RealView™ Compilation Tools

Version 2.0

Developer Guide

ARM®

ARM DUI 0203C

RealView Compilation Tools
Developer Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
August 2002	A	Release 1.2
January 2003	B	Release 2.0
September 2003	C	Release 2.0.1 for RVDS 2.0

Proprietary Notice

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

The information in this document is final (information on a developed product).

Web Address

`http://www.arm.com`

Contents

RealView Compilation Tools Developer Guide

Preface

This preface introduces the *RealView Compilation Tools v2.0 Developer Guide*. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page ix.

About this book

This book provides tutorial information on writing code targeted at the ARM family of processors.

Intended audience

This book is written for all developers writing code for the ARM. It assumes that you are an experienced software developer, and that you are familiar with the ARM development tools described in the *RealView Compilation Tools v2.0 Essentials Guide*.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to *RealView Compilation Tools* (RVCT).

Chapter 2 ***Embedded Software Development***

Read this chapter for details of how to develop embedded applications with RVCT. It describes the default RVCT behavior in the absence of a target system, and how to tailor the C library and image memory map to your target system.

Chapter 3 ***Using the Procedure Call Standard***

Read this chapter for details of how to use the *ARM-Thumb Procedure Call Standard*. Using this standard makes it easier to ensure that separately compiled and assembled modules work together.

Chapter 4 ***Interworking ARM and Thumb***

Read this chapter for details of how to change between ARM state and Thumb® state when writing code for processors that implement the Thumb instruction set.

Chapter 5 ***Mixing C, C++, and Assembly Language***

Read this chapter for details of how to write mixed C, C++, and ARM assembly language code. It also describes how to use the ARM inline and embedded assembler from C and C++.

Chapter 6 ***Handling Processor Exceptions***

Read this chapter for details of how to handle the various types of exception supported by ARM processors.

Chapter 7 ***Debug Communications Channel***

Read this chapter for a description of how to use the *Debug Communications Channel* (DCC).

Typographical conventions

The following typographical conventions are used in this book:

`monospace` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

<u>`mono`</u>`space` Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda, and the ARM Frequently Asked Questions.

ARM publications

This book contains general information on developing applications for the ARM family of processors. Refer to the following books in the RVCT document suite for information on other components:

- *RealView Compilation Tools v2.0 Essentials Guide* (ARM DUI 0202)

- *RealView Compilation Tools v2.0 Assembler Guide* (ARM DUI 0204)
- *RealView Compilation Tools v2.0 Compiler and Libraries Guide* (ARM DUI 0205)
- *RealView Compilation Tools v2.0 Linker and Utilities Guide* (ARM DUI 0206).

The following additional documentation is provided with RealView Compilation Tools:

- *ARM FLEXlm License Management Guide* (ARM DUI 0209). This is supplied in DynaText and PDF format.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied as a PDF file, `ARMELF.pdf`, in *`install_directory`*`\Documentation\Specifications\1.0\release\`*`platform`*`\PDF`.
- *TIS DWARF 2 specification*. This is supplied as a PDF file, `TIS-DWARF2.pdf`, in *`install_directory`*`\Documentation\Specifications\1.0\release\`*`platform`*`\PDF`.
- *ARM-Thumb Procedure Call Standard specification*. This is supplied as a PDF file, `ATPCS.pdf`, in *`install_directory`*`\Documentation\Specifications\1.0\release\`*`platform`*`\PDF`.

In addition, refer to the following documentation for specific information relating to ARM products:

- *RealView ARMulator ISS v1.3 User Guide* (ARM DUI 0207)
- *Multi-ICE v2.2 User Guide* (ARM DUI 0048), or later version
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

The following book gives general information about the ARM architecture:

- *ARM System-on-chip Architecture* (second edition), Furber, S., (2000). Addison Wesley. ISBN 0-201-67519-6.

Feedback

ARM Limited welcomes feedback on both RealView Compilation Tools, and its documentation.

Feedback on RealView Compilation Tools

If you have any problems with RealView Compilation Tools, contact your supplier. To help them provide a rapid and useful response, give:

- your name and company
- the serial number of the product
- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you notice any errors or omissions in this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the *RealView™ Compilation Tools* (RVCT). It contains the following sections:

- *About the RVCT Developer Guide* on page 1-2
- *General programming issues* on page 1-5
- *Developing for the ARM processors* on page 1-10.

——— **Note** ———

The examples given in this document use single-dashes for the options to the compiler, assembler and linker. In RVCT v2.0 you can specify command-line keywords using double dashes --, for example `--partial`. However, the single-letter options in the compiler, for example `-E`, do not have double-dash equivalents.

The single-dash command-line options used in previous versions of ADS and RVCT are still supported for backwards-compatibility.

———————————

1.1 About the RVCT Developer Guide

This book contains information that helps you with specific issues when developing code for the ARM family of RISC processors. In general, the chapters in this book assume that you are using the *RealView Compilation Tools* (RVCT) to develop your code.

RVCT consists of a suite of tools, together with supporting documentation and examples, that enable you to write and build applications for the ARM family of RISC processors. You can use RVCT to build C, C++, or ARM assembly language programs.

The RVCT toolkit consists of the following major components:

- command-line development tools
- utilities
- supporting software.

See *Further reading* on page vii for a list of the RVCT documentation.

1.1.1 Example code

The code for many of the examples in this book is located in the directory:

`install_directory\RVCT\Examples\2.0\release\platform`

where `install_directory` is the directory where you installed RVCT v2.0, and `platform` is either `windows` or `unix`. For clarity, this directory is referred to as `Examples_directory` in the rest of this document.

In addition, the directory contains example code that is not described in this book. Read the `readme.txt` in each example directory for more information. The examples are installed in the following subdirectories:

`asm`	This directory contains some examples of ARM assembly language programming. The examples are used in the *RealView Compilation Tools v2.0 Assembler Guide*.
`cpp`	This directory contains some simple C++ examples. In addition, the subdirectory `rw` contains the Rogue Wave manual and tutorial examples.
`databort`	This directory contains design documentation and example code for a standard data abort handler.
`dcc`	This directory contains example code that demonstrates how to use the Debug Communications Channel. The example is described in Chapter 7 *Debug Communications Channel*.

`cached_dhry`		This directory contains examples to routines to initialize cache and TCMs, built around the Dhrystone example.
`dhry`		This directory contains source for Dhrystone.
`dhryansi`		This directory contains an ANSI C version of Dhrystone.
`emb_sw_dev`		This directory contains the example projects referenced in Chapter 2 *Embedded Software Development*. The following subdirectories are included:
	`build`*`n`*	Batch and make files to build the example projects.
	`dhry`	Source files for the Dhrystone benchmarking program. This program provides the code base for the example projects in the individual `build`*`n`* directories.
	`source`	All other source files needed to build the example projects.
	`include`	User defined header files.
	`scatter`	Scatter files used to build the example projects.
`embedded`		This directory contains source code examples that show how to write code for ROM. The examples are targeted at the ARM Integrator™ board.
`explasm`		This directory contains additional ARM assembly language examples.
`fft_5te`		Fast Fourier Transform code optimized for ARM architecture v5TE.
`inline`		This directory contains examples that show how to use the inline assembler when compiling ARM C and C++ code. The examples are described in Chapter 5 *Mixing C, C++, and Assembly Language*.
`interwork`		This directory contains examples that show how to interwork between ARM code and Thumb code. The examples are described in Chapter 4 *Interworking ARM and Thumb*.
`mmugen`		This directory contains the source and documentation for the MMUgen utility. This utility can generate MMU pagetable data from a rules file that describes the virtual to physical address translation required.
`picpid`		This directory contains an example of how to write position-independent code. See the `readme.txt` for a detailed description.
`sorts`		This directory contains example code that compares an insertion sort, shell sort, and the quick sort used in the ARM C libraries.
`swi`		This directory contains an example SWI handler.

`unicode`	This directory contains example code that enables you to test the multibyte character support in RVCT v2.0.
`vfpsupport`	This directory contains example code for enabling and carrying out *Vector Floating Point* (VFP) operations. Also included are various utility files for configuring the debug system when using VFP.

1.2 General programming issues

The ARM family of processors are RISC processors. Many of the programming strategies that give efficient code are generic to RISC processors. As with many RISC processors, the ARM family of processors are designed to access aligned data, that is, words that lie on addresses that are multiples of four, halfwords that lie on addresses that are multiples of two. This data is located on its natural size boundary.

ARM compilers normally align global variables to these natural size boundaries so that these items can be accessed efficiently using the `LDR` and `STR` instructions.

This contrasts with most CISC architectures where instructions are available to directly access unaligned data. Therefore, you must take care when porting legacy code from CISC architectures to the ARM processors. Accesses to unaligned data can be expensive in code size or performance.

The following sections discuss the programming issues in more detail:

- *Using the ARM-Thumb Procedure Call Standard efficiently*
- *Unaligned pointers*
- *Unaligned fields in structures* on page 1-6
- *Unaligned LDR for accessing halfwords* on page 1-8
- *Porting code and detecting unaligned accesses* on page 1-9.

1.2.1 Using the ARM-Thumb Procedure Call Standard efficiently

Under the *ARM-Thumb Procedure Call Standard* the ARM compiler passes the first four integer-sized function parameters in registers r0 to r3. Additional parameters are passed on the stack. This means that:

- it is more efficient to pass large parameters, such as **`struct`**s, by reference
- it is more efficient to restrict functions to four or fewer integer-sized parameters, where possible.

1.2.2 Unaligned pointers

The ARM compiler expects normal C pointers to point to an aligned word in memory, because this enables the compiler to generate more efficient code.

For example, if the pointer `int *` is used to read a word, the ARM compiler uses an `LDR` instruction in the generated code. This works as expected when the address is a multiple of four (that is, on a word boundary). However, if the address is not a multiple of four, then an `LDR` instruction returns a rotated result rather than performing a true unaligned word load. The rotated result depends on the offset and endianness of the system.

If your code loads data from a pointer that points to the address 0x8006, for example, then you might expect to load the contents of bytes from 0x8006, 0x8007, 0x8008, and 0x8009. However, on an ARM processor, this access loads the rotated contents of bytes from 0x8004, 0x8005, 0x8006, and 0x8007.

Therefore, if you want to define a pointer to a word that can be at any address (that is, that can be at a non-natural alignment) then you must specify this using the __packed qualifier when defining the pointer:

```
__packed int *pi; // pointer to unaligned int
```

The ARM compiler does not then use an LDR, but generates code that correctly accesses the value regardless of the alignment of the pointer. This generated code is a sequence of byte accesses or, depending on the compile options, variable alignment-dependent shifting and masking. Therefore a performance and code size penalty is incurred.

Note

You must not access memory-mapped peripheral registers using __packed, because the ARM compiler can use multiple memory accesses to retrieve the data. Therefore, nearby locations can be accessed, which might correspond to other peripheral registers. When bitfields are used, the ARM compiler currently accesses the entire container, not just the field specified.

1.2.3 Unaligned fields in structures

In the same way that global variables are located on their natural size boundary, so are the fields in a structure. This means that the compiler often has to insert padding between fields to ensure that fields are aligned. Compile with -W+s to generate a warning when the compiler inserts padding.

Sometimes, you might not want this to occur. You can use the __packed qualifier to create structures without padding between fields, and these structures require unaligned accesses.

If the ARM compiler knows the alignment of a particular structure, it can work out whether or not the fields it is accessing are aligned within a packed structure. In these cases, the compiler carries out the more efficient aligned word or halfword accesses, where possible. Otherwise, the compiler uses multiple aligned memory accesses (LDR, STR, LDM, and STM) combined with fixed shifting and masking to access the correct bytes in memory.

Whether these accesses to unaligned elements are done inline or by calling a function is controlled by using the compiler options -Ospace (default, calls a function) and -Otime (do unaligned access inline).

For example:

1. Create a file `foo.c` that contains:

   ```
   __packed struct mystruct {
       int aligned_i;
       short aligned_s;
       int unaligned_i;
   };
   struct mystruct S1;

   int foo (int a, short b)
   {
       S1.aligned_i=a;
       S1.aligned_s=b;
       return S1.unaligned_i;
   }
   ```

2. Compile this using `armcc -c -Otime foo.c`. The code produced is:

   ```
   MOV      r2,r0
   LDR      r0,|L1.84|
   MOV      r12,r2,LSR #8
   STRB     r2,[r0,#0]
   STRB     r12,[r0,#1]
   MOV      r12,r2,LSR #16
   STRB     r12,[r0,#2]
   MOV      r12,r2,LSR #24
   STRB     r12,[r0,#3]
   MOV      r12,r1,LSR #8
   STRB     r1,[r0,#4]
   STRB     r12,[r0,#5]
   ADD      r0,r0,#6
   BIC      r3,r0,#3
   AND      r0,r0,#3
   LDMIA    r3,{r3,r12}
   MOV      r0,r0,LSL #3
   MOV      r3,r3,LSR r0
   RSB      r0,r0,#0x20
   ORR      r0,r3,r12,LSL r0
   BX       lr
   ```

   However, you can give the compiler more information to enable it to know which fields are aligned and which are not. To do this you must declare non-aligned fields as `__packed`, and remove the `__packed` attribute from the **`struct`** itself. This is the recommended approach, and the only way of guaranteeing fast access to naturally aligned members within the **`struct`**. It is also clearer which fields are non-aligned, but care is needed when adding or deleting fields from the **`struct`**.

3. Now modify the definition of the structure in `foo.c` to:

```
struct mystruct {
    int aligned_i;
    short aligned_s;
    __packed int unaligned_i;
};
struct mystruct S1;
```

4. Compile `foo.c` and the following, more efficient code, is generated:

```
MOV      r2,r0
LDR      r0,|L1.32|
STR      r2,[r0,#0]
STRH     r1,[r0,#4]
LDMIB    r0,{r3,r12}
MOV      r0,r3,LSR #16
ORR      r0,r0,r12,LSL #16
BX       lr
```

The same principle applies to unions. Use the `__packed` attribute on the components of the union that will be unaligned in memory.

Note

Any `__packed` object accessed through a pointer has unknown alignment, even packed structures.

1.2.4 Unaligned LDR for accessing halfwords

In some circumstances the ARM compiler can intentionally generate unaligned `LDR` instructions. In particular, the compiler does this to load halfwords from memory. This is because by using an appropriate address the required halfword can be loaded into the top half of a register and then shifted down to the bottom half. This requires only one memory access whereas doing the same operation using `LDRB` instructions requires two memory accesses, plus instructions to merge the two bytes. On ARM architecture v3 and earlier, this is typically done for any halfword loads. On ARMv4 and later, this is done less often because dedicated halfword load instructions exist, but unaligned `LDR` instructions might still be generated. For instance to access an unaligned **`short`** within a packed structure.

Note

Such unaligned `LDR` instructions are only generated by the RVCT compiler if you enable them using the `--memaccess +L41` compiler option.

1.2.5 Porting code and detecting unaligned accesses

Legacy C code for other architectures (for example, x86 CISC) might perform accesses to unaligned data using pointers that do not work on ARM processors. This is non-portable code, and such accesses must be identified and corrected to work on RISC architectures, which expect aligned data.

Identifying the unaligned accesses can be difficult, because the use of load or store operations with unaligned addresses gives incorrect behavior. It is difficult to trace which part of the C source is causing the problem.

ARM processors with full *Memory Management Units* (MMUs), for example, the ARM920T™, support optional alignment checking, where the processor checks every access to ensure it is correctly aligned. The MMU raises a Data Abort if an incorrectly aligned access occurs.

For simple cores such as the ARM7TDMI®, it is recommended that alignment-checking be implemented within the ASIC/ASSP. You can do this with an additional hardware block that is external to the ARM core, and that monitors the access size and the least significant bits of the address bus for every data access. You can configure the ASIC/ASSP to raise the **ABORT** signal in the case of an unaligned access. ARM Limited recommends that such logic is included on ASIC/ASSP devices where code is ported from other architectures.

If the system is configured to abort on unaligned accesses, a Data Abort exception handler must be installed. When an unaligned access occurs, the Data Abort handler is entered, and this can identify the erroneous data access instruction, which is located at (r14-8).

When identified, you must fix the data access by changing the C source. These changes can be made conditional using the following:

```
#ifdef __arm
  #define  PACKED  __packed
#else
  #define  PACKED
#endif
:
  PACKED int *pi;
:
```

It is best to minimize accesses to unaligned data because of code size and performance overheads.

1.3 Developing for the ARM processors

This book gives information and example code for some of the most common ARM programming tasks. The following sections summarize the subject of each chapter:

- *Embedded Software Development*
- *Using the Procedure call standards* on page 1-11
- *Interworking ARM and Thumb code* on page 1-11
- *Mixing C, C++, and Assembly Language* on page 1-12
- *Handling Processor Exceptions* on page 1-12.

