space on the stack by changing the value of the SP register to adjust the top of stack. When this subroutine returns it must restore the **SP** to its previous value, thereby releasing the temporary space that it had occupied on the stack during its life-time.

In your application program, allocate space for the program stack, and initialize the sp register to the stack bottom. This is done in the start-up routine. For more details about the start-up routine, see Section 5.7 in the *CompactRISC Toolset - Introduction*.

6.2.6 Alignment of Variables

Local Variables. Local variables on the stack are aligned to the minimum between their size and the bus width, as specified by the **-KB** option. The default bus width is the internal bus width of the specific architecture i.e., 2 for CR16 and 4 for CR32A.

Parameters on Stack. Parameters that are passed on the stack are always aligned to the default bus width (2 for CR16, 4 for CR32A). The alignment is relative to the bottom of the stack. Parameter alignment is not affected by any compiler option and this ensures compatibility between functions from different modules.

Structures. Members are aligned within the structure, relative to the beginning of the structure. Each member is aligned to the minimum between its size and the structure member alignment, as specified by the **-J** compiler option. The default value for structure member alignment is 2 bytes for CR16 and 4 bytes for CR32A. Padding bytes are inserted before fields to keep their alignment, if necessary. The structure itself is aligned in memory to the smaller of the size of its largest member and the structure member alignment. In addition, the structure is padded such that its total size is a multiplication of its alignment boundary.

Since alignment of structure members is controlled by the **-J** option, C modules which access the same structures must be compiled with the value for the **-J** option. In general, we recommend compiling all the modules in your program with the same value. Do not worry about the CompactRISC libraries, since their structures were designed to be unaffected by the value of the **-J** option.

Unions. A union is aligned to its largest member. The union is padded such that its total size is a multiplication of its alignment.

Bit-Fields. Bit-fields are packed consecutively, in groups. Each such group is aligned to the minimum between the base-type size of the first bit-field in the group, and the struct-align-width set by the **-J** option. Bit-fields are then packed consecutively, as long as they remain in the same memory unit whose size is the maximum between the group first bit-field base-type and the struct-align-width.

If, because of the bit-fields, the structure does not terminate on a byte boundary, padding bits are added to fill it up to the end of the last byte it occupies. Additional padding bytes may be needed as mentioned above.

Strings. Strings are aligned to a word.

Chapter 7
GUIDELINES FOR USING THE COMPILER

7.1 INTRODUCTION

The following sections are provided as guidelines for using the CompactRISC C Compiler.

7.2 OPTIMIZATIONS

Experienced programmers should understand this compiler's optimization techniques in order to:

- Avoid using programming tricks based on the way ordinary compilers generate code.
- Avoid performing "manual optimizations" that the optimizer does anyway.
- Avoid writing code that may prevent certain optimizations.
- Select command line optimization options for optimal performance.

See Chapter 3 for a complete description of the optimization techniques.

Optimization options

The **-O** option enables the optimizer. Specify **-O** on the command line for the fastest possible code without an undue increase in code size.

In special cases, e.g., compiling a compact code with space limitations, you may need to refine the optimization phase by specifying optimization options.

Individual optimization options can be specified by adding additional options.

The parser and the code generator perform some local optimizations, even when optimization is not enabled.

Default optimization options

There is normally no reason to turn off any of the optimization options; the default produces the best results.

Speed optimization Use the following options to get the best time performance from your program:

`-O, -ON, -finline-functions.`

Space optimization Use the following options to compile your program with the minimum size:

`-Os, -msave-emul` (see also “Space optimization” on page 3-3)

7.3 PORTING EXISTING C PROGRAMS

Most programs, which run when compiled by other C compilers, compile and run on the CompactRISC C Compiler with no change in the sources. However, a few programs may perform differently, when compiled by the compiler. In addition, some programs which seem to work when compiled without the optimizer, may not work when optimized. The following sections describe some of the reasons for this phenomenon.

Undetected program errors The single most-common reason for a non-functioning program is an undetected program error, which only becomes apparent when compiling under a different compiler, or when optimizing. Many such errors result from the program author relying on the way the compiler compiled, and thus creating a program which is clearly non-portable.

Some of the most common problems are:

- Uninitialized local variables.
  Since the memory and register allocation algorithms of the CompactRISC C Compiler are very different from those of other compilers, a local variable may be put in a completely different place. For example, a programmer may not initialize a local variable, assuming that it contains zero at the start of the program. This may become false as a result of the register allocation phase of the CompactRISC C Compiler.

- Relying on memory allocation
  You cannot assume that two variables will be allocated in the same order they are declared. Thus a program that uses address calculations to go from one declared variable to another declared variable may not work.

- Failing to declare a function
  A **`char`** returning function returns a value in the lower-order byte of R0, without affecting the other bytes. Failure to declare that function where it is used may result in an error.

