CompactRISC™

Introduction

Part Number: 424521772-001

August 1998

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
0.6	July 1995	First beta release.
0.7	January 1996	Minor corrections and additions.
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.
1.2 & 2.0	January 1998	Product release for version 1.2. Beta release for version 2.0.
2.1	August 1998	Product release.

PREFACE

Welcome to the CompactRISC Development Toolset for the CompactRISC microprocessor family. This manual presents a first view of the CompactRISC Toolset, and shows you how to make the best use of the tools.

This manual includes detailed descriptions of all the manuals in the toolset, and illustrates the use of the toolset with several examples for each of the tools.

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

CONTENTS

FIGURES

Chapter 1
OVERVIEW

1.1 INTRODUCTION

Welcome to the CompactRISC software cross-development toolset for National Semiconductor's CompactRISC embedded-microprocessor family.

CompactRISC is a family of RISC microprocessors targeted for embedded systems, and covering a wide range of applications.

The CompactRISC Toolset supports software cross-development. Software is developed on a host computer, while the target is an embedded CompactRISC-based system. The toolset is designed to take full advantage of the CompactRISC architecture. It supports the complete range of CompactRISC-based microprocessors using a common user-interface.

1.2 MANUAL ORGANIZATION

This manual presents a first view of the CompactRISC Toolset, and shows you how to make the best use of the tools. It contains the following Chapters and Appendices:

Chapter 1 *Overview*, introduces the tools, and describes the structure of the documentation.

Chapter 2 *Using the Toolset*, provides a detailed tour through the CompactRISC development tools. It uses a "getting-started" example to introduce each tool together with its most important options.

Chapter 5 *Embedded Programming Topics,* deals with various aspects of embedded application programming and the way they are handled by the CompactRISC Toolset.

1.3 THE COMPACTRISC MICROPROCESSOR FAMILY

The CompactRISC core is the first CPU core, specifically designed for the embedded microprocessor market, to use RISC technology. CompactRISC CPU cores can range from 8-bit to 64-bit. The current CompactRISC implementation consists of 16 and 32-bit CPU cores. All CompactRISC-based CPU cores have very similar instruction sets, register sets, addressing modes, interrupt and trap handling and debug support. This common programming model ensures efficient high-level language execution, and highly portable code.

The CompactRISC CPU cores are designed for minimum die size, small code size, high performance, reduced power consumption, and flexible design methodology. These features make the CompactRISC competitive in a wide range of applications, including cost-sensitive and power-sensitive applications.

The current implementations of CompactRISC-based CPU cores include the following:

- CR16A - the first CompactRISC-based 16-bit core.
- CR16B - the second generation enhanced 16-bit core.
- CR32A - the first CompactRISC-based 32-bit core.

We recommend that you familiarize yourself with the CompactRISC architecture (i.e., its programming model, register set, instruction set etc.), especially if you are going to program in assembly language. Refer to the appropriate Chip Reference Manual.

1.4 THE COMPACTRISC DEVELOPMENT TOOLSET

The CompactRISC tools are cross-development tools. You develop a program, compile, link and optionally simulate it, on a host computer. The program's machine code and data is then downloaded to a CompactRISC microprocessor-based target system for execution and debugging, using a debugger which runs on the host. Once it is verified it can be programmed in ROM and executed (stand-alone) on the target system.

The CompactRISC Development Toolset includes the following components:

- Optimizing ANSI C Compiler
- C run-time libraries
- Assembler with Macro Pre-processor
- Linker
- Archiver
- Miscellaneous Object File utilities
- Performance and Functional Simulators
- Firmware Run-time environment (Monitor)
- Debugger Communication Layer (DBGCOM)
- Graphical Source Debugger

The CompactRISC Toolset is currently available for IBM PCs, and compatibles, using Windows 95 and Windows NT.

The following table shows the minimal PC configuration requirements, for running the CompactRISC Toolset:

Table 1-1. CompactRISC Toolset System Requirements

Processor	i486 or later
OS	MS Windows 95® MS Windows NT ®
Memory	8 Mbytes
Communication hardware (required only when connecting to a development board)	RS-232 port or RS-422 add-in card, or JTAG add-in card

1.5 MAIN FEATURES OF THE TOOLSET

1.5.1 The CompactRISC C Compiler

The toolset includes a full ANSI C compiler and optimizer. It generates high-quality code for the CompactRISC architecture. The CompactRISC C Compiler 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.

Using the advanced features of this compiler you can develop your application entirely in the C language, thereby reducing development time and improving the portability of your code. The C compiler's main features are:

- Full ANSI C
- GNU C compatible
- Allows programming of interrupt/trap handlers in C
- Fully supports the underlying CompactRISC architecture
- Optimizations can be tuned either to improve speed or save space
- User-controlled optimization level
- Global variables may be placed in registers for the entire program
- User-controlled alignment of variables and structure members
- Assembly output can be annotated with source lines
- Fast compilation mode
- Enables errors dump to file.
- Supports code Bank Switching (CR16A-based chips with add-on bank-switching support only).

The C compiler performs global optimizations by looking at the code of a whole procedure at a time, not only in the local context of a line or loop. A wide range of optimization options allows you to finely control the performance and memory allocation of your program.

The C compiler is aware of the hardware support available on each CompactRISC-based microprocessor, and includes support and extensions for embedded programming and application-specific instructions. The compiler uses its knowledge of each specific implementation of the CompactRISC instructions to generate the most efficient code sequence. Thus, you can write even your most time-critical applications in C.

1.5.2 The CompactRISC Assembler

The toolset enables you to develop your application entirely in the C language. You can, however, develop your application, or parts of it, in assembly language using the CompactRISC Assembler.

The assembler assembles CompactRISC assembly language source programs and generates relocatable object files. The CompactRISC Assembler supports the Common Object File Format (COFF). The main features of the assembler are:

- Powerful macro capability
- Supports the whole range of CompactRISC architectures
- Generates relocatable object files
- Optionally produces assembly program listing
- Displacement optimization reduces code-size and increases performance
- Enables errors dump to file.

1.5.3 The CompactRISC Linker

A linker is a very important tool, especially for development of embedded applications in which the developer requires full control over memory allocation. The CompactRISC Linker links one or more object files and libraries and creates one executable object file. In addition it supports the needs of embedded applications and provides useful features for developers of such applications.

The linker's main functions and features are:

- Resolves external symbolic references
- Relocates code and data
- Gives you full control over memory allocation and program layout using linker directive language.
- Automatic data initialization support for embedded application software
- Produces a detailed memory map
- Supports partial (incremental) linking
- Enables errors dump to file.
- Supports code and data overlays
- Supports code Bank Switching (CR16A-based chips with Bank Switching hardware support)

The CompactRISC Linker supports the Common Object File Format (COFF).

1.5.4 The CompactRISC Object Utilities

The toolset includes several utilities which handle object files:

The CompactRISC Archiver: crlib. The archiver is used to combine groups of object files into a single software library, or archive file.

The CompactRISC ROM Programing Utility: crprom. Translates an executable file into one of several formats used by PROM programmers. The whole process of creating an executable file and programmable parts of it on PROMs is made simple by specifying both the RAM and ROM configuration in the Linker Definition File. The complete memory map is then automatically passed on to the programming interface utility.

The CompactRISC Object File Viewer Utility: crview. Displays information about all parts of a COFF object file including object file headers, section headers, raw data (i.e., sections contents like code, initialized data), symbol table, line number information and relocation information. All the information is displayed in a formatted and easy to read manner. The symbol table can be displayed in different formats, and can be filtered and sorted. In addition, there is an option to display section size information in a very convenient format to evaluate program memory consumption.

1.5.5 The CompactRISC Run-time Libraries

The toolset contains a set of run-time libraries. You can link your program with any of these libraries according to the needs of your application. The libraries are:

libc

The CompactRISC standard ANSI-C library is based on the Dinkum C library by Dinkumware Ltd. This library contains all the standard C functions defined by ANSI. It includes standard I/O functions, string processing functions, dynamic memory allocation functions, mathematical functions and other ANSI functions. In addition, this library includes division and modolu emulation functions, emulation functions for National Semiconductor's Series 32000 embedded microprocessor instructions (for backward compatibility), and 32-bit emulation functions for users of the CompactRISC 16 processors.

In addition, **`libc`** includes low-level I/O functions that implement "virtual I/O". The virtual I/O feature of the CompactRISC toolset enables programs that are executed by the debugger, on a development board, or in simulation mode, to read data from, and write data to, the host computer. Using virtual I/O, these programs can read from the standard input, or from any file on the host, write to the standard output, or to any file on the host, and manipulate files on the host (e.g., remove or rename them). Virtual I/O is implemented by a special protocol between the application program and the CompactRISC Debugger. The low-level I/O functions implement this protocol from the application program side. These functions are used by high-level I/O functions like the ANSI-C standard I/O functions, but they can also be called directly by the application program.

libstart The default start-up library contains several run-time initialization services that are required to run user applications on either an Application Development Board (ADB) or a simulator. These services include the default initialization routines that are essential for proper execution (e.g., stack initialization), and the default interrupt dispatch table, that is essential for debugging the application program. You should use this library to communicate with the new TMON (i.e., TMON ver 2.1.0 or higher), or when developing in simulation mode. This is the default start-up library.

libadb The start-up library which is compatible with old versions of TMON (i.e., TMON ver 2.0.x or lower). This library supplies the same functionality as the new **`libstart`**, but is used to communicate with the old TMON. This is the default start-up library for version 1.2, or lower.

libhfp The Floating-Point Emulation library contains functions which emulate floating-point operation, since the CompactRISC CPU does not support these operations at machine-level. The CompactRISC C compiler automatically generates a call to one of these functions, whenever it detects a floating-point operation in the source code.

libd The Dummy Floating-Point Emulation library has the same interface as the Floating-Point emulation library, however, all its functions are empty. Use **`libd`** instead of **`libhfp`** when the application program does not perform floating-point operations.