1.3.1 Embedded Software Development

Many applications written for ARM-based systems are embedded applications that are contained in ROM and execute on reset. There are a number of factors that you must consider when writing embedded operating systems, or embedded applications that execute from reset without an operating system, including:

- address remapping, for example initializing with ROM at address `0x0000`, then remapping RAM to address `0x0000`
- initializing the environment and application
- linking an embedded executable image to place code and data in specific locations in memory.

The ARM core usually begins executing instructions from address `0x0000` at reset. For an embedded system, this means that there must be ROM at address `0x0000` when the system is reset. Typically, however, ROM is slow compared to RAM, and often only 8 or 16 bits wide. This affects the speed of exception handling. Having ROM at address `0x0000` means that the exception vectors cannot be modified. A common strategy is to remap ROM to RAM and copy the exception vectors from ROM to RAM after startup. See *ROM/RAM remapping* on page 2-25 for more information.

After reset, an embedded application or operating system must initialize the system, including:

- initializing the execution environment, such as exception vector, stacks, and I/O peripherals
- initializing the application, for example copying initial values of nonzero writable data to the writable data region and zeroing the ZI data region.

See *Initialization sequence* on page 2-23 for more information.

Embedded systems often implement complex memory configurations. For example, an embedded system might use fast, 32-bit RAM for performance-critical code, such as interrupt handlers and the stack, slower 16-bit RAM for application RW data, and ROM for normal application code. You can use the linker scatter-loading mechanism to construct executable images suitable for complex systems. For example, a scatter-load description file can specify the load address and execution address of individual code and data regions. See Chapter 2 *Embedded Software Development* for a series of worked examples, and for information on other issues that affect embedded applications, such as semihosting.

1.3.2 Using the Procedure call standards

The *ARM-Thumb Procedure Call Standard* (ATPCS) defines register usage and stack conventions that must be followed to enable separately compiled and assembled modules to work together. There are a number of variants on the base standard. The ARM compiler always generates code that conforms to the selected ATPCS variant. The linker selects an appropriate standard C or C++ library to link with, if required.

When developing code for the ARM, you must select an appropriate ATPCS variant. For example, if you are writing code that interworks between ARM and Thumb state you must select the `--apcs /interwork` option in the compiler and assembler.

If you are writing code in C or C++, you must ensure that you have selected compatible ATPCS options for each compiled module.

If you are writing your own assembly language routines, you must ensure that you conform to the appropriate ATPCS variant. See Chapter 3 *Using the Procedure Call Standard* for more information.

If you are mixing C and assembly language, ensure that you understand the ATPCS implications.

1.3.3 Interworking ARM and Thumb code

If you are writing code for ARM processors that support the Thumb 16-bit instruction set, you can mix ARM and Thumb code as required. If you are writing C or C++ code you must compile with the `--apcs /interwork` option. The linker detects when an ARM function is called from Thumb state, or a Thumb function is called from ARM state and alters call and return sequences, or inserts interworking veneers to change processor state as necessary.

If you are writing assembly language code you must ensure that you comply with the interworking ATPCS variant. There are several ways to change processor state, depending on the target architecture version. See Chapter 4 *Interworking ARM and Thumb* for more information.

1.3.4 Mixing C, C++, and Assembly Language

You can mix separately compiled and assembled C, C++, and ARM assembly language modules in your program. You can write small assembly language routines within your C or C++ code. These routines are compiled using the inline or embedded assembler of the ARM compiler. However, there are a number of restrictions to the assembly language code you can write if you are using the inline or embedded assembler. These restrictions are described in:

- *Differences between the inline assembler and armasm* on page 5-7
- *Restrictions on embedded assembly* on page 5-13.

In addition, Chapter 5 *Mixing C, C++, and Assembly Language* gives general guidelines and examples of how to call between C, C++, and assembly language modules.

1.3.5 Handling Processor Exceptions

The ARM processor recognizes the following exception types:

Reset Occurs when the processor reset pin is asserted. This exception is only expected to occur for signaling power-up, or for resetting as if the processor has powered up. A soft reset can be done by branching to the reset vector, `0x0000`.

Undefined Instruction

Occurs if neither the processor, or any attached coprocessor, recognizes the currently executing instruction.

Software Interrupt (SWI)

This is a user-defined interrupt instruction. It enables a program running in User mode, for example, to request privileged operations that run in Supervisor mode, such as an RTOS function.

Prefetch Abort

Occurs when the processor attempts to execute an instruction that has been prefetched from an illegal address. An illegal address is one at which memory does not exist, or one that the memory management subsystem has determined is inaccessible to the processor in its current mode.

Data Abort Occurs when a data transfer instruction attempts to load or store data at an illegal address.

Interrupt (IRQ)

Occurs when the processor external interrupt request pin is asserted (LOW) and IRQ interrupts are enabled (the I bit in the CPSR is clear).

Fast Interrupt (FIQ)

Occurs when the processor external fast interrupt request pin is asserted (LOW) and FIQ interrupts are enabled (the F bit in the CPSR is clear). This exception is typically used where interrupt latency must be kept to a minimum.

In general, if you are writing an application such as an embedded application that does not rely on an operating system to service exceptions, you must write handlers for each exception type.

In cases where an exception type can have more than one source, for example SWI or IRQ interrupts, you can chain exception handlers for each source. See *Chaining exception handlers* on page 6-41 for more information.

On Thumb-capable processors, the processor switches to ARM state when an exception is taken. You can either write your exception handler in ARM code, or use a veneer to switch to Thumb state. See *The return address and return instruction* on page 6-9 for more information.

Chapter 2
Embedded Software Development

This chapter describes how to develop embedded applications with RVCT, with or without a target system present. It contains the following sections:

2.1 About embedded software development

Most embedded applications are initially developed in a prototype environment with resources that differ from those available in the final product. Therefore, it is important to consider the processes involved in moving an embedded application from one that relies on the facilities of the development or debugging environment to a system that runs standalone on target hardware.

When developing embedded software using RVCT, you must consider the following:

- How the C library uses hardware.
- Some C library functionality executes by using debug environment resources. If used, you must re-implement this functionality to make use of target hardware.
- RVCT has no inherent knowledge of the memory map of any given target. You must tailor the image memory map to the memory layout of the target hardware.
- An embedded application must perform some initialization before the main application can be run. A complete initialization sequence requires code that you implement as well as RVCT C library initialization routines.

2.1.1 Example code

To illustrate the topics covered in this chapter, associated example projects are provided. The code for the Dhrystone builds described in this chapter is in the directory `*Examples_directory*\emb_sw_dev`. Each build is in a separate directory, and provides an example of the techniques discussed in successive sections of this chapter. Specific information regarding each build can be found in the Example code sections.

The Dhrystone benchmarking program provides the code base for the example projects. Dhrystone was chosen because it enables many of the concepts described in this chapter to be illustrated.

The example projects are tailored to run on the ARM Integrator development platform. However, the principles illustrated by the examples apply to any target hardware.

Note

The focus of this chapter is not specifically the Dhrystone program, but the steps that must be taken to enable it run on a fully standalone system. For further discussion of Dhrystone as a benchmarking tool, see Application Note 93 - *Benchmarking with ARMulator*. You can find the ARM Application Notes in the Documentation area of the ARM website at `http://www.arm.com`.

Running the Dhrystone builds on an Integrator

To run the Dhrystone builds described in this chapter on an Integrator, you must:

- Perform ROM/RAM remapping. To achieve this, run the Boot Monitor by setting switches 1 and 4 on, then reset the board.
- Set `top_of_memory` to `0x40000`, or fit a DIMM memory module. If this is not done, the stack, which defaults to `0x80000`, might not be in valid memory.

  To set top of memory in RealView Debugger, for example, open the Connection Properties for the connection. In the `Advanced_Information\Default\ARM_config` group, set `Top_memory` to the required value, then save the settings.

2.2 Default RVCT behavior in the absence of a target system

When you begin working on software for an embedded application, you might not be aware of the full technical specifications of the target hardware. For example, you might not know the details of target peripheral devices, the memory map, or even the processor itself.

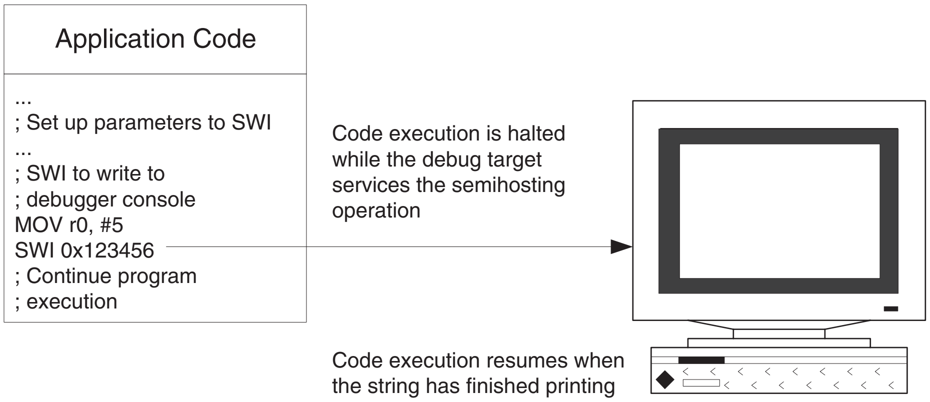
To enable you to proceed with software development before such details are known, the RVCT tools have a default behavior that enables you to start building and debugging application code immediately. It is useful to be aware of this default behavior, so that you appreciate the steps necessary to move from a default build to a fully standalone application.

2.2.1 Semihosting

In the RVCT C Library, support for some ISO C functionality is provided by the host debugging environment. The mechanism that provides this functionality is known as *semihosting*.

Semihosting is implemented by a set of defined software interrupt (SWI) operations. When a semihosting SWI is executed, the debug agent identifies it and briefly suspends program execution. The semihosting operation is then serviced by the debug agent before code execution is resumed. Therefore, the task performed by the host itself is transparent to the program.

Figure 2-1 shows an example of semihosting operation, which prints a string to the debugger console.


Application Code
Code execution is halted while the debug target services the semihosting operation
Code execution resumes when the string has finished printing

```
...
; Set up parameters to SWI
...
; SWI to write to
; debugger console
MOV r0, #5
SWI 0x123456
; Continue program
; execution
```

Figure 2-1 Example semihosting operation

Note

For more information on semihosting, see the chapter on Semihosting in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide*.

2.2.2 C library structure

Conceptually, the C library can be divided into functions that are part of the ISO C Language specification, and functions that provide support to the ISO C language specification. This is illustrated in Figure 2-2.

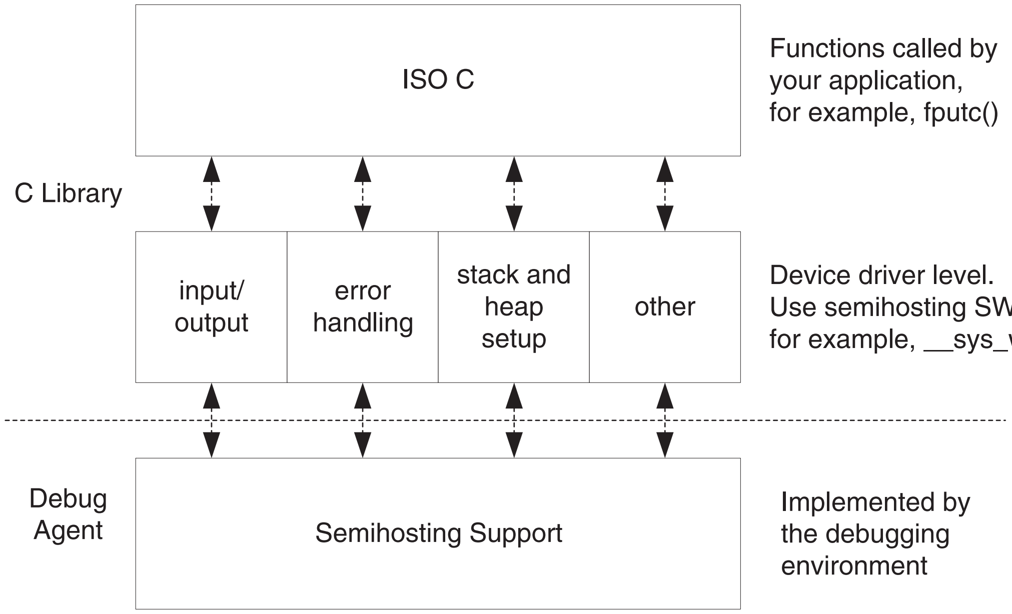

ISO C
Functions called by your application, for example, fputc()
C Library
input/ output
error handling
stack and heap setup
other
Device driver level. Use semihosting SWIs, for example, __sys_write()
Debug Agent
Semihosting Support
Implemented by the debugging environment

Figure 2-2 C library structure

Support for some ISO C functionality is provided by the host debugging environment at the device driver level of support functions.

For example, the RVCT C library implements the ISO C `printf()` family of functions by writing to the debugger console window. This functionality is provided by calling `__sys_write()`. This is a support function that executes a semihosting SWI, that results in a string being written to the console.

2.2.3 Default memory map

In an image where you have not described the memory map, RVCT places code and data according to a default memory map, as shown in Figure 2-3.

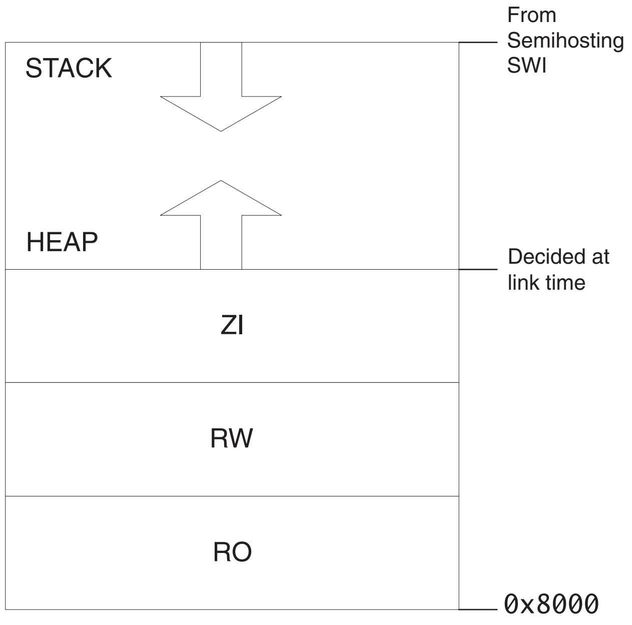


Figure 2-3 Default memory map

The default memory map can be described as follows:

- The image is linked to load and run at address `0x8000`. All RO (Read Only) sections are placed first, followed by RW (Read-Write) sections then ZI (Zero-Initialized) sections.
- The heap follows directly on from the top of the ZI section, so the exact location is decided at link time.
- The stack base location is provided by a semihosting operation during application startup. The value returned by this semihosting operation depends on the debug environment:
- ARMulator returns the value set in the configuration file `peripherals.ami`. The default is `0x08000000`.
- Multi-ICE returns the value of the debugger internal variable `top_of_memory`. The default is `0x00080000`.

2.2.4 Linker placement rules

The linker observes a set of rules, shown in Figure 2-4, to decide where in memory code and data is located.

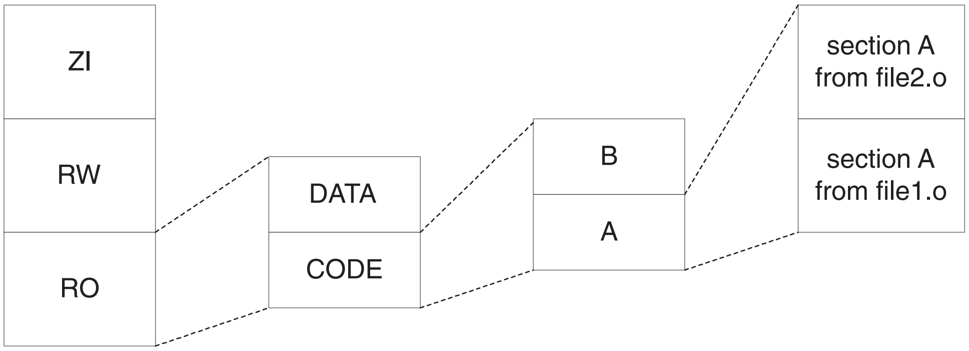


Figure 2-4 Linker placement rules

The image is organized first of all by attribute - RO at the lowest memory address, then RW, then ZI. Within each attribute code precedes the data.

From there, the linker places input sections alphabetically by name. Input section names correspond with assembler `AREA` directives.

In input sections, code and data from individual objects are placed in the order the object files are specified on the linker command line.

ARM Limited does not recommend relying on these rules for precise placement of code and data. Instead, you must use the scatter-loading mechanism for full control of placement of code and data. See *Tailoring the image memory map to your target hardware* on page 2-13.