Example assume that **`get_code()`** is defined to return a **`char`**, then

```
main() {
     int  i;
     if ((i = get_code()) == 17)  do_something();
}
```

might never execute **do_something** even if **get_code** returns **17** since the whole register is compared to **17,** not just the low-order byte.
A similar problem exists for functions which return **short** or **float**, or those which return a structure

Compiling embedded application code

Embedded application code is distinguished from general "high-level" code, by the fact that it is machine-dependent, often contains real-time aspects and interspersed **asm** statements, and is often driven by asynchronous events, such as interrupts. Examples of such code are interrupt routines and device handlers. From the optimizer's point of view, ordinary looking global variables can actually be semaphores or memory-mapped I/O, that can be affected by external events, which are not under the optimizer's control. Even so, it is still possible to optimize such code, by taking some precautions, and by activating some special optimization options. Some of these aspects are discussed in the following sections.

- Volatile variables
  Volatile variables are variables which might be used or changed by asynchronous events, such as I/O or interrupts. The **-Oi** qualifier treats all global variables, static variables, and pointer dereferences as volatile, which means that they are not subject to any optimizations. As a result, the number and nature of memory references to them do not change. Remember that individual identifiers can be declared as volatile by using C volatile type qualifiers. The following examples demonstrate the consequences of volatile variables and pointer dereferences.

Examples

1. `x = 17; x = 18;`

   If **x** is volatile, both of the two assignments to **x** are executed even though the first one seems redundant.

2. **x = 9**
   y = x + 1;

   If **x** is volatile, this program segment is not optimized to:

   y = 10;

3. **short count;**
   count++;

   The CompactRISC-family processors do not support atomic memory operations. If changing the value of **count** should be an "atomic" operation, it is not sufficient to declare **count** as volatile. This command is interpreted as:

```
loadw          _count,r0
addw           $1,r0
storw          r0,_count
```

To make **count++** a real atomic command, see Section 7.10.6 (semaphores).

Timing assumptions Optimizing a program changes the timing of various constructs. In particular, delay-loops may run faster than before.

Relying on register order A program that relies on the fact that a given register variable resides in a specific register must be compiled using the **-Ou** option. If this option is not used, a request to allocate a specific register for a specific variable may not be honored.

Relying on frame structure A program, that relies on a specific frame structure, may not work.

Referring to variables on the frame of a different function (such as the caller of this function) by complex pointer arithmetic may also cease to work. See Chapter 6 for more details.

Using asm statements The code inserted by **asm** statements may cease to work because the surrounding code produced by the CompactRISC C Compiler is normally different from another compiler's code. See Section 7.6.

7.4 DEBUGGING OPTIMIZED CODE

Most of the time, you should not need to debug an optimized program. The majority of bugs can be found before optimization. However, there are some rare bugs which appear only when the optimizer is introduced; bugs that are difficult to find without a debugger.

The problem is that code motion optimizations and register allocation obsolete most of the symbolic debugging information generated by the compiler. The following "rules of thumb" can be employed when using symbolic debug information together with the optimizer:

- Line number information is correct, but the code performed at the specified lines may be different from non-optimized code as a result of various code motion optimizations, e.g., moving loop invariant expressions out of loops.
- Symbolic information for global variables, and variables whose address is taken, is always correct since they are never put in registers.
- Local variable values can be printed. However you must exercise caution when the variable is not active since it may contain an invalid value.

It is helpful to have an assembly listing of the program in question which has been compiled with the **-S** and **-n** qualifiers. Such a listing contains the assembly code with C code annotations.

7.5 ADDITIONAL GUIDELINES FOR IMPROVING CODE QUALITY

The following programming guidelines show how to take advantage of the CompactRISC C Compiler optimizations.

Integer variables

Many operators, including index calculations, are defined in C to operate on integers, and imply a conversion when given non-integer operands. Therefore, to avoid frequent run-time conversions from **char** to **int**, define integer variables, particularly variables which serve as array-indices, as type **int** and not **long** or **char**.

Local variables

Use local variables as much as possible, particularly when they are employed as loop counters or array indices, as they have a chance of being placed in registers.

Global variables

Optimizations are not carried out on global variables. If you use a global variable extensively in a routine, we recommend that you assign its value to a local variable at the beginning of the routine, and, if necessary, store it back at the end. This enables the compiler to optimize references to this variable.

Floating-point computations

In programs which do not require double-precision floating-point computations, you can achieve a significant run-time improvement by paying attention to the following points:

- define all functions as returning float type, not double
- define all constants to be 'float' using the **f** suffix or cast expressions explicitly to float
- use the single precision version of the standard floating-point routines such as **fabs()** instead of **abs()**, **fsin()** instead of **sin()**, etc.

Common subexpression optimization

The optimizer normally recognizes multiple uses of the same expression, and saves that expression in a temporary variable (usually a register). This cannot be done when worst case assumptions are made about references or changes of value. Expressions that contain pointer dereferences or global variables are vulnerable; therefore, if many uses of the same expression span across procedure calls, it is advisable to save them in local variables.