For more information about the CompactRISC Libraries refer to the *CompactRISC Toolset - Compiler Reference Manual*.

1.5.6 The CompactRISC Debugger

The CompactRISC Debugger is a graphical C source-level debugger with special support for the various CompactRISC run-time environments. Its main features are:

- Supports C or assembly programs
- Initialization files for automatic initialization and restart
- On-line Help
- User configurable buttons and keys
- Execution of commands from a preconstructed command (input) file
- Recording your debugging session, for later analysis and/or re-execution. Facilitates automation of regression test suites
- Function call stack display in a window
- Traversal of data structures (lists, etc.) through pointers using mouse click
- **ALIAS** and **SET** commands, for customizing the command interface
- Recall, editing, and re-execution of previous commands
- Multiple commands on a single line
- Mixed-mode debugging (both C source and assembly code are displayed in the source window at the same time)
- Save and restore debug environment
- Call user subroutines/functions from command level
- Access to host file system using Virtual I/O

1.6 COMPACTRISC DOCUMENTATION OVERVIEW

The CompactRISC documentation set describes and specifies the CompactRISC cross development tools. It consists of the following manuals:

The CompactRISC Toolset - Introduction (This manual)

Provides an overview of the CompactRISC Development Tools.

The CompactRISC Toolset - C Compiler Reference Manual

This manual provides guidelines for using the compiler and information regarding the compilation process.

In addition, you will find an overview of the CompactRISC support libraries, including a complete C run-time library, system calls and emulation functions.

The CompactRISC Toolset - Assembler Reference Manual

This manual describes the CompactRISC Assembler and assembly language in detail.

The CompactRISC Toolset - Object Tools Reference Manual

This manual describes the various tools that are used to maintain and combine object and executable (COFF) files. This manual includes the following parts:

- CompactRISC Linker
- CompactRISC Archiver
- The CompactRISC ROM Programing Utility
- The CompactRISC Crview Utility

Additional utilities, for viewing and controlling object files, are also described in this manual.

The CompactRISC Toolset - Debugger Reference Manual

This manual describes the CompactRISC graphical source-level debugger, its user interface and its method of operation. It also describes the CompactRISC execution and debugging environment, including the Debugger Communication Layer (DBGCOM) software, add-in communication board (e.g., NSV-RS422-COM), function and performance simulators, and the Application Development Board (ADB).

Chapter 2
USING THE TOOLSET

2.1 INSTALLING THE TOOLSET

The release letter, which you received with the CompactRISC Toolset, contains detailed installation instructions. Read the instructions carefully before you install the toolset.

2.2 DOCUMENTATION CONVENTIONS

The documentation conventions used in the CompactRISC Toolset manuals are shown below. The commands are case-sensitive, and although shown in uppercase, they must be input in lowercase.

The commands you enter are shown in **`boldface-courier`**. Variables, pathnames, and filenames are in *`italic-courier`*.

Syntax notation

Spaces or blanks, when present, are significant; they must be entered as shown. Multiple blanks may be used in place of a single blank.

{ }	Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by a logical OR sign |.
[ ]	Large brackets enclose optional item(s).
|	Logical OR sign separates options, only one of which can be used.
...	Three consecutive periods indicate optional repetition of the preceding item(s). If a group of items can be repeated, the group is enclosed in large parentheses ( ).
( )	Large parentheses enclose items which need to be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated. The parentheses indicate that the group may be repeated.

All other characters or symbols appearing in the syntax must be entered as shown. Brackets, parentheses, or braces which must be entered are shown smaller than the symbols used to enclose optional items. (Compare user-entered **[ ]**, with **[ ]**, which encloses optional items.)

Conventions for examples

In examples user input is preceded by the **>** sign. System responses are indented. For example:

To display the current selection:

```
> target
     Current_target = SIMULATOR
```

Special keys to be entered are specified in all capital letters (**ESC**, **TAB**, **RETURN**, etc.). Mouse operations are specified as they should be carried out on the host system.

Syntax for examples

This syntax shows how to enter a correct response to a command, or engage in an interactive session with the system.

- All user-input lines are terminated by RETURN. RETURN is not indicated, unless it is the only user input (indicating the default for that input line).

2.3 GETTING STARTED

To demonstrate a development cycle using the CompactRISC Development Toolset, we take a simple example that includes:

- Writing a C program
- Compiling the program
- Running the program

Consider the following program, often used by C beginners:

```
#include <stdio.h>
main()
{
   printf("Hello world !\n");
}
```

Step 1

We use the CompactRISC C Compiler, **crcc**, to compile and link this program.

```
> crcc -g hello.c -o hello.x
```

This command compiles and links the C program, **hello.c**, and generates a CompactRISC executable object file. This file is named **hello.x**, as specified by the **-o** option. The -g option tells the compiler to generate symbolic information, which is useful if the program is to be debugged with the CompactRISC Debugger.

Step 2 We now run this program on a CompactRISC target. At this stage, we do not deal with running the program on a development board. Instead we use the instruction-level simulator which runs on the host platform. By default, the debugger executes programs in simulation mode. To this end, we invoke the debugger by clicking the CRDB icon.

Step 3 We now load the executable object file **`hello`**, using the **Load** option from the **File** menu. The program is now ready to run. Press the **Go** button to run the program.

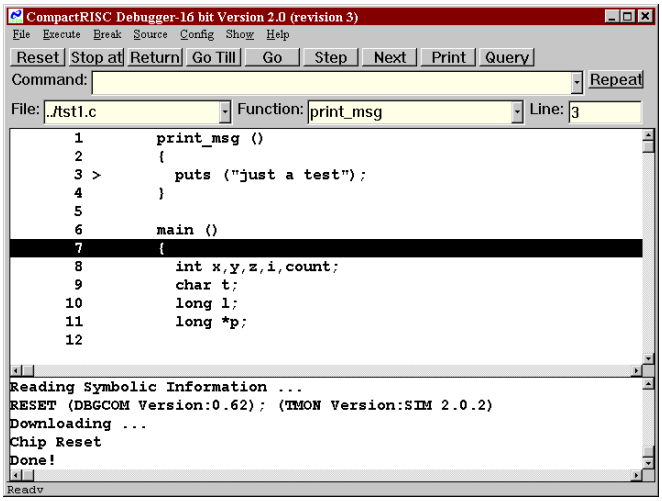


Figure 2-1. The Debugger Window

As you can see, the program has been run and then exited. Note that the output appears in the debugger output window.

2.4 USING THE C COMPILER

You invoke the compiler with the **`crcc`** command. However, **`crcc`** does more than just compile your code. If you invoke **`crcc`** without specifying any option:

```
> crcc prog.c
```

`crcc` carries out the following:

- It invokes the C preprocessor (**cpp**).
  This expands macros (defined by **#define**), includes header files (**#include**), and processes conditional code (e.g., code enclosed by an **#ifdef** statement)

- It invokes the compiler engine (**cc1**).
  This translates the preprocessed C program to a CompactRISC assembly language program.

- It invokes the CompactRISC assembler (**crasm**).
  This converts the assembly program to an object file. The object file contains CompactRISC machine code, program data and in addition symbolic information, relocation information and additional useful information.

- It invokes the CompactRISC linker (**crlink**).
  This links the object file with default run-time libraries in order to generate executable object file. The default executable object file name is **cr.x**.

In addition, **crcc** has useful options which enable you to perform part, or parts, of the above steps, as required.

-c option

Use the **-c** option to generate an object file without linking it. This is useful when you want to invoke the CompactRISC Linker directly, and not via **crcc**. With this option, **crcc** generates an object file. The default object name is identical to the C file name except that the suffix is replaced by **.o**. For example:

```
> crcc -c prog1.c
> crcc -c prog2.c
> crlink prog1.o prog2.o -o prog -ladb -lc
```

-S option

Use the **-S** option to inspect the assembly language program that was generated from your C program. In this case, **crcc** does not invoke the assembler and leaves you with the assembly code. The default output file name suffix is **.s**. For example:

```
> crcc -S prog1.c
```

-n option

Use the **-n** option, in conjunction with the -S option, to get the assembly program generated by the compiler in annotated form. The compiler puts the original C line as a comment in the assembly program just before the assembly code that was generated by this line. This output format is very convenient for analyzing the assembly code that was generated from your C code. For example:

```
> crcc -S -n prog1.c
```

```
#--- foo()
#--- {
#--- int i;
     .globl _a
```

```
     .globl _b
     .text
     .align 2
     .globl _foo
_foo:
#---   for(i = 0; i < 10; i++)
     movw  $0,r3
     movw  $_a:m,r2
     movw  $_b:m,r1
.L5:
#---  a[i] = b[i];
     loadw  0(r1),r0
     storw  r0,0(r2)
     addw  $2,r2
     addw  $2,r1
     addw  $1,r3
     cmpw  $9,r3
     bge   .L5
#--- }
     jump  ra
```

-P option Use the **-P** option to inspect your C code after it has been preprocessed by the C preprocessor (**cpp**) e.g., when you wish to see how your macros were expanded, or which parts of the code are included. In this case **crcc** just invokes **cpp**, and the default output file name has the suffix **.i**. For example:

```
> crcc -P prog1.c
```

-Q option Use the **-Q** option to check whether your program can be compiled without any errors. This mode of compilation is very quick, and does not generate any output.