——— **Note** ———

Refer to the *RealView Compilation Tools v2.0 Linker and Utilities Guide* for more information on placement rules and scatter-loading.

———————————

2.2.5 Application startup

In most embedded systems, an initialization sequence executes to setup the system before the main task is executed.

The default RVCT initialization sequence is shown in Figure 2-5.

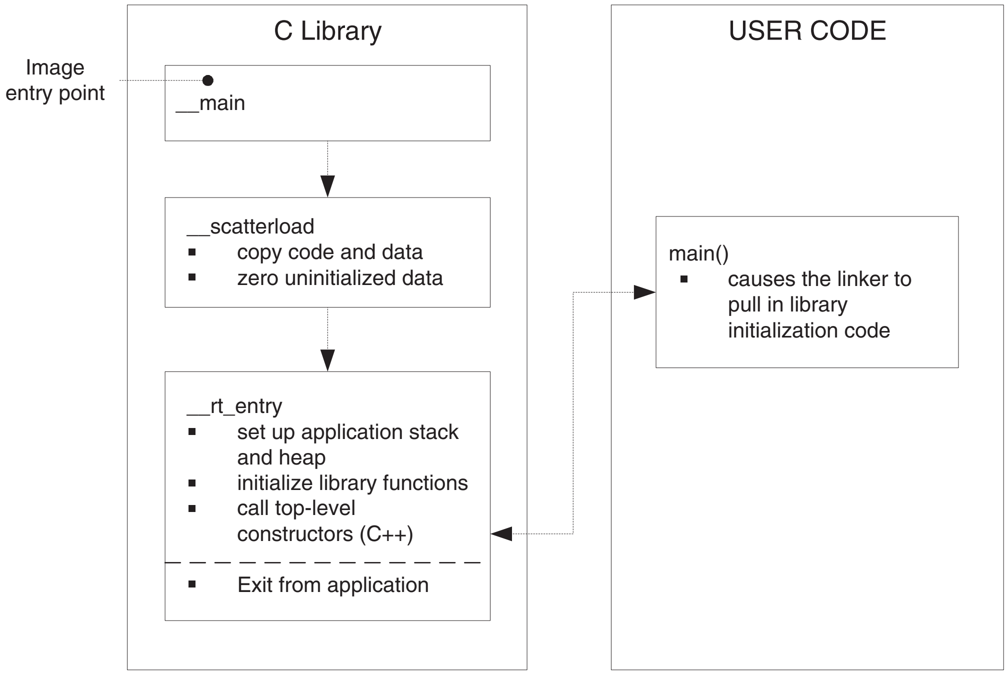

C Library
USER CODE
Image entry point
__main
__scatterload
▪ copy code and data
▪ zero uninitialized data
__rt_entry
▪ set up application stack and heap
▪ initialize library functions
▪ call top-level constructors (C++)
▪ Exit from application
main()
▪ causes the linker to pull in library initialization code

Figure 2-5 Default RVCT initialization sequence

At a high level, the initialization sequence can be divided into three functional blocks. `__main` branches directly to `__scatterload`. `__scatterload` is responsible for setting the run-time image memory map, whereas `__rt_entry` (run-time entry) is responsible for initializing the C library.

`__scatterload` carries out code and data copying, and zeroing of ZI data. This step is always done for zeroing of ZI data and RW data is unchanged.

`__scatterload` branches to `__rt_entry`. This sets up the application stack and heap, initializes library functions and their static data, and calls any constructors of globally declared objects (C++ only).

`__rt_entry` then branches to `main()`, the entry to your application. When the main application has finished executing, `__rt_entry` shuts down the library, then hands control back to the debugger.

The function label `main()` has a special significance in RVCT. The presence of a `main()` function forces the linker to link in the initialization code in `__main` and `__rt_entry`. Without a function labeled `main()` the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

2.2.6 Example code for Build 1

Build 1 is a default build of the Dhrystone Benchmark. Therefore, it adheres to the default RVCT behavior described in this section. See *Running the Dhrystone builds on an Integrator* on page 2-3, and the example build files in `Examples_directory\emb_sw_dev\build1`.

2.3 Tailoring the C library to your target hardware

By default the C library makes use of semihosting to provide device driver level functionality, enabling a host computer to act as an input and an output device. This is useful because development hardware often does not have all the input and output facilities of the final system.

2.3.1 Retargeting the C library

You can provide your own implementation of C Library functions that make use of target hardware, and that are automatically linked in to your image in favor of the C library implementations. This process, known as retargeting the C library, is shown in Figure 2-6.

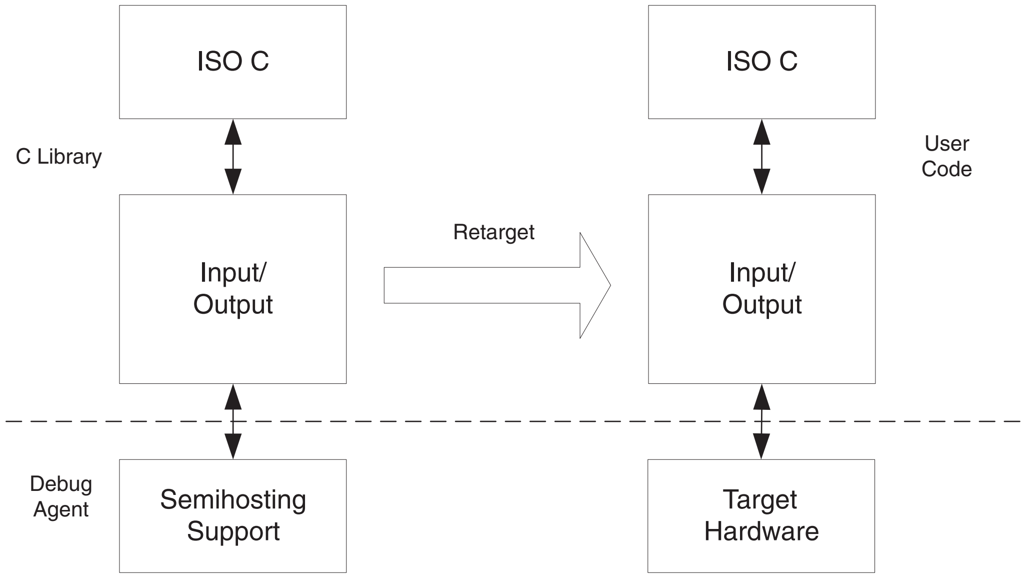


Figure 2-6 Retargeting the C library

For example, you might have a peripheral I/O device such as a UART, and you might want to override the library implementation of `fputc()`, that writes to the debugger console, with one that outputs to the UART. Because this implementation of `fputc()` is linked in to the final image, the entire `printf()` family of functions print out to the UART.

An example implementation of `fputc()` is shown in Example 2-1 on page 2-11.

The example redirects the input character parameter of `fputc()` to a serial output function `sendchar()`, that is assumed to be implemented in a separate source file. In this way, `fputc()` acts as an abstraction layer between target dependent output and the C library standard output functions.

Example 2-1 Implementation of fputc()

```
extern void sendchar(char *ch);

int fputc(int ch, FILE *f)
{   /* e.g. write a character to an UART */
    char tempch = ch;
    sendchar(&tempch);
    return ch;
}
```

2.3.2 Avoiding C library semihosting

In a standalone application, you are unlikely to support semihosting SWI operations. Therefore, you must be certain that no C library semihosting functions are being linked into your application.

To ensure that no functions that use semihosting SWIs are linked in from the C library, you must import the symbol `__use_no_semihosting_swi`. This can be done in any C or assembler source file in your project as follows:

- In a C module, use the `#pragma` directive:

  ```
  #pragma import(__use_no_semihosting_swi)
  ```

- In an assembler module, use the `IMPORT` directive:

  ```
  IMPORT __use_no_semihosting_swi
  ```

If functions that use semihosting SWIs are still being linked in, the linker reports the following error:

```
Error: L6200E: Symbol __semihosting_swi_guard multiply defined (by use_semi.o
and use_no_semi.o).
```

To identify these functions, link using the `-verbose` option. In the resulting output, C library functions are tagged with `__I_use_semihosting_swi`, for example.

```
Loading member sys_exit.o from c_a__un.l.
              definition:  _sys_exit
              reference :  __I_use_semihosting_swi
```

You must provide your own implementations of these functions.

Note

The linker does not report any semihosting SWI-using functions in the your application code. An error only occurs if a semihosting SWI-using function is linked in from the C library.

A full list of C library functions that use semihosting SWIs is available in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide*.

2.3.3 Example code for Build 2

Build 2 of the example uses the hardware of the Integrator platform for clocking and string I/O. See the example build files in `Examples_directory\emb_sw_dev\build2`.

The following changes have been made to Build 1 of the example project:

C Library Retargeting

A retargeted layer of ISO C functions has been added. These include standard input/output functionality, clock functionality, as well as some additional error signaling and program exit.

Target Dependent Device Driver

A device driver layer that interacts directly with target hardware peripherals has been added.

See *Running the Dhrystone builds on an Integrator* on page 2-3.

The symbol `__use_no_semihosting_swi` is not imported into this project. This is because a semihosting SWI is executed during C library initialization to set up the application stack and heap location. Retargeting stack and heap setup is covered in detail in *Placing the stack and heap* on page 2-18.

Note

To see the output, a terminal or terminal emulator (such as Hyperterminal) must be connected to serial port A. The serial port settings must be set to 38400 baud, no parity, 1 stop bit and no flow control. The terminal must be configured to append line feeds to incoming line ends, and echo typed characters locally.

2.4 Tailoring the image memory map to your target hardware

In your final embedded system, without semihosting functionality, you are unlikely to use the default memory map provided by RVCT. Your target hardware usually has several memory devices located at different address ranges. To make the best use of these devices, you must have separate views of memory at load and run-time.

2.4.1 Scatter-loading

Scatter-loading enables you to describe the load-time and run-time location of code and data in memory in a textual description file known as a *scatter-loading description file*. The file is passed to the linker on the command line using the `-scatter` option. For example:

```
armlink -scatter scat.scf file1.o file2.o
```

The scatter-loading description file describes to the linker the desired location of code and data at both load-time and run-time, in terms of addressed memory regions.

Scatter-loading regions

Scatter-loading regions fall into two categories:

- Load Regions that contain application code and data at reset and load time.
- Execution Regions that contain code and data while the application is executing. One or more execution regions are created from each load region during application startup.

All code and data in the image falls into exactly one load region, and one execution region.

During startup, C library initialization code in `__main` carries out the copying and zeroing of code and data necessary to move from the image load view to the execute view.

2.4.2 Scatter-loading description file syntax

The scatter-loading description file syntax reflects the functionality provided by scatter-loading itself. Figure 2-7 shows the file syntax.

A region is defined by a header tag that contains, as a minimum, a name for the region and a start address. Optionally, a maximum length and various attributes can be added.

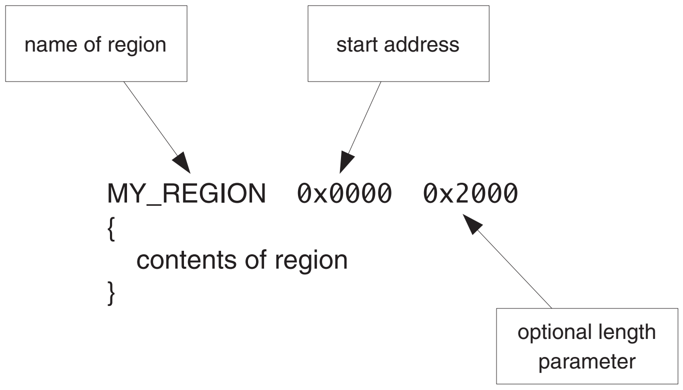


Figure 2-7 Scatter-loading description file syntax

The contents of the region depend on the type of region:

- Load regions must contain at least one execution region. In practice, there are usually several execution regions per load region.
- Execution regions must contain at least one code or data section, unless a region is declared with the `EMPTY` attribute (see *Using the scatter-file EMPTY attribute* on page 2-37). Non-`EMPTY` regions usuallycontain source or library object files. The wildcard (*) syntax can be used to group all sections of a given attribute not specified elsewhere in the scatter-loading description file.

Note

For a more detailed description of scatter-loading description file syntax, see the *RealView Compilation Tools v2.0 Linker and Utilities Guide*.

2.4.3 Scatter-loading description file example

Figure 2-8 illustrates a simple example of scatter-loading.

This example has one load region containing all code and data, starting at address zero. From this load region two execution regions are created. One contains all RO code and data, which executes at the same address at which it is loaded. The other is at address `0x10000`, which contains all RW and ZI data.

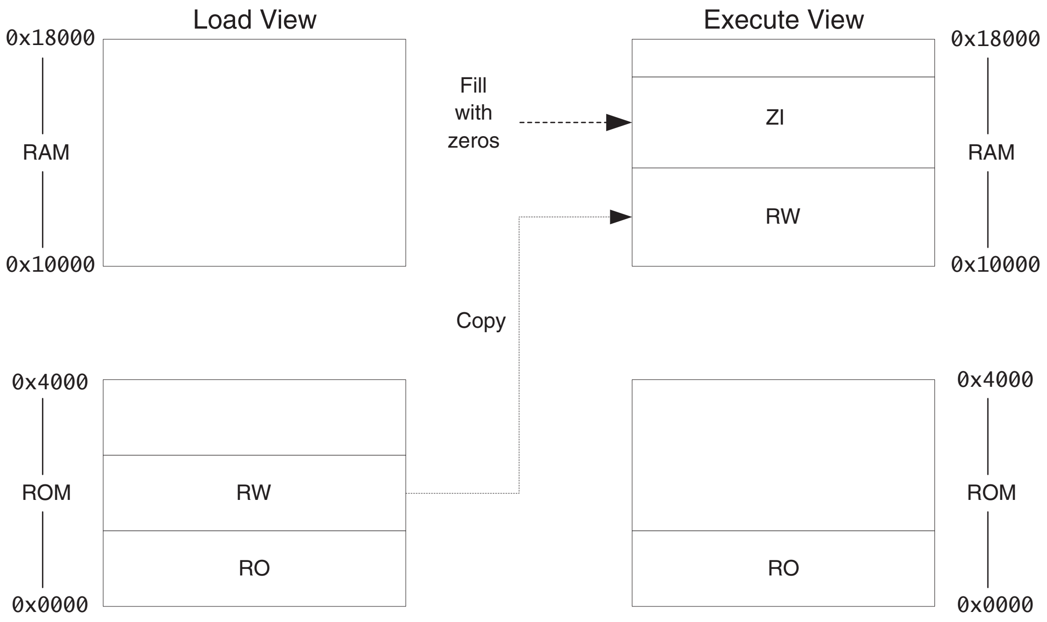


Figure 2-8 Simple scatter-loading example

Example 2-2 shows the description file that describes the above memory map.

Example 2-2 Simple scatter-loading description file

```
ROM_LOAD 0x0000 0x4000
{
    ROM_EXEC 0x0000 0x4000; Root region
    {
        * (+RO); All code and constant data
    }
```

```
     RAM 0x10000 0x8000
     {
          * (+RW, +ZI); All non-constant data
     }
}
```

2.4.4 Placing objects in a scatter-loading description file

For most images, you control the placement of specific code and data sections, rather than grouping all attributes together as in the previous example. You can do this by specifying individual objects directly in the description file, instead of relying only on the wildcard syntax.

Note

The ordering of objects in a description file execution region does not affect their ordering in the output image. The linker placement rules described in *Linker placement rules* on page 2-7 apply to each execution region.

To override the standard linker placement rules, you can use the `+FIRST` and `+LAST` scatter-loading directives. Example 2-3 shows a scatter-loading description file that places the vector table at the beginning of an execution region. In the example, the area `Vect` in `vectors.o` is placed at address `0x0000`. Refer to the *RealView Compilation Tools v2.0 Linker and Utilities Guide* for further information on placing objects in scatter-loading description files.