Example

```
foo1(p->x);
foo2(p->x);
```

If **p** is global, the expression **p->x** cannot be recognized by the optimizer as a common subexpression because **foo1()** may change its value. The following manual optimization may help:

```
t = p->x;            /* t is local, therefore */
foo1(t);             /* not potentially defined by foo1() */
foo2(t);             /* so its value is still valid for foo2()
*/
```

Here you are using your knowledge that **p->x** is not changed by **foo1()** to make this optimization. The optimizer cannot do the same because it assumes the worst case.

7.5.1 Long Functions

If you write a long function, for which the optimizer is unable to provide sufficient registers for local variables, performance is degraded. There are two ways to avoid this problem:

- Split long functions into several shorter ones. This assists the optimizer, and is usually good programming practice.
- Use different local variables for different tasks.

Example

```
foo()
{ int i;
      for (i=0; i<5000;i++)          /* first use of i  */
            {...}
/*... code that does not use i*/
      for (i=0; i<40;i++)            /* second use of i */
            {...}
}
```

in this example, the second loop could use another local variable (say **j**) as loop index. This kind of variable splitting can assist the optimizer in producing better code.

7.5.2 Register Allocation

The C language is unique in that the programmer can specify (or rather recommend) that some variables be allocated to machine registers. The optimizer normally ignores these recommendations, since it turns out that in most cases the optimizer's own register allocation algorithms are as good as, or superior to, the programmer's recommendations. There are several reasons for this:

- You can use a register for one variable only. The optimizer however allocates a register along live ranges of variables, making it possible for several variables with non-conflicting live ranges to use the same register.
- You can allocate variables in safe registers only (i.e., not clobbered by a function call). Therefore, every register which is used has to be saved/restored at the entry/exit of the procedure. The optimizer allocates variables that do not live across procedure calls in unsafe registers. Therefore, these registers need not be saved/restored.

7.5.3 Long Variables

Most 32-bit operations are not supported by the CR16 architecture. You should, therefore, avoid using long variables as much as possible as this may require calls to emulation routines.

7.5.4 Bit-field Operations

The manipulation of bit fields requires a lengthy sequence of CompactRISC instructions. We advise you to optimize such references manually.

Example

```
struct {int a:4; int b:4:} s;
```

Replace:

```
s.a = 0:
s.b = 0;
```

by:

```
*(char*)&s = 0;
```

Note: the CR16B architecture has bit manipulation instructions, thus the above bit assignments are implemented as atomic operations.

7.6 ASM STATEMENTS AND INTRINSIC FUNCTIONS

There are two kinds of **asm** statements, each of which requires different treatment.

Regular asm statements

A regular **asm** statement has the form **__asm__("asm string")**. The compiler puts the string, as is, in the assembly output. Take great care, if you use this kind of **asm** statement, since you are interfering in the compiler code generation.

Advanced asm statements

With advanced **asm** statements it is possible not only to embed an assembly instruction into the compiler generated code, but also to provide information to the compiler regarding the side effects of this instruction. For example, it is possible to report to the compiler which registers are clobbered, whether memory is modified or not, and whether condition code is modified or not. The full syntax of advanced asm statements is beyond the scope of this manual. Readers who are interested in this information can find it in *Using and Porting GNU CC*.

For your convenience, the CompactRISC toolset provides a set of macros, which are implemented using advanced **asm** statements, and enable you to safely embed any CompactRISC instruction into your C code. This is discussed in the following sections.

High-level interface to machine instructions

A high-level interface is provided for most CompactRISC machine instructions in the header file **asm.h**. This file contains a set of macros which are implemented using advanced **asm** statements. These macros allow usage of machine instructions in conjunction with C variables. Each macro may expand to one, or more, machine instructions depending on the context. The usage of advanced **asm** statements in these macros ensures that the assembly instructions are safely embedded into the code, and do not hurt compiler optimizations.

To embed the CompactRISC instruction **addw**, write:

```
#include <asm.h>
short i,j;
...
_addw_(i,j);
```

More useful examples are the **_spr_** and **_lpr_** macros. They provide access to the special purpose registers (e.g., **PSR**, **ISP**), which is not otherwise available from a C program. For example:

```
#include <asm.h>
unsigned int tmp;
...

/* Store the PSR contents in variable 'tmp' */
_spr_("psr", tmp);
```

Two other useful macros, **set_i_bit** and **clear_i_bit**, are available for a specific purpose: setting and clearing the **PSR.I** bit. This bit must be set after reset in order to enable interrupts. In addition, trap and interrupt routines may want to set and clear this bit to enable and disable nested interrupts. For example;

```
#include <asm.h>
...
set_i_bit(); /* Set the PSR.I bit */
```

The following table lists all the macros which are available in **asm.h**.