-g option Use the **-g** option when you want to debug your program. This option is normally used during the whole development phase of the application. When this option is specified the compiler generates symbolic information which is later on used by the CompactRISC Debugger to perform source level debugging. Specifying this option in the compilation of your program is therefore essential for debugging it.

-O option Use the **-O** option to generate optimal code in terms of speed and/or space. In this case the compiler employs several optimization techniques which are not used by default. Note: if your program is compiled with the **-O** option it is still possible to debug it using the CompactRISC Debugger though certain optimizations may somewhat complicate the debugging.

-Os option If the compiler has an optimization trade-off between speed and space, by default it optimizes for speed. However you can override this default by specifying the **-Os** option which implies space is favored over speed in the optimization process.

--z option Use the **-z** option to dump both errors and warnings into an error file. The error file name is ***filename*.err** where ***filename*** is the base name of C source file. Use **-zn*filename*** to override the default error file name.

--mlarge option (CR16B only) Use this option to generate code for the large programming model (CR16B CPU core).

--msmall option (CR16B only) Use this option to generate code for the small programming model (CR16B CPU core).

--mcr16a option Use this option to generate code for the CR16A CPU core.

The main options for **crcc** are listed below.

Main Compiler Invocation Options

Option	Description
-g	Generates symbolic information for the CompactRISC Debugger.
-O	The compiler employs several optimization techniques, which are not used by default, to generate code optimized for speed and/or space.
-Os	Generates code optimized for space only. (By default the compiler optimizes for speed.)
-Q	Checks if your program compiles without errors; no output created.
-S **-n**	Generates the assembly code, but does not invoke the assembler. Allows inspection of the assembly language program, generated from your C program. Annotates this code. Puts the original C line as a comment in the assembly program, just before the assembly code generated by this line
-P	Invokes **cpp** only. Shows how your macros were expanded, or which parts of the code were included.
-c	Invokes the CompactRISC Linker directly, and not via **crcc** to generate an object file, without linking it.
-z[n*filename***]**	Dump errors and warnings into a file.
-mlarge	Generates code for CR16B large model.
-msmall	Generates code for CR16B small model.
-mcr16a	Generates code for CR16A.

For other compiler options refer to the *CompactRISC Toolset - C Compiler Reference Manual*.

2.5 USING THE ASSEMBLER

To assemble your CompactRISC assembly program, invoke **crasm** directly as follows:

```
> crasm prog1.s
```

The assembler generates an object file. The default object file name is the same as the assembly source file except that the suffix is replaced by **.o**. You can override this by specifying the **-o** option with the alternative output object name. For example:

```
> crasm prog1.o -o prog1.obj
```

You can write your whole program in assembly if you wish. In this case you must use the CompactRISC Linker after the assembler to generate an executable object file. For example:

```
> crasm prog1.s
> crlink prog1.o -o prog1
```

or

```
> crasm prog1.s
> crasm prog2.s
> crlink prog1.o prog2.o -o prog
```

Use the **-L** option to generate a program listing in addition to the output object file. The listing is directed to the filename which is specified immediately following the **-L** option. If no file is specified the output is directed to the standard output. For example:

```
> crasm -Lprog1.lis prog1.s
```

```
CompactRISC Assembler (CR16BS) Ver 2.0 (revision 3) 2/1/97 Page: 1

##### File "prog1.s" #####

   1                    .globl _a
   2                    .globl _b
   3                    .text
   4                    .align 2
   5                    .globl foo
   6 T00000000         foo:
   7 T00000000 6038           movw   $0,r3
   8 T00000002 51380000       movw   $_a:m,r2
   9 T00000006 31380000       movw   $_b:m,r1
  10 T0000000a          loop:
  11 T0000000a 02a0           loadw  0(r1),r0
  12 T0000000c 04e0           storw  r0,0(r2)
  13 T0000000e 4220           addw   $2,r2
  14 T00000010 2220           addw   $2,r1
  15 T00000012 6120           addw   $1,r3
  16 T00000014 692e           cmpw   $9,r3
  17 T00000016 be15eaff       bge    .L5
  18 T0000001a dd55           jump   ra
```

The CompactRISC Assembler has a built-in macro processor. Macro processing is performed as the first pass of the assembly process. You may use the **-MO -MP** options of **crasm** if you want it to perform the macro processing phase only, and look at its output. This is useful when you use the macro processing capability of **crasm** and you want to verify that the macros are expanded according to the expected result.

For example:

```
> crasm -MO -MPprog1.m prog1.s
```

The output of macro processing phase is directed to the file **prog1.s**. When no file is specified it is directed to the standard output.

As a brief example of the power of the macro capability of the CompactRISC Assembler we use the macro assembler language to assign values to registers **r0** to **r11**. Register r0 is assigned the value 1, **r1** is assigned the value 2, **r2** is assigned the value 3 and so on. The macro processor can generate the required code with only three lines of macro code.

```
.repeat 12,index
     movw ${index},r{{index}-1}
.endr
```

which is equivalent to

```
movw $1,r0
movw $2,r1
movw $3,r2
movw $4,r3
movw $5,r4
movw $6,r5
movw $7,r6
movw $8,r7
movw $9,r8
movw $10,r9
movw $11,r10
movw $12,r11
```

The assembler can also use the C preprocessor **cpp.** Use the **-c** option to invoke **cpp** as a preprocessor of your assembly program. When **cpp** is used as a preprocessor all traditional **cpp** options (such as **-I -D** and **-U**) are valid as assembler options. For example:

```
crasm -c -Iinclude prog.s
```

The CR16B CPU core has two programming models: the small model and the large model. The main differences between these models, from the point of view of the assembler, are:

- Some of the branch/jump instructions require a register operand in the small model, and a pair of registers operand in large model.
- Some of the instructions are encoded differently in the two models. In other words, the machine code generated for a certain instruction may be different in the two models.

Thus, in the following example, the assembly program is parsed as a CR16B large model program, and the assembler generates code for the CR16B large model:

```
crasm -mlarge prog.s
```

Main Assembler Invocation Options

Option	Description
-L	Generates a program listing, in addition to the output object file. By default, the listing is directed to the standard output.
-MP	Performs the macro processing pass only. Macro processing is performed as the first pass of the assembly process.
-c	Invokes the C preprocessor (cpp) before the assembly process.
-z[n*filename***]**	Dump errors and warnings into a file.
-mlarge	Generates code for CR16B large model.
-msmall	Generates code for CR16B small model.
-mcr16a	Generates code for CR16A.

For other assembler options refer to the *CompactRISC Toolset - Assembler Reference Manual*.

2.6 USING THE LINKER

To link your program, invoke **crlink** and specify the input object file(s) as argument(s). For example:

```
> crlink prog.o
```

The default executable object file name which is generated by the linker is **cr.x**. To override this use the **-o** option with the alternative output file name. For example:

```
> crlink prog.o -o prog
```

-l option

If you wish to add standard libraries to the link process use the **-l** option. For example, if you want to link your program with the CompactRISC standard C library (**libc.a**) specify the **-lc** option as in the following example. The CompactRISC Linker knows where to search for this library. Refer to the *CompactRISC Toolset - C Compiler Reference Manual* for a list of standard libraries and their functions. For example:

```
> crlink prog.o -lc -o prog
```

Always specify libraries in the command line after specifying the input object files. The reason for that is that the linker extracts from object libraries only objects that resolve external symbolic references. By the time it processes a library it should have processed all the input objects that refer to this library. Therefore the library should be specified after the object file in the command line.

-d option

Use the **-d** option to specify a linker directive file. The linker directive file is written in a special linker directive language. When this option is not specified the linker uses a default directive file named **linker.def** which is provided with the CompactRISC Toolset. In most cases, you use your own linker directive file. For example:

```
> crlink prog.o -o prog -d mylinker.def
```

With the linker directive language you can do the following:

- Define the address space of your application i.e. the memory segments which can be used by your application program.
- Define how to combine the input sections from the input object files to output sections of the executable object file which is generated by the linker.
- Allocate memory for the various output section by explicit address assignment or by directing them to one of the memory segments in the application address space.
- Assign addresses for program symbols at link time. These symbols can be referred to by any of the input objects. In other words you can refer from your C or assembly program to symbols which are eventually defined at link time by the linker using the information the linker has about the program layout in memory.

The following example shows a simple linker directive file, which defines ROM and RAM address spaces, directs code and data to these address spaces, allocates a stack in the RAM address space, and defines a symbol whose address is the stack start address:

```
memory {
     ROM : origin=0 length=0x8000    /* ROM address space */
     RAM : origin=0xe000 len=0x800   /* RAM address space */
}

sections {
```

```
     .text into(ROM): { *(.text) }    /* Direct code to ROM */
     .bss into(RAM): { *(.bss) *(.data) }/* Direct data to RAM */
     .stack into(RAM): {. += 0x200;}  /* Allocate 0.5 K of RAM
                                         for program stack */
}

__STACK_START =  ADDR(.stack) + SIZEOF(.stack);
          /* Stack start symbol */
```

For a detailed description of the CompactRISC linker directive language refer to the *CompactRISC Toolset - Object Tools Reference Manual*.

-m option

If you want to produce a memory map of your program use the **-m** option of **crlink**. The output is directed to the standard output and can be redirected to a file.