Example 2-3 Placing a section

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000
  {
    vectors.o (Vect, +FIRST)
    * (+RO)
  }
  ; more exec regions…
}
```

2.4.5 Root regions

A *root region* is an execution region with a load address that is the same as its execution address. Each scatter-loading description file must have at least one root region.

One restriction placed on scatter-loading is that the code and data responsible for creating execution regions, for example, copying and zeroing code and data, cannot itself be copied to another location. As a result, the following sections must be included in a root region:

- `__main.o` that contains the code that copies code and data
- `Region$$Table` and `ZISection$$Table` sections containing the addresses of the code and data to be copied.

Because these sections are attributed as read-only, they are grouped by the `* (+RO)` wildcard syntax. As a result, if `* (+RO)` is specified in a non-root region, these sections must be explicitly declared in a root region. This is shown in Example 2-4.

Example 2-4 Specifying a root region

```
ROM_LOAD 0x0000 0x4000
{
     ROM_EXEC 0x0000 0x4000 ; root region
     {
          __main.o (+RO)      ; copying code
          * (Region$$Table)   ; RO/RW addresses to copy
          * (ZISection$$Table) ; ZI addresses to zero
     }
     RAM 0x10000 0x8000
     {
          * (+RO)             ; all other RO sections
          * (+RW, +ZI)        ; all RW and ZI sections
     }
}
```

Failing to include `__main.o`, `Region$$Table`, and `ZISection$$Table` in a root region results in the linker generating the following error messages:

```
Error: L6202E: Section Region$$Table cannot be assigned to a non-root region.
Error: L6202E: Section ZISection$$Table cannot be assigned to a non-root region.
```

2.4.6 Placing the stack and heap

Scatter-loading provides a method for specifying the placement of code and statically allocated data in your image. This section covers how to place the application stack and heap.

The application stack and heap are setup during C library initialization. You can tailor stack and heap placement by retargeting the routine responsible for stack and heap setup. In the RVCT C library, this routine is `__user_initial_stackheap()`.

Figure 2-9 shows the C library initialization process with a retargeted `__user_initial_stackheap()`.

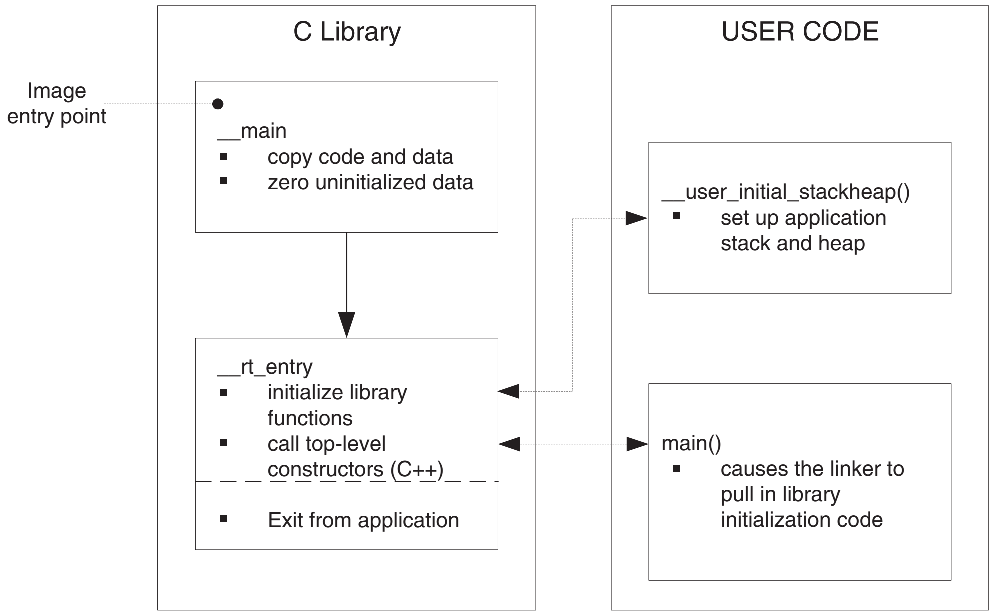

C Library
USER CODE
Image entry point
__main
copy code and data
zero uninitialized data
__rt_entry
initialize library functions
call top-level constructors (C++)
Exit from application
__user_initial_stackheap()
set up application stack and heap
main()
causes the linker to pull in library initialization code

Figure 2-9 Retargeting __user_initial_stackheap()

`__user_initial_stackheap()` can be coded in C or ARM assembler. It must return the following parameters:

- heap base in r0
- stack base in r1
- heap limit in r2, if required
- stack limit in r3, if required.

You must re-implement `__user_initial_stackheap()` if you are scatter-loading your image. Otherwise, the linker generates the following error:

```
Error: L6218E: Undefined symbol Image$$ZI$$Limit (referred from sys_stackheap.o)
```

2.4.7 Run-time memory module

RVCT provides two possible run-time memory models:

- *One-region model*, which is the default
- *Two-region model* on page 2-20.

In both run-time memory models, the stack grows unchecked by default. You can choose to enable software stack checking in your image by compiling all modules with the compiler option `--apcs /swst`. If you are using a two-region model, you must also specify a stack limit in your implementation of `__user_initial_stackheap()`.

Note

Enabling software stack checking introduces a substantial code size and performance overhead, because the value of the stack pointer must be checked against the stack limit with each function call. It also uses register r10, so is not generally recommended for embedded systems.

Both these examples are suitable for the Integrator system.

One-region model

In the default, *one-region model*, the application stack and heap grow towards each other in the same region of memory. In this case, the heap is checked against the value of the stack pointer when new heap space is allocated (that is, when `malloc()` is called).

Example 2-5 and Figure 2-10 on page 2-20 show an example of `__user_initial_stackheap()` implementing a simple one-region model, where the stack grows downwards from address `0x40000`, and the heap grows upwards from `0x20000`. The routine loads the appropriate values into the registers r0 and r1, and then returns. r2 and r3 remain unchanged, because a heap limit and stack limit are not used in a one-region model.

Example 2-5 One-region model routine

```
    EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR r0, =0x20000 ;HB
    LDR r1, =0x40000 ;SB
    ; r2 not used (HL)
    ; r3 not used (SL)
    MOV pc, lr
```

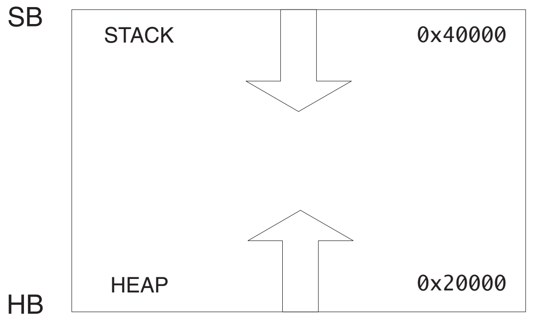


Figure 2-10 One-region model

Two-region model

Your system design might require the stack and heap to be placed in separate regions of memory. For instance you might have a small block of fast RAM that you want to reserve for stack use only. To inform RVCT that you want to use a t*wo-region model*, you must import the symbol `__use_two_region_memory` using the assembler `IMPORT` directive. The heap is then checked against a dedicated heap limit, that is set up by `__user_initial_stackheap()`.

Example 2-6 and Figure 2-11 on page 2-21 show an example of implementing a two-region model. The stack grows downwards from `0x40000` towards a limit of `0x20000`. To make use of this stack limit, all modules using this implementation must be compiled for software stack checking. The heap grows upwards from `0x28000000` to `0x28080000`.

Example 2-6 Two-region model routine

```
    IMPORT __use_two_region_memory
    EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR r0, =0x28000000 ;HB
    LDR r1, =0x40000 ;SB
    LDR r2, =0x28080000 ;HL
    LDR r3, =0x20000 ;SL
    MOV pc, lr
```

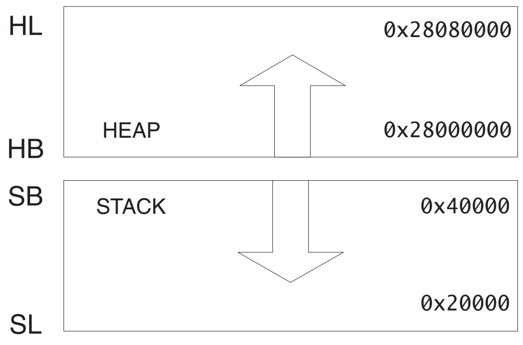


Figure 2-11 Two-region model

2.4.8 Example code for Build 3

Build 3 of the example implements scatter-loading, and contains a retargeted `__user_initial_stackheap()`. See the example build files in *`Examples_directory`*`\emb_sw_dev\build3`.

The following modifications have been made to Build 2 of the example project:

Scatter-loading

A simple scatter-loading description file is passed to the linker.

Retargeted `__user_initial_stackheap()`

You have the option of selecting either a one-region or a two-region implementation. The default build is one-region. You can select the two-region implementation by defining `two_region` at the build step.

Avoiding C library Semihosting

The symbol `__use_no_semihosting_swi` is imported into this build, because there are no longer any C library semihosting functions present in the image.

See *Running the Dhrystone builds on an Integrator* on page 2-3.

Note

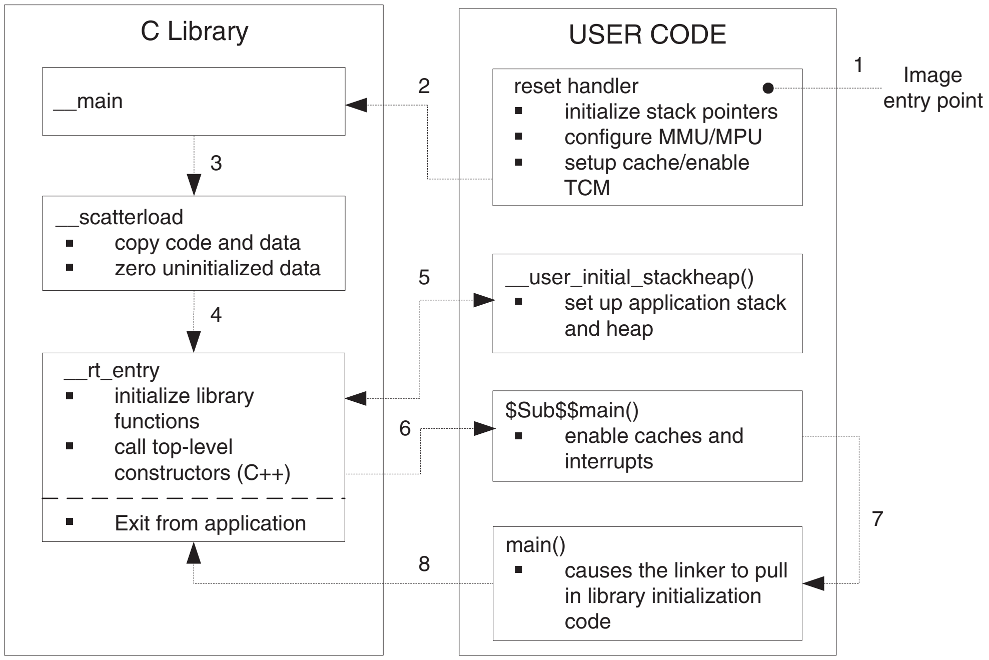
To avoid using semihosting for `clock()`, this is retargeted to read the *Real Time Clock* (RTC) on the Integrator AP. This has a resolution of one second, so the results from Dhrystone are not precise. This mechanism is improved in Build 4.

2.5 Reset and initialization

This chapter has so far assumed that execution begins at `__main`, the entry point to the C library initialization routine. In fact, any embedded application on your target hardware performs some system-level initialization at startup. This section describes this in more detail.

2.5.1 Initialization sequence

Figure 2-12 shows a possible initialization sequence for an ARM-based embedded system.



C Library
__main
__scatterload
▪ copy code and data
▪ zero uninitialized data
__rt_entry
▪ initialize library functions
▪ call top-level constructors (C++)
▪ Exit from application
USER CODE
reset handler
▪ initialize stack pointers
▪ configure MMU/MPU
▪ setup cache/enable TCM
__user_initial_stackheap()
▪ set up application stack and heap
$Sub$$main()
▪ enable caches and interrupts
main()
▪ causes the linker to pull in library initialization code
Image entry point
1
2
3
4
5
6
7
8

Figure 2-12 Initialization sequence

The reset handler in Figure 2-12 executes immediately on system startup. The block of code labeled `$Sub$$main()` executes immediately before entering the main application.

The reset handler is a short module coded in assembler that is executed on system reset. As a minimum, your reset handler initializes stack pointers for the modes that your application is running in. For cores with local memory systems, such as caches, *Tightly Coupled Memories* (TCMs), *Memory Management Units* (MMUs) and *Memory*

Protection Units (MPUs), some configuration must be done at this stage in the initialization process. After executing, the reset handler typically branches to `__main` to begin the C library initialization sequence.

There are some components of system initialization, for example the enabling of interrupts, that are generally performed after the C library initialization code has finished executing. The block of code labeled `$Sub$$main()` performs these tasks immediately before the main application begins executing.

See *The vector table* for a more detailed description of the various components of the initialization sequence.

2.5.2 The vector table

All ARM systems have a *vector table*. The vector table does not form part of the initialization sequence, but it must be present for any exception to be serviced.

The code in Example 2-7 imports the various exception handlers, that might be coded in other modules. The vector table is a list of branch instructions to the various exception handlers.

The FIQ handler is placed at address `0x1C` directly. This avoids having to execute a branch to the FIQ handler, so optimizing FIQ response time.

Example 2-7 Importing exception handlers

```
    AREA Vectors, CODE, READONLY
    IMPORT Reset_Handler
; import other exception handlers
    ; …
        ENTRY
    B   Reset_Handler
    B   Undefined_Handler
    B   SWI_Handler
    B   Prefetch_Handler
    B   Data_Handler
    NOP ; Reserved vector
    B   IRQ_Handler
        ; FIQ_Handler will follow directly
        END
```

Note

The vector table is marked with the label `ENTRY`. This label informs the linker that this code is a possible entry point, and so it cannot be removed from the image at link time. You must select one of the possible image entry points as the true entry point to your application using the `-entry` linker option. Refer to the *RealView Compilation Tools v2.0 Linker and Utilities Guide* for more information.

2.5.3 ROM/RAM remapping

You must consider what sort of memory your system has at address `0x0000`, the address of the first instruction executed.

Note

This section assumes that the ARM core begins fetching instructions at `0x0000`, which is the norm for ARM core based systems. However, some ARM cores can be configured to begin fetching instructions from `0xFFFF0000`.

There has to be a valid instruction at `0x0000` at startup, so you must have non-volatile memory located at `0x0000` at the moment of reset.

One way to achieve this is to have ROM located at `0x0000`. However, there are some drawbacks to this configuration. Access speeds to ROM are generally slower than to RAM, and your system might suffer if there is too great a performance penalty when branching to exception handlers. Also, locating the vector table in ROM does not enable you to modify it at run time.

Another solution is shown in Figure 2-13 on page 2-26. ROM is located at address `0x10000`, but this memory is aliased to zero by the memory controller at reset. Following reset, code in the reset handler branches to the real address of ROM. The memory controller then removes the aliased ROM, so that RAM is shown at address `0x0000`. In `__main`, the vector table is copied into RAM at `0x0000`, so that exceptions can be serviced.

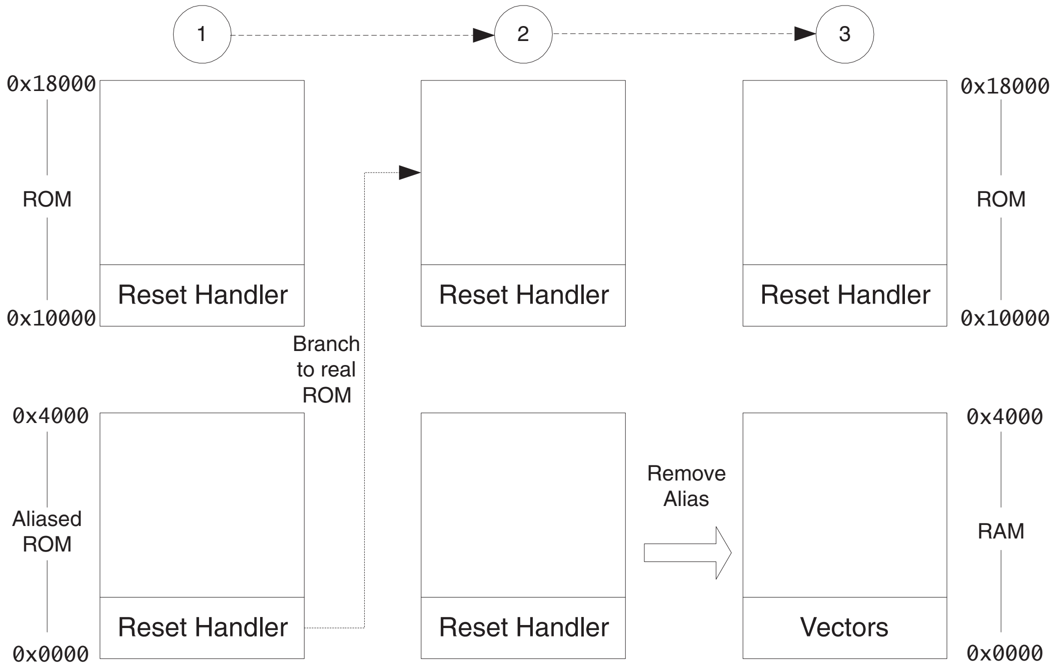


Figure 2-13 ROM/RAM remap

Example 2-8 on page 2-27 shows how you might implement ROM/RAM remapping in an ARM assembler module. The constants shown here are specific to the Integrator platform, but the same method is applicable to any platform that implements ROM/RAM remapping in a similar way.

The first instruction is a jump from aliased ROM to real ROM. This can be done because the label `Instruct_2` is located at the real ROM address.

After this step, the alias of ROM is removed by inverting the remap bit of the Integrator Core Module control register.

This code is normally executed immediately after system reset. Remapping must be completed before C library initialization code can be executed.

Note

In systems with MMUs, remapping can be implemented through MMU configuration at system startup.

Example 2-8 ROM/RAM remapping