addb	_bhs_	_loadw_	_sgt_
addw	_blt_	**_loadd_** (CR32)	_sle_
addd (CR32)	_bge_	_lshb_	_sfs_
addub	_br_	_lshw_	_sfc_
adduw	_cmpb_	**_lshd_** (CR32)	_slo_
addud (CR32)	_cmpw_	_movb_	_shs_
addcb	**_cmpd_** (CR32)	_movw_	_slt_
addcw	_di_	**_movd_** (CR32)	_sge_
addcd (CR32)	_ei_	_movxb_	_storb_
andb	_excp_	_movzb_	_storw_
andw	_jeq_	**_movxw_** (CR32)	**_stord_** (CR32)
andd (CR32)	_jne_	**_movzw_** (CR32)	_subb_
ashub	_jcs_	_mulb_	_subw_
ashuw	_jcc_	_mulw_	**_subd_** (CR32)
ashud (CR32)	_jhi_	**_muld_** (CR32)	_subcb_
beq	_jls_	_nop_	_subcw_
bne	_jgt_	_orb_	**_subcd_** (CR32)
bcs	_jle_	_orw_	_tbit_
bcc	_jfs_	**_ord_** (CR32)	_xorb_
bhi	_jfc_	_retx_	_xorw_
bls	_jlo_	_seq_	**_xord_** (CR32)
bgt	_jhs_	_sne_	_wait_
ble	_jlt_	_scs_	_lpr_
bfs	_jge_	_scc_	_spr_
bfc	_jump_	_shi_	set_i_bit
blo	_loadb_	_sls_	clear_i_bit

7.7 SETJMP()

When you use **setjmp()** and **longjmp()**, the only automatic variables guaranteed to remain valid are those declared volatile. This is a consequence of automatic register allocation. Consider the function:

```
#include <setjmp.h>
jmp_buf j;
foo()
{
      int a
      a = fun1();
      if (setjmp(j))
           return a;
      a = fun2();
      /* longjmp(j) may occur in fun3(). */
      return a + fun3();
}
```

Here **a** may, or may not, be restored to its first value when the **longjmp()** occurs. If **a** is allocated in a register, its first value is restored; otherwise, it retains the last value stored in it. If you use both the **-W** and **-O** options together, the CompactRISC C Compiler issues a warning if such a problem is likely to occur.

7.8 OPTIMIZING FOR SPACE

By default, the CompactRISC C Compiler optimizes for optimal speed without undue increase in code size. There are several things that can be done to improve code density:

- Optimize with the **-Os** option
- Do not use loop-unrolling optimization.
  The optimizer uses a heuristic approach based on size considerations, whether to perform loop-unrolling. Nevertheless, if code density is important, it is advisable not to use the loop-unrolling optimization.
- Squeeze all structure definitions by using the **-J1** option.
- Use the `-sbrel` option to force register-relative addressing mode for accessing all non-constant global and static variables (CR32), or a selected group (using `#pragma sbrel`) of variables (CR16).

7.9 COMPILATION TIME REQUIREMENTS

Using the optimizer slows down the compilation process. We therefore recommend that you use the optimizer only on final production versions of a program. The resources required (time and memory) vary strongly from program to program and actually depend on the size of the routines in the compiled program file. The larger a routine, the more time and memory needed to optimize it.

7.10 EMBEDDED PROGRAMMING HINTS

The CompactRISC C Compiler provides several features which allow for programming of embedded applications in C. These features help solve the following issues:

- full control over memory allocation - including RAM, ROM, stack space, trap and interrupt vectors, peripheral memory-mapped control registers.
- start-up actions performed at system reset - including initializing stack pointers, configuration registers, peripheral control registers, and timers.
- initialization of RAM data variables - usually by copying from ROM or by zeroing.
- interrupt/trap handling.

This section provides suggestions and examples for using the C compiler in embedded applications.

7.10.1 Volatile and Const Type Qualifiers

The **`const`** and **`volatile`** type qualifiers can be used in embedded applications to indicate ROM entities and memory mapped entities, respectively. A general overview of the semantics and use of these qualifiers is explained below. For further details see the ANSI C standard.

const

The value of an object (any value expression) whose type includes the **`const`** qualifier cannot be modified. The **`const`** qualifier is mainly used to make constant strings and variables a part of the program code and place them into ROM.

A non-volatile global or static object declared as **`const`**, is allocated in read-only memory (the **`.rdata`** sections) if it is initialized.

Example

```
const int i = 137;          /* i is defined as const */
i = 17;                     /* this is illegal !!  */
i += 12;                    /* this is illegal !!  */
```

The **const** syntax allows for the declaration of both *constant pointers* and *pointers to constants*.

Example

```
const char * pcc            /* pcc is defined as pointer to*/
                            /* const char*/
char * const cpc;           /* cpc is defined as const pointer*/
                            /* to char*/
const char * const cpcc;    /* cpcc is defined as const*/
                            /* pointer to const char*/
```

The types *pointer to const object* and *const pointer to object*, as in the above example, have different meanings. The value of a *pointer to const object* can be modified; however the value of the pointed object can not be modified. In contrast, the value of a *const pointer to object* can *not* be modified; however the value of the pointed object can be modified.