Example

```
> crlink prog.o -m > prog.map
```

```
                CompactRISC LINKER MEMORY MAP


outputinput   memory size   section
sectionsectionaddress       contents

.text         0       38
      .text   0       38    main.o

.data         38      10
      .data_238       10    main.o

.bss          48      4
      .bss_2  48      4     main.o

UNUSED4c   to 8000    7fb4

.stack        e000    200
              e000    200   fill space

UNUSED e200 toe800    600
```

Note

If the linker reports an error because your program cannot fit into memory as specified in your linker directive file, you can still consult the partial memory map that the linker creates. This memory map shows you which parts of your program have already been located in memory and how much space is left in each memory segment. Thus you can determine why the remainder of your program cannot fit into memory.

-e option

Use the **-e** option of **crlink** to specify the entry point of your program. By the default **crlink** looks for a global symbol named **start** and takes its address as the entry point to your program. You can override this default by specifying the **-e** option and the alternative symbol whose address is to be used as entry point to your program. For example:

```
> crlink prog.o -o prog -e mystart
```

Note If you run your program stand-alone (i.e., you burn it and run it from ROM), make sure to place your start-up routine at address 0 so that your entry point symbol has the address 0. This is because CompactRISC-architecture CPUs start execution at address 0 after reset.
If your program is not run stand-alone (i.e., you use the CompactRISC Debugger on a development board, or in simulation mode) you can use an entry point other than address 0. The debugger starts executing your program from this entry point.

Main Linker Invocation Options

Option	Description
-o	Overrides the default name for the executable object file.
-l	Add standard libraries to the link process. Libraries are specified in the command line, after specifying the input object files.
-d	Specifies a linker directive file. The default directive file is **linker.def**
-m	Produces a memory map of your program. The output is directed to the standard output or can be redirected to a file.
-e	Specify the entry point of your program. By default **crlink** uses the global symbol **start** as the entry point to your program.
-z[n*filename*]	Dump errors and warnings into a file name.

2.7 READING ARGUMENTS FROM A FILE

In some environments, command lines are limited in size. To bypass this problem, some CompactRISC development tools use a convention which allows you to specify the command line arguments, or part of them, in a file. If you specify **@filename** as an argument, **filename** is read and its contents are used as part of the command line. For example:

```
>crcc @comflags.cmd prog1.c
```

2.8 USING THE DEBUGGER

Once you have compiled and linked your program, you can run it, and debug it, with the CompactRISC Debugger.

To invoke the CompactRISC Debugger, use the **crdb** command in your window environment. For example:

Invoke the debugger by double-clicking the CRDB icon.

In this chapter we discuss only a few of the **crdb** commands and options.

The debugger supports two execution modes:

- Simulation mode. The program is executed by a built-in CompactRISC instruction simulator and not on a real target hardware platform.
- Real execution mode. The program is downloaded to and executed on a target hardware platform. This platform may be an Application Development Board (ADB) which includes, among other components, a CompactRISC based microprocessor.

By default, the debugger executes programs in simulation mode. In this chapter we ignore the differences between the two execution modes and focus on more general aspects of debugging. For details about the different debugger targets refer to Chapter 3.

Step 1 Use the **Load** option in the **File** menu to load your program into the simulator. The debugger displays the source lines of your program's entry point in its source window.

Step 2 Use the **Go** button to run the program.
or
Set a breakpoint in one or more places in your program.

To set a breakpoint, display the desired point in your code in the source window and use the mouse to double-click on the source line in which you want your program to be stopped. To cancel this breakpoint, double-click it again. Now press the **Go** button. Your program execution starts and if it reaches any of the breakpoints you have set, the program stops. The debugger notifies you about the breakpoint you have just reached and displays its source code in the source window.

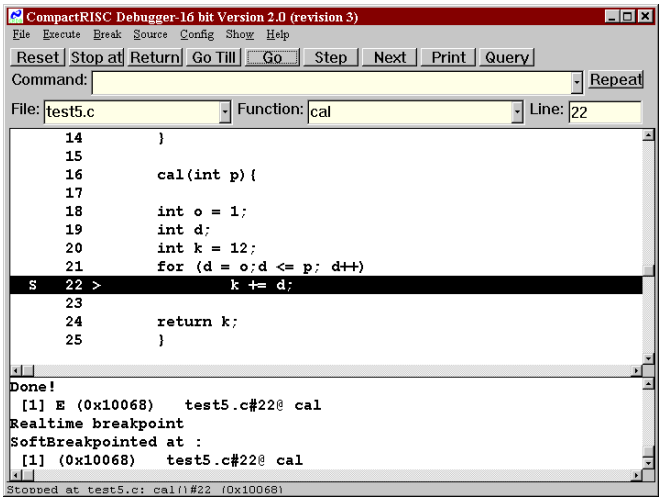


Figure 2-2. Debugger Main Window after Stopping at Breakpoint

Now that your program has stopped in a breakpoint you can examine its state in various ways. You can first examine the calling chain at this points by using the SHOW STACK option in the **Show** menu. You can inspect the value of variables or any part of your memory using the **Memory** option in the **Show** menu. You can see the contents of the registers in the **Register** option of the **Show** menu.

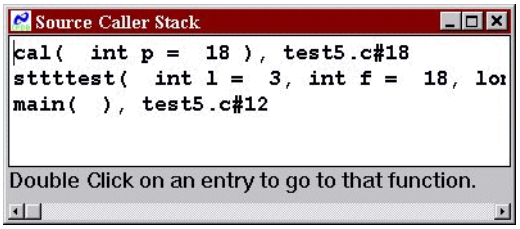


Figure 2-3. Caller Stack Window

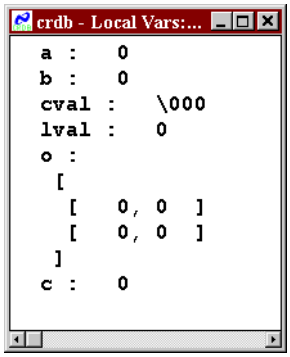


Figure 2-4. Local Variable Window

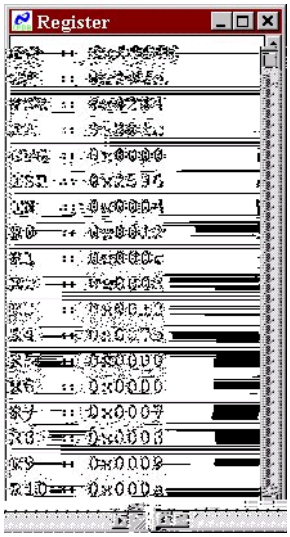


Figure 2-5. Register Window

Chapter 3
THE DEBUGGING ENVIRONMENT

3.1 INTRODUCTION

To run and debug your program, you must understand the CompactRISC debugging environment. So far we have discussed the CompactRISC Debugger, and mentioned that it supports both simulation and real execution modes. In this chapter we concentrate on the real execution mode.

In real execution mode, you run your program on a target system, called an Application Development Board (ADB). You use the debugger to download the program to the ADB, and then to run and debug it. To perform these operations a connection is needed between the CompactRISC Debugger, running on the host computer, and the target board. The debugger communicates with firmware on the board called Target MONitor (TMON) using the Debugger Communication Layer (DBGCOM) software.

Debugger Communication Interface

In the CompactRISC debugging environment, version 2.0 or higher, we use an additional component, the Debugger Communication Interface (DBGCOM), as a mediator between the debugger and a target-specific dll. The DBGCOM hides the physical communication media from the debugger by providing the debugger with a standard API to communicate with all targets, including a simulated target running on the host. The physical communication layer is thus fully transparent to the debugger.

3.2 THE TARGET MONITOR (TMON)

As mentioned above, each ADB contains software, called the target monitor (TMON), which is responsible for debugging support.

There are two families of TMON:

- Old TMON
  TMON ver 2.0.x or lower. To communicate with old TMON, link your application with the **libadb** start-up library.

- New TMON
  TMON ver 2.1.0 or higher. To communicate with new TMON, link your application with the **libstart** start-up library.
  This is the default start-up library.

Using the TMON protocol, the debugger can retrieve ADB status information (e.g., contents of registers and memory), or modify information on the ADB. It can also download the application code to the ADB. In addition, TMON can report different events that occur on the ADB (like breakpoints or other traps) to the debugger, and finally, it is used for virtual I/O requests issued by the application on the ADB. For more details about virtual I/O see Section 3.3.2.

There is a different version of TMON for each ADB. The appropriate TMON software version is installed on one or more EPROMs which are part of your ADB. If you develop your own ADB, you must customize the TMON software and make the necessary modification for this ADB. In a typical case most parts of TMON require no change, hence customization is not expected to be a major task.

To make your application independent of the TMON version, the CompactRISC Toolset has changed to a new TMON approach. In this new approach, TMON updates all the debugging traps located in your dispatch table (e.g., bpt traps) in run-time using the supervisor call. In the old TMON approach, you had to update the dispatch table in link time, i.e., TMON trap addresses had to be located in the linker definition file. Refer to your ADB reference manual to clarify which TMON you are using (usually, new TMON is version 2.0, or higher).

For proper communication with TMON, link your application with a start-up library that provides the necessary initializations. (The start-up library is provided with the CompactRISC Toolset).

Use the **libstart.a** start-up library to communicate with either a new TMON, or the simulator. This is the default start-up library.

Use the **libadb.a** start-up library to communicate with an old TMON. To use this library, you must define **__tmon**, **__eop** and **__vio** in your linker definition file, or the default file.

Environment	Start-up Library	Comments
Simulator	**libstart** *	
New TMON	**libstart** *	
Old TMON	**libadb**	The symbols **__tmon**, **__eop**, **__vio** must be defined in your linker directive file

* Default Start-up Library.