```
; --- Integrator CM control reg
CM_ctl_reg     EQU  0x1000000C     ; Address of CM Control Register
Remap_bit      EQU  0x04           ; Bit 2 is remap bit of CM_ctl

    ENTRY

; Code execution starts here on reset
; On reset, an alias of ROM is at 0x0, so jump to 'real' ROM.
        LDR     pc, =Instruct_2

Instruct_2
; Remap by setting Remap bit of the CM_ctl register
        LDR     r1, =CM_ctl_reg
        LDR     r0, [r1]
        ORR     r0, r0, #Remap_bit
        STR     r0, [r1]

; RAM is now at 0x0.
; The exception vectors must be copied from ROM to RAM (in __main)

; Reset_Handler follows on from here
```

2.5.4 Local memory setup considerations

Many ARM processor cores have on-chip memory systems, such as MMUs or MPUs. These devices are normally setup and enabled during system startup. Therefore, the initialization sequence of cores with local memory systems requires special consideration.

As described in this chapter, C library initialization code in `__main` is responsible for setting up the execution time memory map of the image. Therefore, the run-time memory view of the processor core must be set up before branching to `__main`. This means that any MMU or MPU must be set up and enabled in the reset handler.

TCMs must also be enabled before branching to `__main` (normally before MMU/MPU setup), because you generally want to scatter-load code and data into TCMs. You must be careful that you do not have to access memory that is masked by the TCMs when they are enabled.

You also risk problems with cache coherency if caches are enabled before branching to `__main`. Code in `__main` copies code regions from their load address to their execution address, essentially treating instructions as data. As a result, some instructions can be cached in the data cache, in which case they are not visible to the instruction path.

To avoid these coherency problems, enable caches after the C library initialization sequence finishes executing.

2.5.5 Scatter-loading and memory setup

In a system where the reset-time memory view of the core is altered, either through ROM/RAM remapping or MMU configuration, the scatter-loading description file must describe the image memory map after remapping has taken place.

The description file in Example 2-9 relates to the example in *ROM/RAM remapping* on page 2-25 after remapping.

Example 2-9

```
ROM_LOAD 0x10000 0x8000
{
     ROM_EXEC 0x10000 0x8000
     {
          reset_handler.o (+RO, +FIRST)   ; executed on hard reset
          …
     }

     RAM 0x0000 0x4000
     {
          vectors.o (+RO, +FIRST)  ; vector table copied from ROM to RAM at zero
          …
     }
}
```

The load region `ROM_LOAD` is placed at `0x10000`, because this indicates the load address of code and data after remapping has occurred.

2.5.6 Stack pointer initialization

As a minimum, your reset handler must assign initial values to the stack pointers of any execution modes that are used by your application.

In Example 2-10, the stacks are located at `stack_base`. This symbol can be a hard coded address, or it can be defined in a separate assembler source file and located by a scatter-loading description file. Details of how this is done are given in *Placing the stack and heap in the scatter-loading description file* on page 2-34.

The example allocates 256 bytes of stack for FIQ and IRQ mode, but you can do the same for any other execution mode. To set up the stack pointers, enter each mode (interrupts disabled) and assign the appropriate value to the stack pointer. To make use of software stack checking, you also have to set up a stack limit here.

Stack pointer and stack limit values set up in the reset handler are automatically passed as parameters to `__user_initial_stackheap()` by C library initialization code. Therefore, these values must not be modified by `__user_initial_stackheap()`.

Example 2-10 Initializing stack pointers

```
; --- Amount of memory (in bytes) allocated for stacks
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
…

Offset_FIQ_Stack         EQU     0
Offset_IRQ_Stack         EQU     Offset_FIQ_Stack + Len_FIQ_Stack
…

Reset_Handler

; stack_base could be defined above, or located in a description file
        LDR     r0, stack_base ;

; Enter each mode in turn and set up the stack pointer
        MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
        SUB     sp, r0, #Offset_FIQ_Stack

        MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
        SUB     sp, r0, #Offset_IRQ_Stack
        …
; Set up stack limit if needed
        LDR     r10, stack_limit
```

Example 2-11 shows an implementation of `__user_initial_stackheap()` that you can use with the stack pointer setup shown in Example 2-10 on page 2-29.

Example 2-11

```
    IMPORT heap_base
    EXPORT __user_initial_stackheap()

__user_initial_stackheap()

; heap base could be hard coded, or placed by description file
    LDR   r0,=heap_base
    ; r1 contains SB value
    MOV   pc,lr
```

2.5.7 Hardware initialization

In general, it is beneficial to separate all system initialization code from the main application. However, some components of system initialization, for example enabling of caches and interrupts, must occur after executing C library initialization code.

You can make use of the `$Sub` and `$Super` function wrapper symbols to effectively insert a routine that is executed immediately before entering the main application. This mechanism enables you to extend functions without altering the source code.

Example 2-12 shows how `$Sub` and `$Super` can be used in this way. The linker replaces the function call to `main()` with a call to `$Sub$$main()`. From there you can call a routine that enables caches, and another to enable interrupts.

The code branches to the real `main()` by calling `$Super$$main()`.

Note

More information on $Sub and $Super is in the *RealView Compilation Tools v2.0 Linker and Utilities Guide*.

Example 2-12 Use of $Sub and $Super

```
extern void $Super$$main(void);

void $Sub$$main(void)
{
  cache_enable();           // enables caches
  int_enable();      // enables interrupts
  $Super$$main();            // calls original main()
}
```

2.5.8 Execution mode considerations

You must consider in what mode the main application is to run. Your choice affects how you implement system initialization.

Much of the functionality that you are likely to implement at startup, both in the reset handler and `$Sub$$main`, can only be done while executing in privileged modes. For example, on-chip memory manipulation, and enabling interrupts.

If you want to run your application in a privileged mode (for example, Supervisor), this is not an issue. Be sure to change to the appropriate mode before exiting your reset handler.

If you want to run your application in User mode, you can only change to User mode after completing the necessary tasks in a privileged mode. The most likely place to do this is in `Sub$$main()`.

Note

`__user_initial_stackheap()` must set up the application mode stack. Because of this, you must exit your reset handler in system mode, which uses the User mode registers. `__user_initial_stackheap()` then executes in system mode, and so the application stack and heap are still set up when User mode is entered.

2.5.9 Example code for Build 4

Build 4 of the example can be run standalone on the Integrator platform. See the example build files in *`Examples_directory`*`\emb_sw_dev\build4`.

The following modifications have been made to Build 3 of the example project:

Vector table

A vector table has been added to the project, and placed by the scatter-loading description file.

Reset handler

The reset handler is added in `init.s`. Two separate modules, responsible for TCM and MMU setup respectively, are included in the ARM926EJ-S build. These are excluded from the ARM7TDMI build, that runs on Integrator systems with any core. ROM/RAM remapping occurs immediately after reset.

`$Sub$$main()`

For the ARM926EJ-S build, Caches are enabled in `$Sub$$main()` before entering the main application.

Embedded description file

An embedded description file is used, which reflects the post-remapping view of memory.

The build files for both of these builds produce a binary file suitable for downloading into the Integrator AP application Flash at address `0x24000000`.

A precise timer is implemented using a timer on the AP motherboard. This generates an IRQ, and a handler is installed which increments a counter every 1/100 second.

2.6 Further memory map considerations

The previous sections in this chapter describe the placement of code and data in a scatter-loading description file, but the location of target hardware peripherals and the stack and heap limits are assumed to be hard coded in source or header files. It would be beneficial to locate all information pertaining to the memory map of a target in your description file, so removing all references to absolute addresses from your source code.

2.6.1 Locating target peripherals in the scatter-loading description file

Conventionally, addresses of peripheral registers are hard-coded in project source or header files. You can also declare structures that map on to peripheral registers, and place these structures in the description file.

For example, a target could have a timer peripheral with two memory mapped 32-bit registers. Example 2-13 shows a C structure that maps to these registers.

Example 2-13 Mapping to a peripheral register

```
struct {
            volatile unsigned ctrl; /* timer control */
            volatile unsigned tmr;  /* timer value   */
        } timer_regs;
```

To place this structure at a specific address in the memory map, create a new execution region to hold the structure.

The description file shown in Example 2-14 locates the `timer_regs` structure at `0x40000000`.

It is important that the contents of these registers are not initialized to zero during application startup, because this is likely to alter the state of your system. Marking an execution region with the `UNINIT` attribute prevents ZI data in that region from being zero-initialized.

Example 2-14 Placing the mapped structure

```
ROM_LOAD 0x24000000 0x04000000
{
 ; …
 TIMER 0x40000000 UNINIT
 {
     timer_regs.o (+ZI)
 }
 ; …
}
```

2.6.2 Placing the stack and heap in the scatter-loading description file

In many cases, it is preferable to specify the location of the stack and heap in the description file. This has two main advantages:

- all information about the memory map is kept in one file
- changes to the stack and heap only require relinking, not recompiling.

This section describes the following methods for implementing this:

- *Placing symbols explicitly*, which is the simplest of the two methods
- *Utilizing linker generated symbols* on page 2-35
- *Using the scatter-file EMPTY attribute* on page 2-37.

Placing symbols explicitly

Stack pointer initialization on page 2-29 refers to the symbols `stack_base` and `heap_base` as reference symbols that can be placed in a description file. To do this, create symbols labeled `stack_base` and `heap_base` in an assembler module called `stackheap.s`. The same can be done for the stack and heap limits in a two-region memory model.

You can locate each of the symbols within their own execution region in the description file, as shown in Example 2-15.

Example 2-15 Placing symbols explicitly in stackheap.s

```
        AREA   stacks, DATA, NOINIT
        EXPORT stack_base

stack_base        SPACE   1

        AREA   heap, DATA, NOINIT
        EXPORT heap_base

heap_base         SPACE   1
        END
```

Example 2-16 on page 2-35 and Figure 2-14 on page 2-35 shows how you can place the heap base at `0x20000` and the stack base at `0x40000`. The stack and heap base locations can be altered by editing the addresses of the respective execution regions.

The disadvantage of this approach is that one word of `SPACE` (`stack_base`) is occupied above the stack region.

Example 2-16 Placing symbols explicitly in a scatter file

```
LOAD_FLASH 0x24000000 0x04000000
{
 ; …
    HEAP 0x20000 UNINIT
    {
        stackheap.o (heap)
    }

    STACKS 0x40000 UNINIT
    {
        stackheap.o (stacks)
    }
 ; …
}
```

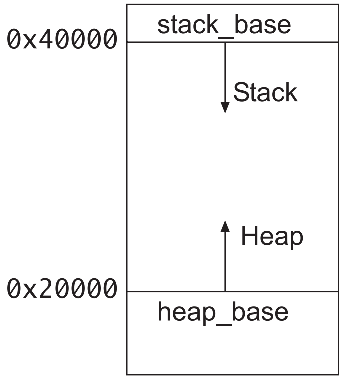


Figure 2-14 Placing symbols explicitly

Utilizing linker generated symbols

This method requires that the stack and heap size are specified in an object file.

First, define areas of an appropriate size for the stack and heap in an assembler source file, for example, `stackheap.s`, as shown in Example 2-17 on page 2-36. Use the `SPACE` directive to reserve a zeroed block of memory. Set the `NOINIT` area attribute to prevent this zeroing. During development, you might choose to zero-initialize the stack so that the maximum stack usage can be seen. Labels are not required in this source file.

Example 2-17 Placing sections for stack and heap

```
        AREA stack, DATA, NOINIT
    SPACE   0x3000 ; Reserve stack space

        AREA heap, DATA, NOINIT
    SPACE   0x3000 ; Reserve heap space

        END
```

You can then place these sections in their own execution region in the scatter-loading description file, as shown in Example 2-18.

Example 2-18 Placing sections for stack and heap

```
LOAD_FLASH 0x24000000 0x04000000
{
    :
    STACK 0x1000 UNINIT   ; length = 0x3000
    {
        stackheap.o (stack)  ; stack = 0x4000 to 0x1000
    }

    HEAP 0x15000 UNINIT   ; length = 0x3000
    {
        stackheap.o (heap)  ; heap = 0x15000 to 0x18000
    }
}
```

The linker generates symbols that point to the base and limit of each execution region, which can be imported into the retargeting code to be used by `__user_initial_stackheap()`:

```
Image$$STACK$$ZI$$Limit = 0x4000
Image$$STACK$$ZI$$Base  = 0x1000
Image$$HEAP$$ZI$$Base   = 0x15000
Image$$HEAP$$ZI$$Limit  = 0x18000
```

You can make this code more readable by using the `DCD` directive to give these values more meaningful names, as shown in Example 2-19 on page 2-37.

Example 2-19 Using the DCD directive

```
        IMPORT      ||Image$$STACK$$ZI$$Base||
        IMPORT      ||Image$$STACK$$ZI$$Limit||
        IMPORT      ||Image$$HEAP$$ZI$$Base||
        IMPORT      ||Image$$HEAP$$ZI$$Limit||

    stack_base  DCD     ||Image$$STACK$$ZI$$Limit||       ; = 0x4000
    stack_limit DCD     ||Image$$STACK$$ZI$$Base||        ; = 0x1000

    heap_base   DCD     ||Image$$HEAP$$ZI$$Base||         ; = 0x15000
    heap_limit  DCD     ||Image$$HEAP$$ZI$$Limit||        ; = 0x18000
```

You can use these examples to place the heap base at `0x15000` and the stack base at `0x1000`. You can then alter the stack and heap base locations easily by editing the addresses of the respective execution regions.

Using the scatter-file EMPTY attribute

This method uses the scatter-file `EMPTY` attribute of the linker. This enables regions to be defined that contain no object code or data. This is a convenient method of defining a stack or heap. The length of the region is specified after the `EMPTY` attribute. For the case of a heap, which grows upwards in memory, the region length is positive. For the case of a stack, the region length is marked as negative, to indicate that it grows downwards in memory. Example 2-20 shows how to use the `EMPTY` attribute.

The benefit of this approach is that the size and position of the stack and heap is defined in one place, that is, in the scatter file. You do not have to create a `stackheap.s` file.

Example 2-20 Placing stack and heap regions using EMPTY

```
ROM_LOAD 0x24000000 0x04000000
{
:
     HEAP 0x30000 EMPTY 0x3000
     {
     }

     STACKS 0x40000 EMPTY -0x3000
     {
     }
:
}
```

At link time, the linker generates symbols to represent these EMPTY regions:

```
Image$$HEAP$$ZI$$Base      = 0x30000
Image$$HEAP$$ZI$$Limit     = 0x33000
Image$$STACKS$$ZI$$Base    = 0x3D000
Image$$STACKS$$ZI$$Limit   = 0x40000
```

Your application code can then process these symbols as shown in Example 2-21.

Example 2-21 Linker generated symbols representing EMPTY regions

```
                IMPORT      ||Image$$HEAP$$ZI$$Base||
                IMPORT      ||Image$$HEAP$$ZI$$Limit||

heap_base       DCD         ||Image$$HEAP$$ZI$$Base||
heap_limit      DCD         ||Image$$HEAP$$ZI$$Limit||