Example

```
const char * pcc;           /* pcc is defined as pointer to */
                            /* const char         */

pcc ++;                     /* this is O.K.   */
*pcc = 17;                  /* this is an error */
```

Volatile

The value of an object (any value expression) whose type includes the **volatile** qualifier can be used or changed by asynchronous events (such as I/O or interrupts). Such an object should not be subject to any optimization that changes or delays references to it.

By using the **volatile** qualifier, you can specify volatile objects. Therefore, full optimization is carried out on all other objects, including global variables and pointer dereferences.

Example

In the following code:

```
volatile int i;
int j;
. . .
. . .
foo() {
      . . .
      . . .
      for (i=1 ; i<j; i++) {
      . . .
      . . .
      }
}
```

The compiler can put **j** in a register. This optimization is not permitted for **i**.

The **volatile** syntax allows for the declaration of both *volatile pointers* and *pointers to volatiles*.

Example

```
     char * pc;                 /* pc is defined as pointer to char
*/
     volatile char * pvc;       /* pvc is defined as pointer to   */
                                /* volatile char          */
     char * volatile vpc;       /* vpc is defined volatile pointer
*/
                                /* to char             */
     volatile char * volatile vpvc;
                                /* vpvc is defined as volatile
*/
                                /* pointer to volatile char   */
```

The types *pointer to volatile object* and *volatile pointer to object*, as in the above example, have different meanings. References to a *pointer to volatile object* can be optimized; however references to the pointed object can *not* be optimized. In contrast, references to a *volatile pointer to an object* can *not* be optimized; however references to the pointed object can be optimized.

7.10.2 Memory Allocation

Memory allocation is performed by the operating system in native programming environments. However, embedded applications require the ability to control memory allocation. This is achieved by specifying in the linker directive file:

- the memory ranges of various program sections.
- the division of program sections into ROM and RAM.
- the sections to be copied from ROM to RAM at program start-up.

A complete description of the linker directive file is provided in Chapter 3 of the *CompactRISC Toolset - Object Tools Reference Manual*. An example of a simple linker definition file for defining two areas of memory is shown below:

```
MEMORY {
     ROM : origin=0x1000  length=0x2000
     RAM : origin=0x10000 length=0x80000
}
```

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

7.10.3 Initialized C Variables

The C programming language allows compile-time initialization of global and static variables. In addition, uninitialized global and static variable are defined by the C language to have a zero value at program start-up.

In native environment, initialization is handled by the compiler and the operating system. On the other hand, in the CompactRISC development environment, these initializations are performed either by the debugger, (when loading the program), or by the program itself, (when executing, in which case the initialized data is copied from the ROM to the RAM). By default, the second option is used, because this is required in production mode.

The CompactRISC Linker directive file and the CompactRISC run-time library are used to automatically initialize RAM variables.

Refer to the *CompactRISC Toolset - Object Tools Reference Manual* for further details.

7.10.4 Programming Memory Mapped Devices

When writing code for the registers of memory mapped peripherals, correct and efficient access to these entities can be problematic. However, the CompactRISC C Compiler allows optimization of such code.

The **`volatile`** qualifier should be used to specify the memory mapped entities. This allows the optimizer to perform optimizations without changing or delaying references to these entities.

Example The correct way to code memory mapped entities is:

```
#define CTRL_REG *((volatile short *)0xff01)
if (CTRL_REG & 0x0f)
      j++;
```

results in:

```
#---if (*((volatile short *) 0xff01) & 0x0f)
      loadw     65281,r0
      andb      $15,r0
      cmpb      $0,r0
```

```
        beq       .L2
    #--j++;
        loadw     _j,r0
        addw      $1,r0
        storw     r0,_j
    .L2:
```

Note Do not define a global pointer variable, such as **`volatile short *ctrl_reg = (volatile short *) 0xff01`** for memory mapped entities. Dereferencing such a pointer, as in **`*ctrl_reg`**, results in less efficient code.

7.10.5 Programming Trap/Interrupt Routines

The example used in this section is a clock display for the time of day.

The routine **`clock_handler()`** handles a clock interrupt, which occurs **`TICKS_PER_SECOND`** times per second. The time display is updated every second.

A trap/interrupt routine saves all the registers it uses. In addition, scratch registers are also saved, if the routine calls another routine (as in the following example where **`update_time_display()`** is called).