The start-up library is also provided in source form so that you can customize it for your application needs. For example, when you move your application to production mode you no longer need TMON services, and you can replace the default trap handlers. To customize any module of the start-up library, simply modify its source, compile it, and add it to the list of your objects in the linking phase before the start-up library so that it overrides the start-up library.

3.3 CONECTING THE APPLICATION TO TMON

TMON is an independent program which is not directly linked with your application. It resides in a pre-allocated memory area that typically requires about 1.5 - 2 Kbytes for its code and data. See your ADB manual for details of TMON's pre-allocated memory. Make sure that you do not allocate any part of this memory space to your program. This can best be done by dedicating an address range for TMON in your linker directive file. For example:

```
MEMORY {

 ROM:          origin=0,          length=0xa000
 RAM:          origin=0xa000,     length=0x1000
 /* The following address range is reserved for TMON */
 tmon_data:    origin=0xD800,     length= 0x280
 tmon_code:    origin=0xDB00,     length= 0x500
}
```

3.3.1 Sharing the Interrupt Dispatch Table

Although the application program is not linked with TMON, it shares one important resource with it. This resource is the interrupt dispatch table. This table contains addresses of all exception handlers. The exception list includes traps (e.g., breakpoint trap, trace trap) which are handled by TMON, and also application-specific interrupts (e.g., timer, UART) which are handled by the application.

There are two approaches for sharing this interrupt dispatch table between the application and TMON, depending on the TMON you are using (see also Section 3.2):

New TMON approach

Initially, TMON holds the interrupt table, as it is the first program to receive control on reset. However, the application can set up the interrupt table in its own address space. This is done as part of the start-up routine. To move the table to the application space, and maintain debugging capability, the first part of the table, containing the addresses of trap handlers that are handled by TMON, must be copied to the new location of the table. The application achieves this by generating a supervisor call (SVC) trap, that is a service request from TMON. TMON handles this request by copying the first 16 entries of the table to the new table address, that is provided as one of the parameters of the request. After the table has been moved to the application space, trap handlers are still handled by TMON when they occur, and thus it is possible to debug the application. The following example, taken from the start-up routine, demonstrates how the application sets up the interrupt table in its own space.

Example

The following start-up routine is for the CR16B core (large mode) used with new TMON:

```
movd    $__dispatch_table, (r2,r1)    # User table
movw    $0x102, r0
excp    svc                   # now TMON updates the user
                              # dispatch table.
movw    $0x100,r0             # Set dispatch table width
lpr     r0,cfg                # to 32 bits.
lpr     r1,intbasel           # Switch to user dispatch
lpr     r2,intbaseh           # table.
```

Old TMON approach

As in the new approach, above, TMON holds the dispatch table initially. However, once the application moves the dispatch table to its own space it also sets up its own trap handlers in the table. These new handlers are part of the `libadb` library that is linked with the application. These handlers simply mark the trap number, and jump into TMON. Thus trap handlers are still handled by TMON. To jump into TMON, the application uses a predefined symbol, `__tmon`. This symbol is typically defined in the linker directive file, and is assigned a hardwired address.

3.3.2 Virtual I/O Service

One important feature of the CompactRISC debugging environment is the virtual I/O mechanism. This feature is useful only during the development phase of your application. It enables your application to perform standard input and output to the host computer, and to access files on the host as if it were running on the host.

Virtual I/O is another case where the application has to communicate with TMON. Virtual I/O requests are handled by the host debugger through TMON.

The low-level virtual I/O routines, that are part of the CompactRISC C library (`libc`), issue an SVC trap with the appropriate parameters according to the virtual I/O request type. The SVC trap is handled by TMON, which transfers the request to the debugger. The debugger then handles the virtual I/O request using standard TMON services (i.e., read/write memory or registers). The debugger handles each virtual I/O low-level routine by knowing the calling convention of the CompactRISC Toolset, and the prototype of each routine.

3.3.3 End of Program

To exit a program the application also requires a TMON service. Once again there is a difference between old TMON and new TMON in the way this is done:

New TMON approach The application issues an SVC request with the appropriate parameter. TMON then reports the event to the debugger.

Old TMON approach The application jumps to a specific location in TMON, according to a predefined symbol, __eop. This symbol is typically defined in the linker directive file. TMON in turn reports end-of-program event to the debugger.

3.3.4 Summary of SVC Calls (new TMON only)

Each SVC call has one or more parameters which are passed in registers. The main parameter is the request type, which is passed in r0. The following table summarizes SVC request types:

```
0x101    Copy interrupts dispatch table
0x102    Copy interrupts dispatch table (CR16B large model)
0x401    Virtual I/O low-level file open request
0x402    Virtual I/O low-level file close request
0x403    Virtual I/O low-level file read request
0x404    Virtual I/O low-level file write request
0x405    Virtual I/O low-level file lseak request
0x406    Virtual I/O low-level file rename request
0x407    Virtual I/O low-level file unlink request
0x407    Virtual I/O low-level getenv request
0x410    End Of Program indication.
```

3.3.5 Summary of Symbol Definitions (old TMON only)

To link your application with the libadb.a start-up library, you must define some symbol in your linker directive file. These symbols are:

__tmon	TMON entry point for trap handlers
__eop	TMON entry point for program exiting

Chapter 4
ARCHITECTURE TOPICS

4.1 INTRODUCTION

This chapter deals with several aspects of the CompactRISC architecture which may interest you as a user of the CompactRISC Toolset. The first part explains the CompactRISC calling convention, while the second part is intended for CR16A/CR16B users only, and deals with some aspects which are unique to the 16-bit programming environment.

4.2 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 forms a handshake between different pieces of code (subroutines), and defines how control is transferred from one to another. It thus defines a general mechanism for calling subroutines and returning from subroutines.

When 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 we strongly recommend that you use ANSI-C function prototypes. Using function prototypes ensures that the 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.

Note that CR16A, CR32A and CR16B small programming model (see Section 4.3.1) are identical in terms of the calling convention's details. The CR16B large programming model calling convention, is very similar to the others, but has some minor differences.

4.2.1 Calling a Subroutine

The **BAL** or **JAL** instruction is used to call a subroutine. Each of these instructions performs two operations:

- Saves the address of the following instruction (the value of the Program Counter register) in a single general-purpose register or in a specified general-purpose register pair (for CR16B large model only).

  The calling convention requires the use of the RA register, or of the register pair (ERA, RA) (for CR16B large model only). This single register (or register pair) is used to store the return address.

- Transfers control to a specified location in the program (the subroutine address).

Example 1

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

or

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

Example 2 For CR16B large model only:

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

or

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

4.2.2 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 or in (ERA, RA) register pair (for CR16B large model only), as follows:

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

For CR16B large model only:

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

4.2.3 Passing Arguments to a Subroutine

Normally arguments to subroutines are passed in registers. The CompactRISC calling convention uses the general purpose registers **r2**, **r3**, **r4** and **r5** for this purpose. Usually, each argument is passed in one register.

For example, consider the following C line:

```
extern foo(int, int);
...
foo(5,7);
```

The following assembly code is generated for this subroutine call (on CR16A):

```
movw $5, r2          # pass 1st argument
movw $7, r3          # pass 2nd argument
bal ra, _foo         # call foo
```

If there are more than four arguments, registers **r2**-**r5** are used for passing the first four arguments. The remaining arguments are passed on the program stack (see below).

When a register can not hold an argument because of its size (e.g., long (double-word) on the CR16A/CR16B, or double-precision floating-point (quad-word) on CR32), it is passed in two registers.

The parameters of C functions with a variable number of arguments (vararg) like **printf** are always passed on the stack. Here is the place to emphasize the importance of function prototypes. When calling a vararg function without using a function prototype the arguments to the functions are not passed on the stack (as they should be) since the compiler does not know that the called function is of this kind. Therefore always use a function prototype when calling a vararg function as in the following example:

```
extern int vafunc(int, ...)
...
vafunc(3, a, b, c);
```

4.2.4 Returning a Value

A subroutine can return one value to its caller. The calling convention uses the R0 register for returning an integer-size value. In the 16-bit architectures the register pair R0 and R1 is used for returning a long (32-bit) integer value, with the least significant word in R0.

For example, consider the following C code:

```
return 5;
```

The assembly code generated from this line is:

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

For the CR16B large model, the assembly code generated from this line is:

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

The only exception to this rule is a function that returns a structure. In this case, the calling function must store the address of a structure in R0. The called function then uses R0 as a pointer to store the resulting structure.

4.2.5 Program Stack

The program stack is a contiguous memory space that may 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” on page 4-2.)
- Saving registers before calling a subroutine, or after being called. (See “Scratch Registers” on page 4-5.)

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 usually pointed to by the Stack Pointer (SP) register.

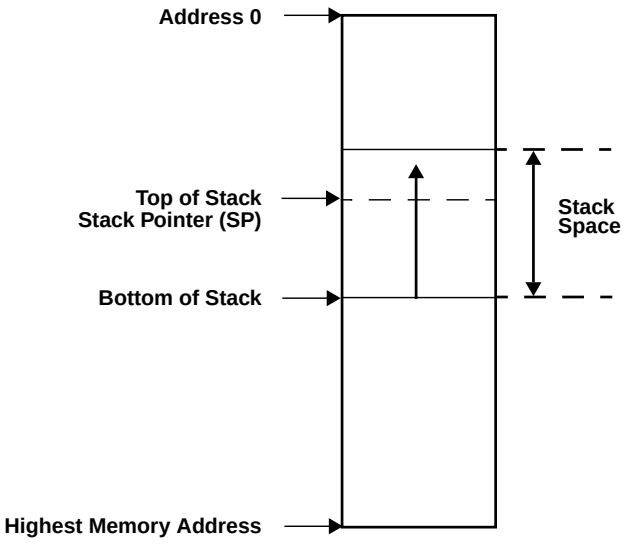


Figure 4-1. The Program Stack

A subroutine can allocate space on the stack by decrementing 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.