                IMPORT      ||Image$$STACKS$$ZI$$Base||
                IMPORT      ||Image$$STACKS$$ZI$$Limit||

stack_base      DCD         ||Image$$STACKS$$ZI$$Limit||
stack_limit     DCD         ||Image$$STACKS$$ZI$$Base||
```

2.6.3 Example code for Build 5

Build 5 of the example is equivalent to Build 4, but with all target memory map information located in the scatter-loading description file as described in *Placing symbols explicitly* on page 2-34:

Scatter-loading description file symbols

Symbols to locate the stack, heap, and peripherals are declared in assembler modules.

Updated Scatter-loading description file

The embedded description file from Build 4 is updated to locate the stack, heap, data TCM, and peripherals.

See the example build files in `*Examples_directory*\emb_sw_dev\build5`.

The stack and heap are located using linker symbols, see *Utilizing linker generated symbols* on page 2-35.

Chapter 3
Using the Procedure Call Standard

This chapter describes how to use the *ARM-Thumb Procedure Call Standard* (ATPCS). Adhere to the ATPCS to ensure that separately compiled and assembled modules can work together. The chapter contains the following sections:

3.1 About the ARM-Thumb Procedure Call Standard

Adherence to the *ARM-Thumb Procedure Call Standard* (ATPCS) ensures that separately compiled or assembled subroutines can work together. This chapter describes how to use the ATPCS.

ATPCS has several variants. This chapter gives information enabling you to choose which variant to use.

Many details of the standard are the same, whichever variant you use. See:

- *Register roles and names* on page 3-4
- *The stack* on page 3-6
- *Parameter passing* on page 3-9.

3.1.1 ATPCS variants

The variants comprise a base standard modified by options that you can select independently. Code conforming to the base standard runs faster than, and occupies less memory than, code conforming to other variants. However, code conforming to the base standard does not provide for:

- interworking between ARM state and Thumb state
- position independence of either data or code
- re-entry to routines with independent data for each invocation
- stack checking.

The compiler or assembler sets *attributes* in the ELF object file which record the variant you have chosen. In general, you must choose one variant and then use it for all subroutines that must work together. Exceptions to this rule are described in the text.

The options are dealt with under the following headings:

- *Stack limit checking* on page 3-11
- *Read-only position independence* on page 3-14
- *Read-write position independence* on page 3-15
- *Interworking between ARM and Thumb states* on page 3-16
- *Floating-point options* on page 3-17.

3.1.2 ARM C libraries

There are several variants of the ARM C libraries (see *RealView Compilation Tools v2.0 Compiler and Libraries Guide*). The linker selects a variant to link with your object files. It selects the best variant compatible with the ATPCS options recorded in your object files. See the linker chapter in *RealView Compilation Tools v2.0 Linker and Utilities Guide*.

3.1.3 Conformance to the ATPCS

`extern` routines compiled using the ARM compiler conform to the selected variant of the ATPCS.

You are responsible for ensuring that routines written in assembly language conform to the selected variant of the ATPCS.

To conform to the ATPCS, an assembly language routine must:

- follow all details of the standard at publicly visible interfaces
- follow the ATPCS rules of stack usage at all times
- be assembled with the appropriate `--apcs` options selected.

3.1.4 Processes and the memory model

ATPCS applies to a single *thread of execution* or *process*. The *memory state* of a process is defined by the contents of the processor registers and contents of the memory that it can address.

A process can address some or all of the following types of memory:

- Read-only memory.
- Statically-allocated read-write memory.
- Dynamically-allocated read-write memory. This is called *heap* memory.
- Stack memory. See *The stack* on page 3-6.

A process must not alter the memory state of another process unless the two processes are specifically designed to cooperate.

3.2 Register roles and names

The ATPCS specifies the registers to use for particular purposes.

3.2.1 Register roles

The following register usage applies in all variants of the ATPCS except where otherwise stated. To comply with the ATPCS you must follow these rules:

- Use registers r0-r3 to pass parameter values into routines, and to pass result values out. You can refer to r0-r3 as a1-a4 to make this usage apparent. See *Parameter passing* on page 3-9. Between subroutine calls you can use r0-r3 for any purpose. A called routine does not have to restore r0-r3 before returning. A calling routine must preserve the contents of r0-r3 if it requires them again.
- Use registers r4-r11 to hold the values of a routine's local variables. You can refer to them as v1-v8 to make this usage apparent. In Thumb state, in most instructions you can only use registers r4-r7 for local variables.

  A called routine must restore the values of these registers before returning, if it has used them.
- Register r12 is the intra-call scratch register, ip. It is used in this role in procedure linkage veneers, for example interworking veneers. Between procedure calls you can use it for any purpose. A called routine does not have to restore r12 before returning.
- Register r13 is the stack pointer, sp. You must not use it for any other purpose. The value held in sp on exit from a called routine must be the same as it was on entry.
- Register r14 is the link register, lr. If you save the return address, you can use r14 for other purposes between calls.
- Register r15 is the program counter, PC. It cannot be used for any other purpose.

3.2.2 Register names

Table 3-1 lists the defined roles of the processor registers, and associated names. These names and their synonyms are predefined in the assembler. The compiler uses the special names and the basic register names when generating assembler language.

Table 3-1 Register roles and names in ATPCS

Register	Synonym	Special	Role in the procedure call standard
r15	-	PC	Program counter.
r14	-	lr	Link register.
r13	-	sp	Stack pointer.
r12	-	ip	Intra-procedure-call scratch register.
r11	v8	-	ARM-state variable register 8.
r10	v7	sl	ARM-state variable register 7. Stack limit pointer in stack-checked variants.
r9	v6	sb	ARM-state variable register 6. Static base in RWPI variants.
r8	v5	-	ARM-state variable register 5.
r7	v4	-	Variable register 4.
r6	v3	-	Variable register 3.
r5	v2	-	Variable register 2.
r4	v1	-	Variable register 1.
r3	a4	-	Argument/result/scratch register 4.
r2	a3	-	Argument/result/scratch register 3.
r1	a2	-	Argument/result/scratch register 2.
r0	a1	-	Argument/result/scratch register 1.

In addition, s0-s31, d0-d15, and f0-f31 are predefined names for registers in floating-point coprocessors. See *The VFP architecture* on page 3-18 and *The FPA architecture* on page 3-20 for more information.

3.3 The stack

This section describes how to use the stack in the base standard. See also *Stack limit checking* on page 3-11.

ATPCS specifies:

- a full, descending stack
- eight-byte stack alignment at all external interfaces.

3.3.1 Stack terminology

The following stack-related terms are used in ATPCS:

The *stack pointer*

Addresses the last value written to the stack (pushed).

The *stack base*

The address of the top of the stack, from which the stack grows downwards. The highest location actually used by the stack is the first word *below* the stack base.

The *stack limit*

The lowest address on the stack that the current process is permitted to use.

The *used stack*

The region of memory between the stack base and the stack pointer. It includes the stack pointer but not the stack base.

The *unused stack*

The region of memory between the stack pointer and the stack limit. It includes the stack limit but not the stack pointer.

Stack frames Regions of memory allocated on the stack by routines for saving registers and holding local variables.

A process might, or might not, have access to the current values of the stack base and stack limit.

An interrupt handler can use the stack of the process it interrupts. In this case, it is the responsibility of the programmer to ensure that stack limits are not exceeded.

Figure 3-1 on page 3-7 shows the stack memory layout.

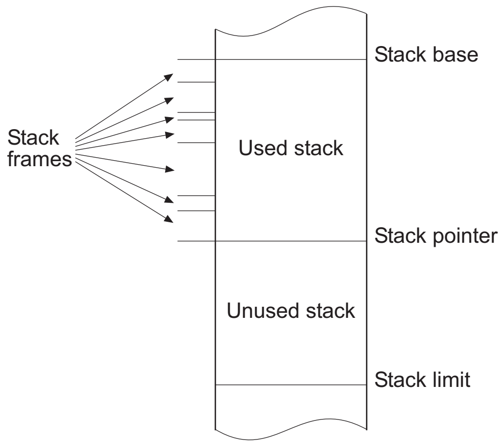


Figure 3-1 Stack memory layout

3.3.2 Stack unwinding

Compiled object files always contain DWARF2 debug frame information. The debuggers use this information to unwind the stack when necessary during debug. This enables you to view the stack backtrace in a debugger

In assembly language, it is your responsibility to describe your stack frames using `FRAME` directives. The assembler uses these to generate DWARF2 debug frame information when assembled with `-g`. See the *Writing ARM and Thumb Assembly Language*, and *Directives Reference* chapters in *RealView Compilation Tools v2.0 Assembler Guide*.

3.3.3 Eight-byte alignment

For multiple transfers on some systems, eight-byte alignment of addresses can improve memory access speed. For `LDRD` and `STRD` instructions on ARMv5TE and later processors, eight-byte alignment is required.

Compiler-generated object files preserve eight-byte alignment of the stack at all external interfaces. The compiler sets a build attribute to indicate this to the linker.

To comply with the ATPCS in assembly language, unless your object file contains no external calls, you must:

- Ensure that eight-byte alignment of the stack is preserved at all external interfaces. (The stack pointer must always move by an even number of words between entry to your code and any external call from your code.)
- Use the `PRESERVE8` directive to inform the linker that eight-byte alignment is preserved (see the *Directives Reference* chapter in *RealView Compilation Tools v2.0 Assembler Guide*).

3.4 Parameter passing

A routine with a variable number of arguments is *variadic*. A routine with a fixed number of arguments is *nonvariadic*. There are different rules about passing parameters to variadic and to nonvariadic routines.

This section describes the base standard. For additional information relating to floating-point options, see *Floating-point options* on page 3-17.

3.4.1 Nonvariadic routines

Parameter values are passed to a nonvariadic routine in the following way:

1. The first integer arguments are allocated to r0-r3 in order (but see *Allocation of long integers*).
2. Remaining parameters are allocated to the stack in order (but see *Allocation of long integers*).

   Warning

   Stack accesses are costly in code size and execution speed. Keep the number of parameters less than five if possible.

Allocation of long integers

An integer parameter longer than 32 bits, for example a `long long`, has eight-byte alignment. When passing a `long long`, it is allocated to either registers r2 and r3, or to the stack.

Allocation of floating-point numbers

If your system has floating point hardware, FP parameters are allocated to FP registers as follows:

1. Each FP parameter is examined in turn.
2. For each parameter, the available set of FP registers is examined.
3. If one is available, the lowest-numbered, contiguous set of FP registers large enough for the parameter is allocated to the parameter.

3.4.2 Variadic routines

Parameter values are passed to a variadic routine in integer registers a1-a4, and on the stack if necessary (a1-a4 are synonyms for r0-r3).

The order of the words used is as if the parameter values were stored in consecutive memory words and then transferred to:

1. a1-a4, a1 first.
2. The stack, lowest address first. (This means that they are pushed onto the stack in reverse order.)

Note

As a consequence, a floating-point value might be passed in integer registers, on the stack, or split between integer registers and the stack.

3.4.3 Result return

A function can return:

- A one-word integer value in a1.
- A two to four-word integer value in a1-a2, a1-a3 or a1-a4.
- A floating-point value in f0, d0, or s0.
- A compound floating-point value (such as `complex`) in f0-f*N*, or d0-d*N*. The maximum value of *N* depends on the selected floating-point architecture (see *Floating-point options* on page 3-17).
- A longer value must be returned indirectly, in memory.

3.5 Stack limit checking

Select the *software stack limit checking* (`-apcs /swst`) option unless the maximum amount of stack memory required by your complete program can be accurately calculated at the design stage.

Select the *no software stack limit checking* (`-apcs /noswst`) option only if you can accurately calculate, at the design stage, the maximum amount of stack memory that your complete program requires. This is the default.

It is possible to write assembly code in such a way that stack limit checking is irrelevant. The code in a file might not require stack limit checking, but still be compatible with other code assembled with either `-apcs /swst` or the default `-apcs /noswst`. Use the *software stack limit checking not applicable* (`-apcs /swstna`) option in this case.

3.5.1 Rules for stack limit checked code

In the stack limit checked variants of the ATPCS:

- The *stack limit pointer* (sl) must point at least 256 bytes above the lowest usable address in the stack.

  Note

  If an interrupt handler can use the User mode stack, you must allocate sufficient space for it, between sl and the lowest usable address in the stack, in addition to the 256 bytes.

- sl must not be altered by code compiled or assembled with stack limit checking selected. sl is altered by run-time support code.
- The value held in the *stack pointer* (sp) must always be greater than or equal to the value in sl.

3.5.2 Register usage with stack limit checking

You must not alter r10, or restore it, in routines assembled or compiled with the stack checking option selected. Register r10 is the stack limit pointer, sl.

In all other respects the usage of registers is the same with or without stack limit checking (see *Register roles and names* on page 3-4).

3.5.3 Stack checking in C and C++

If you select the software stack limit checking (`-apcs /swst`) option, the compiler generates object code that performs stack checking.

3.5.4 Stack checking in assembly language

If you select the software stack checking (`/swst`) option for your assembly code, it is your responsibility to write code that performs stack checking.

A *leaf routine* is a routine that does not call any other subroutine.

The following cases have to be considered:

- *Leaf routine using less than 256 bytes of stack*
- *Nonleaf routine using less than 256 bytes of stack*
- *Routine using more than 256 bytes of stack* on page 3-13.

For this purpose, leaf routines include routines in which every call is a tail call.

Leaf routine using less than 256 bytes of stack

A leaf routine that uses less than 256 bytes of stack does not have to check the stack limit. This is a consequence of the rules above (see *Rules for stack limit checked code* on page 3-11).

For this purpose, a leaf routine can be a combination of routines with a total stack usage less than 256 bytes.

Nonleaf routine using less than 256 bytes of stack

A nonleaf routine that uses less than 256 bytes of stack can use a limit-checking sequence such as the following:

```
    SUB     sp, sp, #size        ; ARM code version
    CMP     sp, sl
    BLLO    __ARM_stack_overflow
```

or in Thumb code:

```
    ADD     sp, #-size           ; Thumb code version
    CMP     sp, sl
    BLLO    __Thumb_stack_overflow
```

Note

The names `__ARM_stack_overflow` and `__Thumb_stack_overflow` are illustrative and do not correspond to any actual implementation.

Routine using more than 256 bytes of stack

In this case, a new value of sp must be *proposed* to the limit-checking code using a sequence such as the following:

```
    SUB     ip, sp, #size           ; ARM code version
    CMP     ip, sl
    BLLO    __ARM_stack_overflow
```

or in Thumb code:

```
    LDR     r7, #-size              ; Thumb code version
    ADD     r7, sp
    CMP     r7, sl
    BLLO    __Thumb_stack_overflow
```

This is necessary to ensure that sp cannot become less than the lowest usable address in the stack.

Note

The names `__ARM_stack_overflow` and `__Thumb_stack_overflow` are illustrative and do not correspond to any actual implementation.

3.6 Read-only position independence

A program is *Read-Only Position-Independent* (ROPI) if all its read-only segments are position independent.

An ROPI segment is often *Position-Independent Code* (PIC), but could be read-only data, or a combination of PIC and read-only data.

Select the ROPI option to avoid committing yourself to having to load your code in a particular location in memory. This is particularly useful for routines that are:

- loaded in response to run-time events
- loaded into memory with different combinations of other routines in different circumstances
- mapped at different addresses during their execution.

3.6.1 Register usage with ROPI

The usage of registers is the same with or without ROPI (see *Register roles and names* on page 3-4).

3.6.2 Writing code for ROPI

When you are writing code for ROPI:

- Every reference from code in an ROPI segment to a symbol in the same ROPI segment must be PC-relative. ATPCS does not define any other base register for a read-only segment. An address of an item in an ROPI segment cannot be assigned to an item in a different ROPI segment.

- Every reference from code in an ROPI segment to a symbol in a different ROPI segment must be PC-relative. The two segments must be fixed relative to each other.

- Every other reference from an ROPI segment must be to either:
  - — an absolute address
  - — an sb-relative reference to writable data (see *Read-write position independence* on page 3-15).

- A read-write word that addresses a symbol in an ROPI segment must be adjusted whenever the ROPI segment is moved.

3.7 Read-write position independence

A program is *Read-Write Position-Independent* (RWPI) if all its read-write segments are position independent.

An RWPI segment is usually *Position-Independent Data* (PID).

Select the RWPI option to avoid committing yourself to a particular location of data in memory. This is particularly useful for data that must be multiply instantiated for reentrant routines.

3.7.1 Reentrant routines

A reentrant routine can be *threaded* by several processes at the same time. Each process has its own copy of the read-write segments of the routine. Each copy is addressed by a different value of the static base register.

3.7.2 Register usage with RWPI

Register r9 is the static base, sb. It must point to the base address of the appropriate static data segments whenever you call any externally visible routine.

You can use r9 for other purposes in a routine that does not use sb. If you do this you must save the contents of sb on entry to your routine and restore it before exit. You must also restore it before any call to an external routine.

In all other respects the usage of registers is the same with or without RWPI (see *Register roles and names* on page 3-4).

3.7.3 Position-independent data addressing

An RWPI segment can be repositioned until it is first used. The address of a symbol in an RWPI segment is calculated as follows:

1. The linker calculates a read-only offset from a fixed location in the segment. By convention, the fixed location is the first byte of the lowest addressed RWPI segment of the program.
2. At runtime, this is used as an offset added to the contents of the static base register, sb.

3.7.4 Writing assembly language for RWPI

Construct references from a read-only segment to the RWPI segment by adding a fixed (read-only) offset to the value of sb (see `DCDO` in the *Directives Reference* chapter in *RealView Compilation Tools v2.0 Assembler Guide*).

3.8 Interworking between ARM and Thumb states

Select the `/interwork` option when compiling or assembling code if you want:

- ARM routines to be able to return to a Thumb state caller
- Thumb routines to be able to return to an ARM state caller
- the linker to provide the code to change state when calling from ARM to Thumb or from Thumb to ARM.

Select the`/nointerwork` option when compiling or assembling code if either:

- your system does not use Thumb
- you provide the assembler code to handle all changes of state.

The default is:

- `/interwork` if you are compiling or assembling for an ARM v5T processor
- `/nointerwork` otherwise.

If you select the interworking option, you can call a routine in a different module without considering which instruction set it uses. If necessary, the linker inserts an interworking call veneer, or patches the call site. This works for compiled or assembled code.

See Chapter 4 *Interworking ARM and Thumb* for detailed information.

3.8.1 Register usage with interworking

The usage of registers is the same with or without interworking (see *Register roles and names* on page 3-4).

3.9 Floating-point options

The ATPCS supports the following floating-point hardware architectures and instruction sets:

- The VFP architecture (see *The VFP architecture* on page 3-18).
- The FPA architecture (see *The FPA architecture* on page 3-20). This is for backwards compatibility only.

Code for one architecture cannot be used on the other architecture.

The ARM compiler and assembler have the following floating-point options:

- `-fpu vfp`
- `-fpu vfpv1`
- `-fpu vfpv2`
- `-fpu fpa`
- `-fpu softvfp`
- `-fpu softvfp+vfp`
- `-fpu softvfp+vfpv2`
- `-fpu softfpa`
- `-fpu none`.

If your target system has floating-point hardware, choose `vfp`, `softvfp+vfp`, or `fpa`.

Use `softvfp+vfp` or `softvfp+vfpv2` if your system has floating-point hardware, and you want to use floating-point library routines from Thumb code.

If your target system does not have floating-point hardware:

- If you require compatibility with an FPA system, or objects produced under SDT, choose `softfpa`.
- If the module you are compiling or assembling does not use floating-point arithmetic, and you require compatibility with both FPA and VFP systems, choose `none`.
- Otherwise, choose `softvfp`. This is the default.

See also *No floating-point hardware* on page 3-21.

3.9.1 The VFP architecture

The VFP architecture has sixteen double-precision registers, d0-d15. Each double-precision register can be used as two single-precision registers. As single-precision registers they are called s0-s31. d5 for example, is the same as s10 and s11.

The VFP architecture does not support extended precision.

Vector and scalar modes

The VFP architecture has the following modes of operation:

- scalar mode
- vector mode.

The ATPCS applies only to scalar mode operation. On entry to and exit from any publicly visible routine conforming to the ATPCS the vector length and vector stride must both be set to 1.

Register usage with VFP

You can use the first eight double-precision registers, d0-d7:

- to pass floating-point values into a routine
- to pass floating-point values out of a routine
- as scratch registers within a routine.

Each double-precision register can hold one double-precision value or two single-precision values. Floating-point argument values are assigned to floating-point registers by assigning each value in turn to the next free register of the appropriate type.

Figure 3-2 shows the assignment of parameter values to registers when passing the values `1.0` (double), `2.0` (double), `3.0` (single), `4.0` (double), `5.0` (single), and `6.0` (single).

Double view	d0		d1		d2		d3		d4		d5	
Single view	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	
Argument view	1.0		2.0		3.0	5.0	4.0		6.0			

Figure 3-2 Assignment of parameter values to registers

To comply with ATPCS, if you use registers d8-d15 within a routine, you must save their values on entry and restore them before exit. You can save them using a single `FSTMX` instruction and restore them using a single `FLDMX` instruction. They are saved and restored as bit patterns, without interpretation as single or double-precision numbers. *N* single-precision values saved occupy *N*+1 words.

Format of VFP values

Single-precision and double-precision values conform to the IEEE 754 standard formats. Double-precision values are treated as true 64-bit values:

- in little-endian mode, the more significant word of a two-word double-precision value, containing the exponent, has the higher address
- in big-endian mode, the more significant word has the lower address.

Note

Little-endian double-precision values in VFP are pure little-endian. This is different from FPA architecture.

Big-endian double-precision values are the same, pure big-endian, in both VFP and FPA architectures.

IEEE rounding modes and exception enable flags

The ATPCS does not specify any constraint on the state of these on entry to, or exit from, conforming routines.

3.9.2 The FPA architecture

The FPA architecture has eight floating-point registers, f0-f7. Each register can hold a single-precision, double-precision, or extended-precision value.

Register usage with FPA

You can use the first four floating-point registers, f0-f3:

- to pass floating-point values into a routine
- to pass floating-point results out of a routine
- as scratch registers within a routine.

To comply with ATPCS, if you use floating-point registers f4-f7 within a routine, you must save their values on entry and restore them before exit. You can save them using a single `SFM` instruction and restore them using a single `LFM` instruction. Each value saved occupies three words.

Format of FPA values

Single-precision and double-precision values conform to the IEEE 754 standard formats. The most significant word of a floating-point value, containing the exponent, has the lowest memory address. This is the same whether the byte order within words is big-endian or little-endian.

Note

Little-endian double-precision values are neither pure little-endian nor pure big-endian.

IEEE rounding modes and exception enable flags

The ATPCS does not specify any constraint on the state of these on entry to, or exit from, conforming routines.

3.9.3 No floating-point hardware

The only difference between `softvfp` and `softfpa` is the order of words in double-precision values in little-endian mode (see *Format of VFP values* on page 3-19 and *Format of FPA values* on page 3-20).

If you specify `-fpu none`, you cannot use floating-point values.

Register usage with softvfp and softfpa

Each floating-point argument is converted to a bit pattern in one or two integer words as if by storing to memory. The resulting integer values are passed as described in *Parameter passing* on page 3-9.

A single-precision floating-point result is returned as a bit pattern in r0.

A double-precision floating-point result is returned in r0 and r1. r0 contains the word corresponding to the lower-addressed word of the representation of the value in memory.

3.9.4 softvfp+vfp and softvfp+vfpv2

Thumb code cannot pass floating-point values in floating-point registers, as Thumb does not have coprocessor instructions.

If you have a VFP coprocessor and want to use floating-point routines from Thumb code, select the `-fpu softvfp+vfp`, or `-fpu softvfp+vfpv2` option.

This instructs the compiler to generate code using the same parameter passing rules as for `-fpu softvfp`. The C library floating-point routines use VFP instructions from ARM state.

Chapter 4
Interworking ARM and Thumb

This chapter explains how to change between ARM state and Thumb state when writing code for processors that implement the Thumb instruction set. It contains the following sections:

4.1 About interworking

You can mix ARM and Thumb code as you require, provided that the code conforms to the requirements of the *ARM-Thumb Procedure Call Standard* (ATPCS). The ARM compiler always creates code that conforms to this standard. If you are writing ARM assembly language modules you must ensure that your code conforms.

The ARM linker detects when an ARM function is being called from Thumb state, or a Thumb function is being called from ARM state. The ARM linker alters call and return instructions, or inserts small code sections called *veneers*, to change processor state as necessary.

ARM architecture v5T provides methods of changing processor state without using any extra instructions. There is normally no cost associated with interworking on ARM architecture v5T processors.

If you are linking several source files together, all your files must use compatible ATPCS options. If incompatible options are detected, the linker produces an error message.

4.1.1 When to use interworking

When you write code for a Thumb-capable ARM processor, you probably write most of your application to run in Thumb state. This gives the best code density. With 8-bit or 16-bit wide memory, it also gives the best performance. However, you might want parts of your application to run in ARM state for reasons such as:

Speed Some parts of an application might be speed critical. These sections might be more efficient running in ARM state than in Thumb state. In some circumstances, a single ARM instruction can do more than the equivalent Thumb instruction.

Some systems include a small amount of fast 32-bit memory. ARM code can be run from this without the overhead of fetching each instruction from 8-bit or 16-bit memory.

Functionality

Thumb instructions are less flexible than their equivalent ARM instructions. Some operations are not possible in Thumb state. For example, you cannot enable or disable interrupts, or access coprocessors. A state change is required to carry out these operations.

Exception handling

The processor automatically enters ARM state when a processor exception occurs. This means that the first part of an exception handler must be coded with ARM instructions, even if it re-enters Thumb state to carry out the main processing of the exception. At the end of such processing, the processor must be returned to ARM state to return from the handler to the main application.

Standalone Thumb programs

A Thumb-capable ARM processor always starts in ARM state. To run simple Thumb assembly language programs under the debugger, add an ARM header that carries out a state change to Thumb state and then calls the main Thumb routine. See *Example ARM header* on page 4-6 for an example.

4.1.2 Using the /interwork option

The option `--apcs /interwork` is available for the ARM compiler and assembler. If you set this option:

- The compiler or assembler records an interworking attribute in the object file.
- The linker provides interworking veneers for subroutine entry.
- In assembly language, you must write function exit code that returns to the instruction set state of the caller, for example `BX lr`.
- In C or C++, the compiler creates function exit code that returns to the instruction set state of the caller.
- In C or C++, the compiler uses `BX` instructions for indirect or virtual calls.

Use the `/interwork` option if your object file contains:

- Thumb subroutines that might have to return to ARM code
- ARM subroutines that might have to return to Thumb code
- Thumb subroutines that might make indirect or virtual calls to ARM code
- ARM subroutines that might make indirect or virtual calls to Thumb code.

Note

If a module contains functions marked with `#pragma arm` or `#pragma thumb`, the module must typically be compiled with `--apcs /interwork`. This ensures that the functions can be called successfully from the other (ARM or Thumb) state.

Otherwise, you do not have to use the `/interwork` option. For example, your object file might contain any of the following without requiring `/interwork`:

- Thumb code that can be interrupted by an exception. The exception forces the processor into ARM state so no veneer is required.
- Exception handling code that can handle exceptions from Thumb code. No veneer is required for the return.
- Thumb code that calls ARM subroutines in other files (the interworking return sequences belong to the callee, not the caller).
- ARM code that calls Thumb subroutines in other files (the interworking return sequences belong to the callee, not the caller).

4.1.3 Detecting interworking calls

The linker generates an error if it detects a direct ARM/Thumb interworking call where the called routine is not built for interworking. You must rebuild the called routine for interworking.

For example, Example 4-1 shows the error that is produced if the ARM routine in Example 4-3 on page 4-11 is compiled and linked without the `--apcs /interwork` option.

Example 4-1

```
Error: L6239E: Cannot call ARM symbol 'arm_function' in non-interworking object
armsub.o from THUMB code in thumbmain.o(.text)
```

These types of error indicate that an ARM-to-Thumb or Thumb-to-ARM interworking call has been detected from the object module object to the routine symbol, but the called routine has not been compiled for interworking. You must recompile the module that contains the symbol and specify `--apcs /interwork`.

4.2 Assembly language interworking

In an assembly language source file, you can have several areas (these correspond to ELF sections). Each area can contain ARM instructions, Thumb instructions, or both.

You can use the linker to fix up calls to, and returns from, routines that use a different instruction set from the caller. To do this, use `BL` to call the routine (see *Assembly language interworking using veneers* on page 4-14).

If you prefer, you can write your code to make the instruction set changes explicitly. In some circumstances you can write smaller or faster code by doing this.

The following instructions perform the processor state changes:

- `BX`, see *The branch and exchange instruction*
- `BLX`, `LDR`, `LDM`, and `POP` (ARM architecture v5 and above only), see *ARM architecture v5T* on page 4-8.

The following directives instruct the assembler to assemble instructions from the appropriate instruction set (see *Changing the assembler mode* on page 4-6):

- `CODE16`
- `CODE32`.

4.2.1 The branch and exchange instruction

The `BX` instruction branches to the address contained in a specified register. The value of bit 0 of the branch address determines whether execution continues in ARM state or Thumb state. See *ARM architecture v5T* on page 4-8 for additional instructions available with ARM architecture v5.

Bit 0 of an address can be used in this way because:

- all ARM instructions are word-aligned, so bits 0 and 1 of the address of any ARM instruction are unused
- all Thumb instructions are halfword-aligned, so bit 0 of the address of any Thumb instruction is unused.

Syntax

The syntax of `BX` is one of:

Thumb	`BX R`*n*
ARM	`BX{`*cond*`} R`*n*

where:

`Rn`	Is a register in the range r0 to r15 that contains the address to branch to. The value of bit 0 in this register determines the processor state: • if bit 0 is set, the instructions at the branch address are executed in Thumb state • if bit 0 is clear, the instructions at the branch address are executed in ARM state.
`cond`	Is an optional condition code. Only the ARM version of `BX` can be executed conditionally.

4.2.2 Changing the assembler mode

The ARM assembler can assemble both Thumb code and ARM code. By default, it assembles ARM code unless it is invoked with the `-16` option.

Because all Thumb-capable ARM processors start in ARM state, you must use the `BX` instruction to branch and exchange to Thumb state, and then use the `CODE16` directive to instruct the assembler to assemble Thumb instructions. Use the corresponding `CODE32` directive to instruct the assembler to return to assembling ARM instructions.

Refer to the *RealView Compilation Tools v2.0 Assembler Guide* for more information on these directives.

4.2.3 Example ARM header

Example 4-2 on page 4-7 contains four sections of code. The first implements a short header section of ARM code that changes the processor to Thumb state.

The header code uses:

- An `ADR` pseudo-instruction to load the branch address and set the least significant bit. The `ADR` pseudo-instruction generates the address by loading r0 with the value `pc+offset+1`. See *RealView Compilation Tools v2.0 Assembler Guide* for more information on the `ADR` pseudo-instruction.
- A `BX` instruction to branch to the Thumb code and change processor state.

The second section of the module, labelled `ThumbProg`, is prefixed by a `CODE16` directive that instructs the assembler to treat the following code as Thumb code. The Thumb code adds the contents of two registers together.

The processor is changed back to ARM state. The code again uses an `ADR` instruction to get the address of the label, but this time the least significant bit is left clear. The `BX` instruction changes the state.

The third section of the code adds together the contents of two registers.

The final section labeled `stop` uses the semihosting SWI to report normal application exit. Refer to the *RealView Compilation Tools v2.0 Compiler and Libraries Guide* for more information on semihosting.

Note

The Thumb semihosting SWI is a different number from the ARM semihosting SWI (`0xAB` rather than `0x123456`).

Example 4-2

```
     AREA     AddReg,CODE,READONLY         ; Name this block of code.
     ENTRY                    ; Mark first instruction to call.
main
     ADR r0, ThumbProg + 1    ; Generate branch target address
                              ; and set bit 0, hence arrive
                              ; at target in Thumb state.
     BX r0                    ; Branch exchange to ThumbProg.