The C code for the clock interrupt handler is:

```
#pragma interrupt(clock_int_routine)
void clock_int_routine(void){
     static int counter;
     static int hours;
     static int minutes;
     static int seconds;

     counter++;
     if (counter == TICKS_PER_SECOND){
          seconds++;
          counter = 0;
          if (seconds == 60){
               minutes++;
               seconds = 0;
               if (minutes == 60){
                    hours++;
                    minutes = 0;
                    if (hours == 24)
                    hours = 0;
                    }
               }
          update_time_display(hours,minutes,seconds);
     }
}
```

The CompactRISC microprocessor family operates in direct exception mode. In this mode, the address of the interrupt handler (residing in the interrupt dispatch table) is interpreted by the CPU as a pointer. The clock interrupt entry in the interrupt dispatch table should be set to the address of **clock_int_routine**. The following line is inserted to the clock interrupt entry in the initialization of the interrupt dispatch table.

```
void (*const _dispatch_table[])() = {
...
_clock_int_routine
...
};
```

7.10.6 Semaphores

Many applications require semaphores. Semaphores protect the entry to critical code segments, and protect the consistency of data (e.g., buffer counter). In the CompactRISC family, the only way to protect such code and data is to use the methodology described below.

The aim is to make critical sections exclusive to one task at a time. In the CompactRISC family this is achieved by making the sections uninterruptable, using a special, efficient, CompactRISC mechanism.

The **PSR** register contains two bits that control interrupt enabling, the **I** bit and the **E** bit. Maskable interrupts are enabled only when these two bits are set.

The **I** bit is the global interrupt enable bit. Upon reset it is cleared by the CPU, and therefore maskable interrupts are disabled. An application program typically sets this bit some time after reset in order to enable interrupts.

The **E** bit is the local interrupt enable bit. Upon reset, it is set by the CPU. The CompactRISC CPU has two dedicated instructions, **di** and **ei**, that clear and set the E bit, respectively.

To protect a code section, call the **_di_()** macro, which clears the **E** bit. When this section is exited, call the **_ei_()** macro, which sets the **E** bit. This process prevents an interrupt from occurring during the critical code section, without changing the global interrupts status as reflected in the **I** bit.

Example Process A:

```
if (buffer_count < BUF_LIMIT) {
      /* puts a new element in the buffer */
      _di_();
```

```
        buffer_counter++;
        _ei_();
    }
```

Process B:

```
    if (buffer_count > 0) {
        /* gets an element from the buffer */
        _di_();
        buffer_counter--;
        _ei_();
    }
```

Processes A and B can now interleave with no consistency problem.

CR16B The CR16B architecture has bit manipulation instructions which are not interruptable, thus if the above semaphore operations involve modifying one bit only, the **_di_()** and **_ei_()** macros are not required.

7.10.7 Alignment

The CompactRISC family allows non-aligned-to-size memory references, although such accesses incur a penalty in memory access time.

Usually, the compiler aligns the memory address of all memory objects. There are, however, a few cases (e.g., structures and cast operations) where this is not so.

Structures alignment Structures elements alignment is controlled by the **-Jalign_size (align_size =1, 2, 4)** option. A smaller value, results in a smaller structure size. The penalty is reduced performance.

Cast operations When casting variables to different types, a variable may be referenced as a type of larger size. This may cause slower program execution because the smaller type object memory address will probably not be aligned, as required by the larger memory reference, thus requiring greater memory access time.

Example

```
int foo(char *s){
    /*probably inefficient memory access*/
    return *(int *) s;
}
```

7.11 LINKER INPUT SECTIONS GENERATED BY THE COMPILER

There are four groups of input sections generated by the compiler:

.text	- program code
.rdata	- read-only data
.data	- initialized data
.bss	- uninitialized data

The data (**.data**, **.rdata** and **.bss**)sections are subdivided (according to the alignment of the variables that reside in them, i.e., the actual names of the sections have underscores ('_') followed by the alignment (1, 2 or 4) appended to them. For example, section **.rdata_1** includes read-only variables whose alignment is 1, whereas **.bss_4** includes uninitialized variables whose alignment is 4.

Chapter 8
IMPLEMENTATION-DEFINED BEHAVIOR

Annex G.3 of the ANSI C standard lists all the issues that are implementation defined. This Appendix defines those issues given by the CompactRISC compiler.

8.1 TRANSLATION

When a diagnostic is detected, a message is given in the following format:

```
<file name>:<line number>:<diagnostic description>
```

8.2 ENVIRONMENT

- No arguments can be passed to the function **`main()`**.
- An interactive device is the terminal from which the debugger was invoked.

8.3 IDENTIFIERS

- All characters in an identifier without external linkage are significant.
- The first 999 characters, in an identifier with external linkage, are significant.
- Case distinctions are significant in identifiers with external linkage.

8.4 CHARACTERS

- The source character set also includes the dollar sign ($).
  The execution character set also includes the dollar sign ($), the at sign (@) and the backquote (`).
- The number of bits in a character in the execution character set is eight.