The program stack always resides in the lowest 64 Kbytes of the CR16B address space.

Note: (CR16B only) The **pop**, **push** and **popret** instructions assume that the SP general purpose register is used as the program stack pointer.

4.2.6 Scratch and non-Scratch Registers

According to the convention, CompactRISC general-purpose registers may be used as scratch registers or non-scratch registers.

Scratch Registers

The scratch registers are r0 to r6. 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 its responsibility to save the scratch register on the stack before calling the subroutine, and to restore it after the subroutine has returned.

Non-Scratch Registers

The non-scratch registers are r7 and above, including RA, SP, and ERA (CR16B large model only). Before using any of these registers, a subroutine must first store its previous value on the stack. Upon returning to the caller the subroutine must restore this value. The caller can always assume that these registers will not be clobbered by any subroutine that it has called.

Exception The interrupt/trap subroutine is an exception to the rule for using scratch registers. This kind of subroutine must always save and restore every scratch register that may 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.

4.3 16-BIT PROGRAMMING ISSUES (CR16 ONLY)

In this section we discuss some special issues which are unique to developing programs for the 16-bit architectures, or are related to differences between the CR16 and the CR32.

4.3.1 Small and Large Programming Models (CR16B only)

The CR16A architecture supports 128 Kbytes of program memory and 256 Kbytes of data memory. The CR16B architecture has enhanced addressing capabilities allowing it to support 2 Mbytes of memory for program and data. This is achieved with two programming models: small model and large model. Only the large model takes full advantage of the CR16B address space. The small model generates more optimized code and data, and in addition is binary backward compatible with the CR16A.

- Address space. Small model is limited to 128 Kbytes of code and 256 Kbytes of data. In the large model, code and data can reside anywhere in the CR16B 2 Mbyte address space.
- Some of the branch/jump assembly instructions require a register operand in small model and a pair of registers operand in large model.
- Some of the instructions are encoded differently in small model and large model. In other words, for a certain instruction, the assembler may generate different machine code in each model.
- In the small model, function pointers are 16 bits long. In the large model, function pointers are 20 bits long, but since their size is rounded-up they actually occupy 32-bits.

The CR16B C compiler and the CR16B assembler have options for generating large model code or small model code.

Note The National Semiconductor Toolset supports 256 Kbytes of data only.

4.3.2 Basic Types

The integer size on CR16 is 16 bits, as opposed to 32 bits on the CR32. Short integers are 16 bits long in both CR16 and CR32. Similarly, long integers are 32 bits long in both architectures. For your application to be efficient, and portable between 16 and 32-bit architectures, you should bear in mind the following rules:

- In general, use **`int`** for integer definitions.
- If an integer variable is not intended to hold more than 16 bits, and your application is space sensitive, define it as short.
- If an integer variable might have to hold more than 16 bits, define it as long.
- Wherever possible, define variables that are used as array indices as **`int`**.

4.3.3 Far Data

Both the CR16A, and the CR16B small programming model, support 256 Kbytes of programming memory. The large programming model of the CR16B supports 2 Mbytes of program and data memory.

In all the CR16 programming models, however, the data in the address space above 64 Kbytes (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 is 16 bits wide, they can point to any address in the range 0 - 64K. For a pointer to far data, the compiler must use far register relative addressing mode. Two 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:

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

Note The National Semiconductor Toolset supports 256 Kbytes of data only.

4.3.4 Code Addresses

CR16A and CR16B small model

The CR16A architecture, and the CR16B small programming model, support a code address space in the range of 0-128K (17-bit addressing). Since CR16 instructions must always start on an even address, the least-significant-bit of the address is always 0, and therefore is redundant. This makes it possible to encode code address into 16 bits. These 16 bits represent bits 1-16 of the address, while bit 0 (the least-significant-bit) is implied.

The CompactRISC C Compiler takes care of address encoding and thus this is transparent to you, as long as you write your program in C.

If, however, you write your program in CompactRISC assembly, and have a label of code, use the **`.code_label`** directive to define this label as a label of code (rather than data) to use a pointer to this place in the code.

Example

```
        .code_label abc
                        # qualify abc as a label of code
abc:    ...
        movw $abc, r5   # load the encoded address of abc to r5
        ...
        jump r5         # jump to the code address in r5
```

Since we define **abc** as a label of code, the **movw** instruction copies bits 1-16 of the address of **abc** (rather than bits 0-15) to **r5**, and this ensures that the jump instruction will later work properly.

Another good example is the interrupt dispatch table:

```
        .code_label       handler_1
handler_1:
...
        .code_label       handler_2
handler_2:
...
        .code_label       handler_3
handler_3:
...
__dispatch_table:
        .word     handler_1
```

```
        .word     handler_2
        .word     handler_3
        ...
```

The addresses of the handlers are encoded correctly in the dispatch table because the respective labels have been defined as labels of code.

CR16B large model

The code address space of the CR16B large programming model is 2 Mbytes. However a code address is encoded in 20 bits, and not in 21 bits address, because code addresses are always even and bit 0 is implied. The 20 bits are encoded over a double word (32 bits). As shown above it is important for an assembly programmer to qualify a code address using the **.code_label** directive. This ensures that the assembler uses the 20 bit encoding of the code address, rather than the full 21-bit address.

Example

```
        .code_label   abc
                    # qualify abc as a label of code
abc:  ...
        movd $abc,  (r6,r5)
                    # load the encoded address of abc to the
                    # register pair (r6,r5)
                    ...
        jump (r6,r5)
                    # jump to the code address in the
                    # register pair (r6,r5)
```

Note that bits 1-16 of the address are loaded into **r5** while bits 17-20 are loaded into **r6**.

The second example, as above, is an assembly definition of the interrupt dispatch table, this time in CR16B large model. Again the **.code_label** directive ensures that the addresses are encoded correctly.

Example

```
        .code_label          handler_1
handler_1:
...
        .code_label          handler_2
handler_2:
...
        .code_label          handler_3
handler_3:
...
__dispatch_table:
        .double    handler_1
        .double    handler_2
        .double    handler_3
```

Chapter 5
EMBEDDED PROGRAMMING TOPICS

5.1 INTRODUCTION

This chapter deals with various aspects of embedded application programming and the way they are reflected in the CompactRISC Toolset. These aspects are:

- Writing trap and interrupt handlers in C.
- Constant and volatile variables and their usage.
- Using the linker for different purposes.
- Data initialization.
- The interrupt dispatch table.
- Interrupt enabling and disabling, and nested interrupts.

The information in this chapter may be very useful for every embedded application programmer who uses the CompactRISC Toolset.

5.2 WRITING TRAP AND INTERRUPT HANDLERS IN C

The CompactRISC C Compiler allows trap and interrupt handlers to be developed entirely in the C language. You do not need to write any part of the handler in assembly. In order to develop your trap/interrupt handler in C, you write it as a normal C function of type **`void`** with no arguments. The only special action you must take is to use the **`#pragma`** directive to inform the compiler that this C function is an interrupt or trap handler.

Example

```
extern volatile int time;

#pragma interrupt (clock_int)

void clock_int()
{
     time++;
}
```

The function `clock_int` is declared as an interrupt handler.

When the compiler generates code for a function which is declared as an interrupt, or trap handler, it does the following:

- Uses the **retx** (return from exception) instruction for returning from the handler.
  This instruction restores the PC and PSR registers from the interrupt stack (pointed to by the ISP register).

- Saves all the scratch registers used by the function on the stack. Normally a function can use scratch registers freely without first saving them (see Section 4.2). However since a handler function occurs asynchronously it must save scratch registers too.
  If this function calls another function, all the scratch registers are saved, since the caller does not know which scratch registers are modified by the called function.

Following is an example of assembly code generated by the CompactRISC C Compiler for the previous example.

Example

```
_clock_int:
        addw
                $-2,sp
        storw   r0,0(sp)
        loadw   _time:m,r0
        addw    $1,r0
        storw   r0,_time:m
        loadw   0(sp),r0
        subw    $-2,sp
        retx
```

5.3 CONST VARIABLES

When you develop an embedded application program, you often put the constant and read-only part of your data in ROM. This part of the data should be distinguished from other, non-constant, data. To this end, use the ANSI C **const** qualifier when you declare such data.

Example

```
const char days[7][3] = { "Mon", "Tue", "Wed", "Thu",
                                  "Fri", "Sat", "Sun" };
```

The CompactRISC C Compiler directs **const** data to different sections that can be later on located in ROM. Note that **const** data must have an initial value. For initializing non-constant data (which resides in RAM) refer to Section 5.6.

5.4 VOLATILE VARIABLES

When you develop an embedded application program in the C language, you must be aware of volatile variables. The volatile qualifier is defined by ANSI C, and as such is supported by the CompactRISC C Compiler. It supports embedded applications in case where a certain memory reference must take place and thus the compiler must avoid using optimizations which eliminate this memory reference.

Normally, the CompactRISC C Compiler can perform several kinds of optimizations which affect the access to a variable in your program. For example:

- The compiler can decide to put a variable in a register.
- The compiler can remove (optimize out) an access to a variable which looks redundant.
  For example, if you have two assignments to the same variable, and you do not use this variable in-between, the compiler can decide that the first assignment is redundant.