- Each character in the source character set is mapped to the identical character in the execution character set.
- The value of a character that is not part of the execution character set but is a member of the ASCII character set is its ASCII value. Characters that are not members of the ASCII character set are illegal. The value of an escape sequence that is not represented in the execution character set is the value of the octal or hexadecimal constant following the escape sequence.
- The value of an integer character constant that contains more than one character depends on the target architecture. If the target architecture is 16-bit CompactRISC, the value is always the second character. If the target architecture is 32-bit CompactRISC, the value is the fourth character or the last character if there are less than four characters.

  The value of an integer character constant, that contains more than one-wide character, is always the first wide character.
- The type **`char`** has the same range of values as **`signed char`**.

8.5 INTEGERS

- The representation of positive integers is the same number in base 2. The representation of negative integers is the 2's complement of its absolute value.

  The following table details the minimum and maximum values of the various integer types:

Type	Min	Max
signed char	−128	127
short int	−32768	32767
long int	−2147483648	2147483647
unsigned char	0	255
unsigned short int	0	65535
unsigned long int	0	4294967295

  The set of values for **`int`** is the same as **`short int`** when compiling for the CR16 architecture, and is the same as **`long int`** when compiling for the CR32A architecture.

- The set of values for **unsigned int** is the same as **unsigned short int** when compiling for the CR16 architecture, and is the same as **unsigned long int** when compiling for the CR32A architecture.
- The result of converting an integer to a shorter signed integer is the low-order bytes of the integer. The number of bytes taken is the same as the size of the smaller signed integer.
- The result of converting an unsigned integer to a signed integer of the same length is the 2's complement interpretation of the unsigned integer.
- The result of a bitwise operation in signed integers is the 2's complement interpretation of the result of the bitwise operation.
- The sign of the remainder on integer division is the same as that of the dividend.
- The semantics of a right shift of a negative-valued signed integral type is the same as that of a non-negative-value.

8.6 FLOATING POINT

- The size of type **float** is 32 bits. The size of types **double** and **long double** are 32 bits for the CR16 architecture and 64 bits for the CR32A architecture. Their representation and range of values are compatible with the IEEE standard for binary floating-point arithmetic.
- The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value is towards the nearest value. If the original number is halfway between two values, the even value (least significant bit = 0) is returned.
- The direction of rounding when a floating-point number is converted to a narrower floating-point number is to the nearest value. If the original number is halfway between two values, the even value (least significant bit = 0) is returned.

8.7 ARRAYS AND POINTERS

- The type **size_t** is equivalent to type unsigned **int**.
- The size of an integer needed to cast a pointer to an integral value, or vice versa, is the size of type **int**. In the CR16 compiler there are far pointers which need to be cast to **long int**. The same is true for function pointers in CR16B large model.

- The result of casting an integral value to a pointer is the address represented by the integral value. There is one exception to this rule: when an integral value is cast to a function pointer in the CR16 compiler, it is first shifted by 1 to the left. The result of casting a pointer to an integral type is the 2's complement interpretation of the address.
- The type **ptrdiff_t** is equivalent to type **int**.
- In CR16A/CR16B a function pointer holds the actual function address shifted by one to the right. The least significant bit of the address, which is implied as zero, is excluded from the address representation. Refer also to the *CompactRISC - CR16A Programmer's Reference Manual* or *CompactRISC - CR16B Programmer's Reference Manual* for details.

8.8 REGISTERS

- When compiling without the optimizer, the compiler will assign a register to variables declared with 'register' storage, if they fit into a register or a pair of registers, and if there are enough registers available. When compiling with the optimizer, the compiler treats any 'register' declaration as an 'auto' declaration.

8.9 STRUCTURES, UNIONS, ENUMERATIONS, AND BIT-FIELDS

- If a member of a union object is accessed using a member of a defined type, the result is unpredictable.
- When compiling with the **-J1** option, there is no padding between structure members. When compiling with the **-J2** option (default for CR16 architecture) anything larger than a byte is word-aligned. When compiling with the **-J4** option (only relevant for CR32A architecture for which it is the default) anything larger than a word is double-word-aligned and words are word-aligned.
- A "plain" **int** bit-field is treated as a **signed int** bitfield.
- The order of allocation of bitfield within a unit is low-order to high-order.
- In the CR16 architecture a bitfield cannot straddle a word boundary.
- In the CR32A architecture a bitfield cannot straddle a double-word boundary.

- The integer type chosen to represent the values of an enumeration type is **`int`**.

8.10 QUALIFIERS

- An access to an object that has volatile qualified type, is either a read or a write to that object.

8.11 DECLARATORS

- The maximum number of declarations that may modify an arithmetic, structure, or union type is 4500.

8.12 STATEMENTS

- There is no limit on the number of 'case' values in a 'switch' statement.

8.13 PREPROCESSING DIRECTIVES

- The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Single-character constant may not have a negative value.
- The compiler looks for includable source files in the following places:
  - in all the directories given with the **`-I`** option
  - in ***`CRDIR`***`/include` (where ***`CRDIR`*** is the directory in which the CompactRISC toolset is installed.
- When the included file is a quoted name, it searches for it in the current directory. If that fails then it will search for it in the same manner as that of regular includable source file.
- The mapping of source file character sequences to specific files is according to the operating system under which the compiler is being run.