In some cases, you do not want the compiler to perform these optimizations. In these cases, use the volatile qualifier in the variable declaration:

```
volatile int i;
```

The access to the integer variable **i** is not optimized by the compiler.

Why would you want to declare a variable as volatile? We discuss here two examples which explain the motivation.

The first example deals with a variable which may be asynchronously accessed by an interrupt. Suppose you have a clock interrupt handler which modifies the time, and some other function that reads the time every now and then. For this function to access the time variable correctly, it should reside in memory, and must be accessed from memory. You do not want the compiler to put this variable in a register, because this register will not reflect the up-to-date value of the time as it is updated by the interrupt handler. This variable should be declared as volatile.

Example 1 clock interrupt handler:

```
volatile int time;
#pragma interrupt (clock_int)
void clock_int()
{
  time++;
}
```

another function:

```
extern volatile int time;
display_state()
{
   ...
   if (time > 1000) do_something();
   ....
}
```

The second example concerns variables that represent memory-mapped I/O registers. When you access these variables, you actually access a peripheral I/O device (e.g., clock, UART, codec, etc.). For this reason you do not want the compiler to put these variable in registers. You want your program to access the memory address of the memory-mapped I/O register. Therefore you define them as volatile.

Example 2

```
volatile unsigned char UART_reg;
...
UART_reg = 'x';     /* send the char 'x' to the serial line */
```

Another motivation for defining memory-mapped I/O registers as volatile variables is that you may have consecutive writes to the same variable without using the variable in between.

Example 3

```
volatile unsigned char UART_reg;
...
UART_reg = 'h';
UART_reg = 'e';
UART_reg = 'l';
UART_reg = 'l';
UART_reg = 'o';
```

By default the compiler may think that all the writes except the last one are redundant, and may be optimized out. However this is not the case because each time you write to this variable there is a side-effect on the I/O device. Since the variable is declared here as volatile, the compiler does not remove any of the lines in this example, and all the writes are actually run.

5.5 USING THE LINKER DIRECTIVES

One of the most useful tools in the CompactRISC Toolset is the linker directive file. This subject was briefly discussed in Section 2.6. In this section we give several examples which demonstrate the use of linker directives in embedded applications.

5.5.1 Binding the Start-up Routine to Address 0

When a CompactRISC-based microprocessor wakes up, it starts executing instructions from address 0. Therefore, when you link a program which is to run stand-alone (i.e., not in the debugging environment) you must make sure that the entry point to your program is in address 0. In other words, you must bind your start-up routine to address 0. The right way to do this is to use linker directives to control the location of the start-up code. With linker directives you can control the location of any of the sections of the input object files.

Example

```
memory {
      ROM : origin=0 length=0x8000
      ...
}

sections {
      .text BIND(0) : { start.o(.text) }
      .text into(ROM) : { *(.text) }
      ...
}
```

In this example there are two **.text** (code) output sections. The first section contains only the code from the object **start.o**, and is bound to address 0. The second **.text** section contains the code from all the input objects that have not yet been located. In other words, it contains the code from all the objects except **start.o**. The second **.text** section is directed to ROM without binding it to any specific address.

5.5.2 Using Overlays to Save Space

Very often, in embedded applications which are sensitive to the RAM size, you must use the same address space in RAM for different entities that are not used at the same time. The linker directives support this requirement in several ways. In the first example we look at two large buffers in RAM, which are not active simultaneously, and using linker directives we allocate a special section for both of them:

Example

```
memory {   ...
      RAM : origin 0xc000 len=0x2000
      ...
}

sections {
      ...
      .overlay into(RAM) : { . += 0x400; } /* allocate 1K for
                                              buffers */
}
```

```
_buf1 = addr(.overlay);
_buf2 = addr(.overlay);
```

As you can see, linker directives create a special section, **.overlay**. This special section is not composed from input sections. It just allocates 1K bytes in RAM. Later, we assign two symbols (**_buf1** and **_buf2**) to the start address of the **.overlay** section. This creates an overlay of two buffers, each of which can use space up to the section size (1K in this example). These buffers, defined in the linker directive file, can be used in a C program. For example:

```
extern unsigned char buf1[];
```

In the next example we look at two modules which are not active at the same time, and therefore their static variables can share the same address space in RAM (by static we mean non-automatic, i.e., not located on the program stack). This can also be done with linker directives.

Example

```
memory {
      ...
      RAM : origin 0xc000 len=0x2000
      ...
}

sections {
      ...
      .overlay into(RAM) : { transmit.o(.bss)
                             . = addr(.overlay);
                             receive.o(.bss)
                           }
      ...
}
```

In this example two modules, transmit and receive, use the same space for their static variables. The **.overlay** sections contain the **.bss** (uninitialized variables) input sections, which are not located one after the other. After locating **transmit.o**, the location counter is returned to the beginning of **.overlay**, and thus **receive.o** and **transmit.o** are located in the same place.

5.5.3 Code Overlay Allocation

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

The **-O** flag informs the linker to enable the code overlay mechanism.

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

Example

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

You should use the following linker definition file:

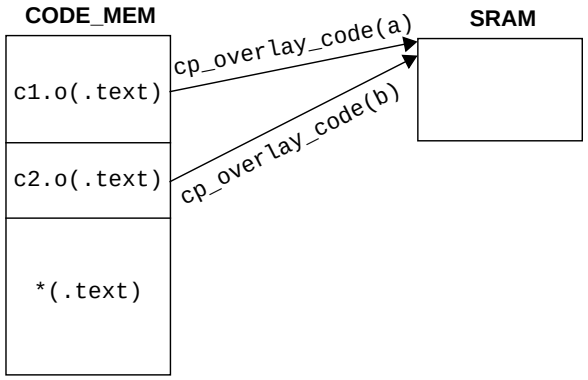
```
MEMORY {
        ...
     CODE_MEM:
        ...
     SRAM:     origin = 0x1000, length = 0x500
        ...
}

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

The application code that calls these functions should be:

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

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

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



5.5.4 Bank Switching

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

Notes

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

The CompactRISC debugger does not support Bank Switching.

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

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

Mediator function

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

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

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

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

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

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

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

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

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

and after **`g()`** returns:

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

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

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

Common-memory

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

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

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

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

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

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

where:

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

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

Example

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

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

     bank0_code_mem:
          origin=0x12000,        length=0xC000

     bank1_code_mem:
          origin=0x32000,        length=0xC000

     bank2_code_mem:
          origin=0x52000,        length=0xC000

     .
     .
     .
}
```

Note

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

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

```
SECTIONS {


.text ALIGN(2) INTO(common_memory): { int1.o(.text) int2.o(.text) ... }
.rdata ALIGN(2) INTO(common_memory): { *(.rdata_2) *(.rdata_1) }
```

```
      .
      .
      .
.text ALIGN(2) INTO(bank0_code_mem): { a.o(.text) }
.text ALIGN(2) INTO(bank1_code_mem): { b.o(.text) }
.text ALIGN(2) INTO(bank2_code_mem): { c.o(.text) }

}
```

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

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

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

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

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

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

5.5.5 Using Link-time Information

In some cases you may find it necessary to use the information that the linker has during link-time in your program. Linker directives provide ways to access useful link-time information that cannot be obtained in any other way. In the following example we want to allocate space to a buffer in a flexible manner. We want this buffer to be as large as possible, while limiting it to the amount of free space which is still left in RAM. Of course, as the program is modified, the amount of free space may increase or decrease, and we want this fact to be transparent to the program.

Example

```
memory {
      ...
      RAM : origin 0xc000 len=0x2000
      ...
}

sections {
      ...
      .bss into(RAM) : { ... }
}

/* Free memory space starts just after the .bss section */
_buf = addr(.bss) + sizeof(.bss);
```

```
/* The amount of free space is end of RAM minus start
   address of the free memory */
_buf_size = 0xe000 - ( addr(.bss) + sizeof(.bss) );
```

Two symbols were defined in this example. These two symbols provide the link-time information to the program. They can be used in a C program as in the following example:

```
extern char buf[]; #define BUF_SIZE (&buf_size) ...

/* Fill buf with binary 1's */
for (i = 0; i < BUF_SIZE; i++)
   buf[i] = 0xff;
```

Note that we use **&buf_size**, and not simply **buf_size**, to get the size of our buffer because the linker assigni thalue, toanyo symbos in thuctve
ss.C Ie thbuve exampl,e thehalue,isps sigsed t2

The CompactRISC Development Toolset supports two aspects of data initialization:

- Copying initialized data from ROM to RAM
- Clearing uninitialized data to zero

The CompactRISC Linker, in conjunction with the CompactRISC start-up code, (which is part of **libadb**) supports data initialization. The linker generates a special initialization table as part of the executable file. This table contains the information which assists the start-up code in performing the necessary data initializations. The initialization table has two types of entries which match the two types of data initialization:

- An entry which describes a section of initialized data. This entry includes ROM address, RAM address and section size.
- An entry which describes a section of uninitialized data. This entry includes RAM address and section size.

Note that, for a section of initialized data, you must specify both a ROM address and a RAM address in the linker directive file if you want it to be initialized. Refer to the *CompactRISC Toolset - Object Tools Reference Manual* for more details.

The start-up routine in **libadb** may call one of several initialization functions which are also part of **libadb**. By default, it calls a function which copies initialized data sections from ROM to RAM, and clears uninitialized data sections. If you like, you can customize the start-up routine to call a different initialization function. For example if your program has no initialized data but you do want to clear the uninitialized data you can call an initialization function which does only the latter. Since this function is smaller, you benefit in code size. If your program does not require data initialization you can remove the call to data initialization function. In this case, you save both the function code and the initialization table space.