- For the behavior of various recognized **#pragma** directives, see Chapter 4.
- The definitions of **__DATE__** and **__TIME__** are available.

8.14 LIBRARY FUNCTIONS

- The macro **NULL** expands to **(void *)0**.

Appendix A
COMPILER LIMITATIONS

A.1 INCLUDING EXECUTABLE C LINES

We do not recommend including a C file (or any file with executable lines) in a C file, although it is possible to do so. This is due to a limitation in COFF (Common Object File Format) files, which do not keep the source-line number for an included file. As a result it is impossible to perform source-level debugging of executable lines which belong to an included file. These lines are, however, executed correctly.

A.2 LARGE ARRAYS (CR16 ONLY)

The CR16 compiler does not support arrays of size greater than 65535 bytes. This is because array indexing is carried out in 16-bit (integer size) arithmetic.

A.3 NEGATIVE ARRAY INDEX (CR16 ONLY)

The CR16 compiler does not support array reference with a negative, non-constant index.

Example

```
int i = -1;
extern int arr[];

foo()
{
     arr[i] = 1;
}
```

A.4 SWITCH STATEMENT CODE SIZE

The size of the code that is generated by the compiler for a single switch statement is limited to 32K (32768) bytes. This is a reasonable limitation that allows the compiler to generate compact jump tables in some cases. Every entry in the jump table contains the offset between the beginning of the switch statement and the code that is generated for a specific case. The offset is a 16-bit signed integer, and hence the limitation.

A.5 JUMP TABLES MUST RESIDE UNDER 64K (CR16 ONLY)

When optimizing switch statements into jump tables, the data is directed into a `.rdata` section. Make sure that the `.rdata` section is located below 64K.

A.6 RECURSIVE CPP MACROS

The compiler ignores a **cpp** macro which calls itself recursively, and does not issue an error or warning message.

Example

```
#define mac(x) mac(x)
```

A.7 DOUBLE PRECISION FLOATING POINT VARIABLES (CR16 ONLY)

The CR16 compiler does not support double-precision floating-point variables i.e., C variables of type **double**. For compatibility the **double** keyword is recognized, but it is equivalent to **float**. In other words, in the CR16 compiler, variables of type **double** are single-precision variables.

Appendix B
COMPATIBILITIES WITH GNX C COMPILER

B.1 DOLLAR SIGN IN IDENTIFIER NAMES

In ANSI C, dollar signs are not allowed in identifier names. The CompactRISC C Compiler, like other traditional compilers, allows such identifiers. Identifier names that contain dollar signs are also valid when running in compatible mode.

B.2 BITFIELDS

In ANSI C only three types of bit field are allowed: **`int`**, **`signed int`** and **`unsigned int`**.

The CompactRISC C Compiler contains no such restriction. Running the compiler in compatible mode also allows the traditional bitfields.

B.3 VARIABLE AND STRUCTURE ALIGNMENT

The ***`-Jwidth`*** option for alignment within structures is supported. The ***`-Jwidth`*** option allows you to set structure-member alignment on bytes (***`width`***=1), words (***`width`***=2), (or double-words (***`width`***=4) for CR32).

B.4 COMPATIBILITY WITH GNX 32000 INTRINSIC FUNCTIONS

The CompactRISC tools enable you to compile and run code that uses intrinsic 32000 commands. CompactRISC provides support via macros and functions. Note that the use of these functions is not recommended, as they have no advantage in performance over regular user code or functions.

These macros and functions are located in the CompactRISC **`libc`** library, and the **`ns32000.h`** header file found in the **`include`** directory under the development tools root directory.

The header file **`ns32000.h`** contains the prototypes of all the 32000-compatible functions, and all the 32000-compatible macro definitions. The actual functions are located in the **`libc`** library.

To use a 32000-compatible function, you must include the header file, **ns32000.h**, prior to any call to the function, and link with the **libc** library. Note that for 32000 intrinsic functions implemented as macros (e.g., **_spr**) run-time checks for parameters are not carried out.

The following functions and macros are supported:

Functions	Macros
_extd	_lpr
_extw	_spr
_extb	_bpt
_ffsb	_flag
_ffsw	_cvtp
_ffsd	_bicpsrw
_ins	_bispsrw
_movsb	
_movsw	
_movsd	
_movsb_b	
_movsw_b	
_movsd_b	
_rotb	
_rotw	
_rotd	
_svc	
_cbit	
_ibit	
_sbit	
_tbit	

For more detailed information about these instructions, see the *Series 32000 Programming Reference Manual.*

B.4.1 Incompatibilities

The following 32000 intrinsic macros can not be implemented on the CR32 due to hardware limitations:

_cbiti - set bit interlocked
_sbiti - clear bit interlocked
_movst - move string and translate.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z