Note that the CompactRISC C run-time library requires data initialization. Thus, if you are using this library in your program, you should not skip data initialization.

5.7 THE START-UP ROUTINE

Embedded applications normally involve code which is executed before your “main” routine, and performs any initializations essential for running your program. For example, you cannot run your C program before the stack Pointer (SP) register has been initialized to point to the bottom of the stack. These initializations are part of the start-up routine which is run immediately after the CompactRISC microprocessor leaves the reset state.

The CompactRISC Toolset includes the ADB library (**libstart.a** or **libadb.a**) which provides a default start-up routine. Like all other modules in the ADB library it is provided in both source code and object form. This makes it possible for you to customize the start-up routine for your application needs. For being able to debug your program you must link it with the appropriate start-up library according to the target to be communicate with for more details refer to TBD.

The default start-up routine in the CompactRISC Toolset is provided in assembly language since no C code can be executed before some initializations are performed. Its entry point, which is also the entry point to your program, is labeled by the name **start**. Note that **start** is also the default entry point symbol used by the CompactRISC Linker.

The default start-up routine carries out the following:

- Initializes the **INTBASE** register.
  It assigns the address of the default interrupt dispatch table to this register. The default interrupt dispatch table is another customizable module in the ADB library.
- Initializes the SP (Stack Pointer) register.
  It assigns the address of the bottom of the stack to this register.
- Initializes the ISP (Interrupt Stack Pointer) register.
  It assigns the address of the bottom of the interrupt stack to this register.
- Calls the **init_bass_data** routine which is responsible for data initialization (both initialized and uninitialized data). For further details, see Section 5.6.
- Calls the **main** routine, which is the actual entry point to your program.
  If **main** ever returns (which does not normally happen in embedded applications) it calls the **exit** routine.

The following is an example of a start-up routine taken from **libstart.a** (CR16B large model):

```
start::
  movd    $__dispatch_table, (r2, r1)# Initialize intbase.
  movw    $0x102, r0
  excp    svc                # Initialize debugger traps
                             # in user dispatch tabel.
  movw    $0x100, r0         # Set dispatch table width
  lpr     r0, cfg            # to be 32 bits.
  lpr     r1, intbasel
  lpr     r1, intbasel

  movw    $__STACK_START, r0  # Initialize program stack.
  movw    r0, sp
```

```
   movw    $__ISTACK_START, r0 # Initialize interrupt stack.

   lpr     r0, isp

   bal     ra, _init_bss_data  # Initialize data and bss

   bal     ra, _main           # C program entry point.

   movw    r0, r2              # main's return value is a
                               # parameter for exit.
   br      _exit
```

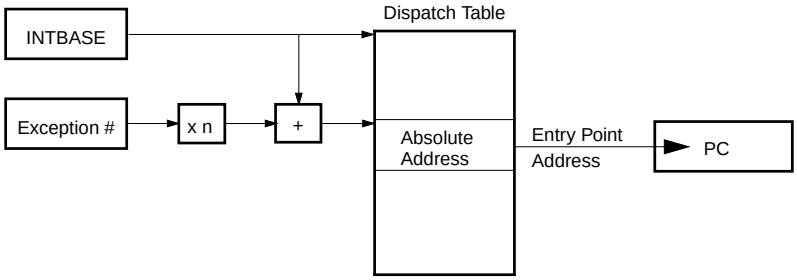
The following is an example of a start-up routine taken from **libadb.a** (CR16A model):

```
start::
   movw    $__dispatch_table, r0# Initialize intbase.
   lpr     r0,intbase

   movw    $__STACK_START, r0  # Initialize program stack.
   movw    r0, sp

   movw    $__ISTACK_START, r0 # Initialize interrupt stack.

   lpr     r0, isp

   bal     ra, _init_bss_data  # Initialize data and bss

   bal     ra, _main           # C program entry point.

   movw    r0, r2              # main's return value is a
                               # parameter for exit.
   br      _exit
```

Note that some of the symbols that are used in the start-up routine are defined, for convenience, in the default linker directive file. For example the symbol **__STACK_START** which marks the program stack bottom is defined as the end address of the **.stack** section in the default linker directive file.

5.8 THE INTERRUPT DISPATCH TABLE

The addresses of all exception handlers, including traps and interrupts, are specified in the interrupt dispatch table. The CompactRISC programming model includes a register called **INTBASE** which contains the address of the interrupt dispatch table. When an exception occurs, the CompactRISC processor uses the **INTBASE** register to determine the location of the interrupt dispatch table. Each exception has a number which is used as an index to the interrupt dispatch table to find the address of the required exception handler.



The interrupt dispatch table has two parts. The first part is used for exception handlers which are common to all CompactRISC-based processors e.g., non-maskable interrupt (NMI) handler, breakpoint trap (BPT) handler, division-by-zero trap (DVZ) handler. The second part is used for handlers of specific interrupts that exist on a certain derivative of the CompactRISC family. These interrupts originate from sources which are peripherals for the CompactRISC core, and reside either on-chip or externally. All these interrupts are controlled by an Interrupt Control Unit (ICU). The ICU is the component that sends the interrupt trigger to the CompactRISC core and also tells the CompactRISC core the number of the interrupt which is currently pending.

```
void (*const _dispatch_table[]) = {
     0,
     nmi,  /* Non-Maskabale Interrupt handler */
     0,    /* Reserved */
     0,    /* Reserved */
     0,    /* Reserved */
     svc,  /* Supervisor call trap handler */
     dvz,  /* Divide by Zero trap handler */
     flg,  /* Flag trap handler */
     bpt,  /* Breakpoint trap handler */
     trc,  /* Trace trap handler */
     und,  /* Undefined Instruction trap */
     0,    /* Reserved */
     0,    /* Reserved */
     0,    /* Reserved */
     0,    /* Reserved */
     ise,  /* In-System Emulator interrupt */
     clock_hnd,      /* Clock interrupt handler */
     uart_hnd,       /* UART interrupt handler */
     codec_hnd,      /* Codec interrupt handler */
     mw_hnd          /* MICROWIRE interrupt handler */
};
```

In your application program, you have full control over the interrupt dispatch table. You can install your exception handlers either statically (i.e., the same exception handlers throughout the program execution) or dynamically (replace one exception handler by another during execution). You can put your interrupt dispatch table in either ROM or RAM.

When you link your program with the ADB library (**libstart** or **libadb.a**) you get a default interrupt dispatch table, and a set of default exception handlers.

You can always customize this table by modifying all or parts of the table, or even the exception handlers themselves.

Note that some of the default exception handlers are essential for program debugging, and you should not override them in the development phase of the application. The default exception handlers are the breakpoint trap (BPT) handler, the trace trap (TRC) handler and the ISE handler. These handlers call the target monitor (TMON) on the ADB for debugging purposes. In the final (production) software version these handlers are no longer needed.

To override any part of the ADB library, modify any of the source files of the library (provided with the CompactRISC Toolset), compile it, and add the object file to the object list in the linking phase *before* the ADB library.

5.9 ENABLING AND DISABLING INTERRUPTS

If you want the application program to serve interrupts, you must first enable maskable interrupts. This is done by setting the **I** bit of the **PSR** register. Normally, you should do this shortly after reset. From a C function you can use the **set_i_bit** macro for this purpose.

```
#include <asm.h>
...
set_i_bit(); /* Enable maskable interrupts. */
```

Another reason to enable interrupts is to enable nested interrupts. When an interrupt or trap is serviced, the CompactRISC CPU automatically clears the **I** bit of the **PSR** register. Thus, at this point nested interrupts are disabled. If you want to enable nested interrupts, set the **I** bit of the **PSR** from within the interrupt handler.

```
#include <asm.h>
...
#pragma interrupt (int_handler)
void int_handler()
{
```

```
    set_i_bit(); /* Enable nested interrupts. */
    ...
}
```

By default, nested interrupts are disabled. If, in the interrupt handler, you set the **I** bit of the **PSR**, all maskable interrupts can be nested. You can, however, selectively enable maskable interrupts as nested interrupts by setting up the Interrupt Control Unit (ICU) of your chip to enable specific interrupts only. This must be done in the interrupt handler just before you set the **I** bit of the **PSR** enable nested interrupts. For information about the ICU, refer to the chip specification or datasheet.

The **E** bit, or local interrupt enable bit of the **PSR** register is related to interrupt enabling. Interrupts are enabled only when both the **I** bit and the **E** bit of the **PSR** register are set. However, the **E** bit is set on reset, whereas the **I** bit is cleared. In addition, the CompactRISC instruction set includes two dedicated instructions, **di** and **ei**, for clearing and setting the **E** bit, respectively.

Use the **E** bit, and its two dedicated instructions, for local interrupt disabling. For example, if you have a critical section in your code, which should be protected from interrupts, enclose it between **di** and **ei** instructions. From a C function you can use the **_di_** and **_ei_** macros.

```
#include <asm.h>
...
_di_(); /* Locally disable interrupts */
/* critical section starts here */
...
_ei_();
```

When the **E** bit is cleared, using the **di** instruction, maskable interrupts are disabled. When the **E** bit is set, using the **ei** instruction, interrupts are not necessarily enabled. Their previous status is restored. If interrupts were initially enabled (**I** bit was set), they are enabled again. On the other hand, if they were initially disabled (**I** bit was cleared), they remain disabled. Thus, the **di** and **ei** pair is used to disable interrupts locally, without changing their global status.

INDEX

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z