     CODE16                   ; Subsequent instructions are Thumb code.
ThumbProg
     MOV r2, #2               ; Load r2 with value 2.
     MOV r3, #3               ; Load r3 with value 3.
     ADD r2, r2, r3           ; r2 = r2 + r3
     ADR r0, ARMProg
     BX r0
     CODE32                   ; Subsequent instructions are ARM code.
ARMProg
     MOV r4, #4
     MOV r5, #5
     ADD r4, r4, r5

stop MOV r0, #0x18            ; angel_SWIreason_ReportException
     LDR r1, =0x20026         ; ADP_Stopped_ApplicationExit
     SWI 0x123456             ; ARM semihosting SWI

     END                      ; Mark end of this file.
```

Building the example

To build and execute the example:

1. Enter the code using any text editor and save the file as `addreg.s`.
2. Type `armasm -g addreg.s` at the command prompt to assemble the source file.
3. Type `armlink addreg.o -o addreg` to link the file.
4. Run the image using an ELF/DWARF2 compatible debugger with an appropriate debug target. If you step through the program one instruction at a time, you see the processor enter the Thumb state. See the user documentation for the debugger you are using to find out how this change is indicated.

4.2.4 ARM architecture v5T

In ARM architecture v5T and above:

- The following additional interworking instructions are available:

  `BLX` *`address`*

  The processor performs a PC-relative branch to *`address`* with link and changes state. *`address`* must be within 32MB of the PC in ARM code, or within 4MB of the PC in Thumb code.

  `BLX` *`register`*

  The processor performs a branch with link to an address contained in the specified register. The value of bit[0] determines the new processor state.

  In either case, bit[0] of `lr` is set to the current value of the Thumb bit in the `CPSR`. The means that the return instruction can automatically return to the correct processor state.

- If `LDR`, `LDM`, or `POP` load to the PC, they set the Thumb bit in the `CPSR` to bit[0] of the value loaded to the PC. You can use this to change instruction sets. This is particularly useful for returning from subroutines. The same return instruction can return to either an ARM or Thumb caller.

For more information, see *RealView Compilation Tools v2.0 Assembler Guide*.

4.2.5 Labels in Thumb code

The linker distinguishes between labels referring to:

- ARM instructions
- Thumb instructions
- data.

When the linker relocates a value of a label referring to a Thumb instruction, it sets the least significant bit of the relocated value. This means that a branch to a label can automatically select the appropriate instruction set. This works if any of the following instructions are used for the branch:

- `BX` in ARM architecture v4T
- `BX`, `BLX`, or `LDR` in architecture v5T and above.

In releases of ADS earlier than 1.2 and SDT, it was necessary to mark data in Thumb code with the `DATA` directive. This is no longer necessary.

4.3 C and C++ interworking and veneers

You can freely mix C and C++ code compiled for ARM and Thumb, but in ARM architecture v4T small code segments called *veneers* are required between the ARM and Thumb code to carry out state changes. The ARM linker generates these interworking veneers when it detects interworking calls.

4.3.1 Compiling code for interworking

The `--apcs /interwork` compiler option enables the ARM compiler to compile C and C++ modules containing routines that can be called by routines compiled for the other processor state:

```
armcc --thumb --apcs /interwork
armcc --arm --apcs /interwork
armcc --cpp --thumb --apcs /interwork
armcc --cpp --arm --apcs /interwork
```

Note

`--arm` is the default option.

Modules that are compiled for interworking on ARM architecture v4T generate slightly larger code, typically 2% larger for Thumb and less than 1% larger for ARM. There is no difference for ARM architecture v5.

In a leaf function, that is a function whose body contains no function calls, the only change in the code generated by the compiler is to replace `MOV pc,lr` with `BX lr`. The `MOV` instruction does not cause the necessary state change.

In nonleaf functions built for ARM architecture v4T in the Thumb mode, the compiler must replace, for example, the single instruction:

```
    POP  {r4,r5,pc}
```

with the sequence:

```
    POP  {r4,r5}
    POP  {r3}
    BX   r3
```

This has a small effect on performance. Compile all source modules for interworking, unless you are sure they are never going to be used with interworking.

The `--apcs /interwork` option also sets the interwork attribute for the code area the modules are compiled into. The linker detects this attribute and inserts the appropriate veneer.

Note

ARM code compiled for interworking can only be used on ARM architecture v4T and above, because earlier processors do not implement the BX instruction.

Use the armlink -info veneers option to find the amount of space taken by the veneers.

C interworking example

Example 4-3 shows a Thumb routine that carries out an interworking call to an ARM subroutine. The ARM subroutine call makes an interworking call to printf() in the Thumb library. These two modules are provided in *Examples_directory*\interwork as thumbmain.c and armsub.c.

Example 4-3

```
/*********************
*      thumbmain.c  *
*********************/
#include <stdio.h>
extern void arm_function(void);
int main(void)
{
     printf("Hello from Thumb World\n");
     arm_function();
     printf("And goodbye from Thumb World\n");
     return (0);
}

/*********************
*       armsub.c    *
*********************/
#include <stdio.h>
void arm_function(void)
{
     printf("Hello and Goodbye from ARM world\n");
}
```

To compile and link these modules:

1. Type armcc --thumb -c --apcs /interwork -o thumbmain.o thumbmain.c at the system prompt to compile the Thumb code for interworking.
2. Type armcc -c --apcs /interwork -o armsub.o armsub.c to compile the ARM code for interworking.

3. Type `armlink thumbmain.o armsub.o -o thumbtoarm.axf` to link the object files.

   Alternatively, to view the size of the interworking veneers (Example 4-4) type:

   `armlink armsub.o thumbmain.o -o thumbtoarm.axf -info veneers`

Example 4-4

```
Adding Veneers to the image

    Adding TA veneer (4 bytes, Inline) for call to 'arm_function' from thumbmain.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '_printf' from armsub.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_cmpge' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_neg' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_fp_display_gate' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_ushift_r' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_cmpu' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_udiv10' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_udiv10' from __vfpntf.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__rt_lib_init' from kernel.o(.text).
    Adding AT veneer (12 bytes, Long) for call to '__rt_lib_shutdown' from kernel.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_memclr_w' from stdio.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_raise' from stdio.o(.text).
    Adding TA veneer (8 bytes, Short) for call to '__rt_exit' from exit.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__user_libspace' from free.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_fp_init' from lib_init.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__heap_extend' from malloc.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__raise' from rt_raise.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_errno_addr' from ftell.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '_no_fp_display' from printf2.o(x$fpl$printf2).

20 Veneer(s) (total 108 bytes) added to the image.
```

4.3.2 Basic rules for interworking

The following rules apply to interworking within an application:

- You must use the `--apcs /interwork` command-line option to compile any C or C++ modules that contain functions that might return to the other instruction set.
- You must use the `--apcs /interwork` command-line option to compile any C or C++ modules that contain indirect or virtual function calls that might be to functions in the other instruction set.
- Never make indirect calls, such as calls using function pointers, to non-interworking code from code in the other state.
- If any input object contains Thumb code, the linker selects the Thumb runtime libraries. These are built for interworking.

  If you specify one of your own libraries explicitly on the linker command line you must ensure that it is an appropriate interworking library.

4.3.3 Using two copies of the same function

You can have two functions with the same name, one compiled for ARM and the other for Thumb. However, we do not recommend this practice. In almost all cases there is no significant performance increase over having a single version of the function.

Note

Both versions of the function must be compiled with the `--apcs /interwork` option. This is because there is no guarantee that the Thumb version is going to be called only from Thumb state, and the ARM version is going to be called only from ARM state.

The linker enables duplicate definitions provided that one definition defines a Thumb routine and the other defines an ARM routine.

4.4 Assembly language interworking using veneers

The assembly language ARM/Thumb interworking method described in *Assembly language interworking* on page 4-5 carried out all the necessary intermediate processing. There was no requirement for the linker to insert interworking veneers.

This section describes how you can make use of interworking veneers to:

- interwork between assembly language modules
- interwork between assembly language and C or C++ modules.

4.4.1 Assembly-only interworking using veneers

You can write assembly language ARM/Thumb interworking code to make use of interworking veneers generated by the linker. To do this, you write:

- A caller routine like any non-interworking routine, using a `BL` instruction to make the call. A caller routine can be assembled with either `--apcs /interwork` or `--apcs /nointerwork`.
- A callee routine using a `BX` instruction to return. A callee routine must be assembled with `--apcs /interwork`.

This is generally only necessary in ARM architecture v4T, or if the caller and callee are widely separated or in different areas. In ARM architecture v5T, if the caller and callee are sufficiently close together, no veneers are necessary.

Example of assembly language interworking using veneers

Example 4-5 shows the code to set registers `r0` to `r2` to the values 1, 2, and 3 respectively. Registers `r0` and `r2` are set by the ARM code. `r1` is set by the Thumb code. Observe that:

- the code must be assembled with the option `--apcs /interwork`
- a `BX lr` instruction is used to return from the subroutine, instead of the usual `MOV pc,lr`.

Example 4-5

```
     ; *****
     ; arm.s
     ; *****
     AREA     Arm,CODE,READONLY    ; Name this block of code.
     IMPORT     ThumbProg
     ENTRY                         ; Mark 1st instruction to call.
ARMProg
     MOV  r0,#1                    ; Set r0 to show in ARM code.
     BL   ThumbProg                ; Call Thumb subroutine.
     MOV  r2,#3                    ; Set r2 to show returned to ARM.
                                   ; Terminate execution.
     MOV  r0, #0x18                ; angel_SWIreason_ReportException
     LDR  r1, =0x20026             ; ADP_Stopped_ApplicationExit
     SWI  0x123456                 ; ARM semihosting SWI
     END

     ; *******
     ; thumb.s
     ; *******
     AREA  Thumb,CODE,READONLY     ; Name this block of code.
     CODE16                        ; Subsequent instructions are Thumb.
     EXPORT ThumbProg
ThumbProg
     MOV  r1, #2                   ; Set r1 to show reached Thumb code.
     BX   lr                       ; Return to ARM subroutine.
     END                           ; Mark end of this file.
```

Follow these steps to build and link the modules, and examine the interworking veneers:

1. Type `armasm arm.s` to assemble the ARM code.
2. Type `armasm -16 --apcs /interwork thumb.s` to assemble the Thumb code.
3. Type `armlink arm.o thumb.o -o count` to link the two object files.

4. Run the image using an ELF/DWARF2 compatible debugger with an appropriate debug target.

4.4.2 C, C++, and assembly language interworking using veneers

C and C++ code compiled to run in one state can call assembly language code designed to run in the other state, and vice versa. To do this, write the caller routine as any non-interworking routine and, if calling from assembly language, use a `BL` instruction to make the call (see Example 4-6). Then:

- if the callee routine is in C, compile it using `--apcs /interwork`
- if the callee routine is in assembly language, assemble with the `--apcs /interwork` option and return using `BX lr`.

Note

Any assembly language code or user library code used in this manner must conform to the ATPCS where appropriate.

Example 4-6

```
     /*********************
     *       thumb.c      *
     *********************/
     #include <stdio.h>
     extern int arm_function(int);
     int main(void)
     {
          int i = 1;
          printf("i = %d\n", i);
          printf("And now i = %d\n", arm_function(i));
          return (0);
     }

     ; *****
     ; arm.s
     ; *****
     AREA  Arm,CODE,READONLY ; Name this block of code.
     EXPORT arm_function
arm_function
     ADD   r0,r0,#4           ; Add 4 to first parameter.
     BX    lr                 ; Return
     END
```

Follow these steps to build and link the modules:

1. Type `armcc --thumb -c --apcs /interwork thumb.c` to compile the Thumb code.
2. Type `armasm --apcs /interwork arm.s` to assemble the ARM code.
3. Type `armlink arm.o thumb.o -o add` to link the two object files.
4. Run the image using an ELF/DWARF2 compatible debugger with an appropriate debug target.

Chapter 5
Mixing C, C++, and Assembly Language

This chapter describes how to write mixed C, C++, and ARM assembly language code. It also describes how to use the ARM inline and embedded assemblers from C and C++. It contains the following sections:

5.1 Using the inline assembler

The inline assembler that is built into the ARM compiler enables you to use features of the target processor that cannot be accessed directly from C. For example:

- saturating arithmetic (see *RealView Compilation Tools v2.0 Assembler Guide*)
- custom coprocessors
- the PSR.

The inline assembler supports very flexible interworking with C and C++. Any register operand can be an arbitrary C or C++ expression. The inline assembler also expands complex instructions and optimizes the assembly language code.

Note

Inline assembly language is subject to optimization by the compiler if optimization is enabled either by default or with the `-O1` or `-O2` compiler options.

The inline assembler for ARM code implements most of the ARM instruction set including generic coprocessor instructions, halfword instructions and long multiply.

Note

The inline assembler for Thumb code is not supported in RVCT v2.0.

See *Differences between the inline assembler and armasm* on page 5-7 for information on restrictions.

The inline assembler is a high-level assembler. The code it generates is not always exactly what you write. Do not use it to generate more efficient code than the compiler generates. Use the ARM assembler `armasm` for this purpose.

Some low-level features that are available to the ARM assembler `armasm`, such as branching by writing to PC, are not supported.

For more details on the inline assembler, see the chapter on inline and embedded assemblers in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide*.

5.1.1 Invoking the inline assembler

How you invoke the inline assembler depends on whether you are compiling C or C++:

- When compiling C, the ARM compiler supports inline assembly language with the `__asm` specifier.
- When compiling C++, the ARM compiler supports inline assembly language with the `__asm` specifier and the **`asm`** keyword.

For more details on the inline assembler, see the chapter on inline and embedded assemblers in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide*.

Inline assembler with the __asm specifier

When compiling C or C++ you can invoke the inline assembler with the `__asm` assembler specifier. The specifier is followed by a list of assembler instructions inside braces. For example:

```
__asm
{
    instruction [; instruction]
    ...
    [instruction]
}
```

If two instructions are on the same line, you must separate them with a semicolon. If an instruction is on multiple lines, line continuation must be specified with the backslash character (\). C or C++ comments can be used anywhere within an inline assembly language block.

An `__asm` statement can be used anywhere a C or C++ statement is expected.

Inline assembler with the asm keyword

When compiling C++, the ARM compiler supports the **`asm`** syntax proposed in the ISO C++ Standard, with the restriction that the string literal must be a single string. For example:

```
asm("instruction[;instruction]");
```

The **`asm`** statement must be inside a C++ function. Do not include comments in the string literal. An **`asm`** statement can be used anywhere a C++ statement is expected.

Example addition with inline assembly code

Example 5-1 shows an example of using inline assembly code to add two values. Compare this with the embedded assembly function in Example 5-4 on page 5-13.

Example 5-1 Addition with inline assembly code

```
#include <stdio.h>

void main() {
    int r0=12345;
    int r1=67890;
    int res;

    __asm {
        ADD res, r0, r1    // r0 and r1 are C/C++ variables (virtual registers),
                           // not the physical registers
    }

    printf("12345 + 67890 = %d\n", res);
}
```

5.1.2 The ARM instruction set

The ARM instruction set is described in the *ARM Architecture Reference Manual*. All instruction opcodes and register specifiers can be written in either lowercase or uppercase.

Operand expressions

Any register or constant operand can be an arbitrary C or C++ expression so that variables can be read or written. The expression must be integer assignable, that is, of type **`char`**, **`short`**, **`int`**, or **`long`**. No sign extension is performed on **`char`** and **`short`** types. You must perform sign extension explicitly for these types. The compiler might add code to evaluate these expressions and allocate them to registers.

When an operand is used as a destination, the expression must be assignable (an lvalue). When writing code that uses both physical registers and expressions, you must take care not to use complex expressions that require too many registers to evaluate. The compiler issues an error message if it detects conflicts during register allocation.

Virtual and physical registers

The inline assembler provides no direct access to the physical registers of an ARM processor. If an ARM register name is used as an operand in an inline assembler instruction it always denotes a virtual register and not the physical ARM integer register.

The ARM compiler allocates physical registers to each virtual register as appropriate during optimization and code-generation. However, the physical register used in the assembled code might be different to that specified in the instruction. You can explicitly define these virtual registers as normal C or C++ variables. If they are not defined then the compiler supplies implicit definitions for the virtual registers.

No virtual registers are created for the PC (r15), lr (r14), and sp (r13) registers, and they cannot be read or directly modified in inline assembly code. Any attempt to use these registers results in the error message.

There is no virtual *Processor Status Register* (PSR). Any references to the PSR are always to the physical PSR.

Note

The initial value in each virtual register is unpredictable. You must write to virtual registers before reading from them. The compiler gives a warning if you attempt to read a virtual register before writing to it.

Constants

The constant expression specifier # is optional. If it is used, the expression following it must be constant.

Instruction expansion

The constant in instructions with a constant operand is not limited to the values permitted by the instruction. Instead, such an instruction is translated into a sequence of instructions with the same effect. For example:

```
ADD r0, r0, #1023
```

might be translated into:

```
ADD r0, r0, #1024
SUB r0, r0, #1
```

With the exception of coprocessor instructions, all ARM instructions with a constant operand support instruction expansion. In addition, the `MUL` instruction can be expanded into a sequence of adds and shifts when the third operand is a constant.

The effect of updating the CPSR by an expanded instruction is:

- arithmetic instructions set the NZCV flags correctly.
- logical instructions:
  - set the NZ flags correctly
  - do not change the V flag
  - corrupt the C flag.

Labels

C and C++ labels can be used in inline assembler statements. C and C++ labels can be branched to by branch instructions only in the form:

```
B{cond} label
```

Storage declarations

All storage can be declared in C or C++ and passed to the inline assembler using variables. Therefore, the storage declarations that are supported by `armasm` are not implemented.

SWI and BL instructions

`SWI` and `BL` instructions of the inline assembler enable you to specify three optional register lists following the normal instruction fields. The register lists specify:

- the registers that are the input parameters
- the registers that are output parameters after return
- the registers that are corrupted by the called function.

The syntax for these instructions is:

```
SWI{cond} swi_num, {input_param_list}, {output_value_list}, {corrupt_reg_list}
BL{cond} function, {input_param_list}, {output_value_list}, {corrupt_reg_list}
```

For example:

```
BL foo {r0=expression1, r1=expression2, r2}, {result=r0, r1}
```

An omitted list is assumed to be empty, except that `BL` always corrupts ip and lr. The default corrupted list for `BL` is r0-r3.

If you do not specify any lists, then:

- `r0-r3` are used as input parameters
- `r0` is used for the output value
- `r12` and `r14` are corrupted.

The register lists have the same syntax as `LDM` and `STM` register lists. If the NZCV flags are modified you must specify PSR in the corrupted register list.

5.1.3 Differences between the inline assembler and armasm

There are a number of differences and restrictions between the assembly language accepted by the inline assembler and the assembly language accepted by the ARM assembler. For the inline assembler:

- You cannot get the address of the current instruction using dot notation (.) or `{PC}`.
- The `LDR Rn, =`*`expression`* pseudo-instruction is not supported. Use `MOV Rn,` *`expression`* instead (this can generate a load from a literal pool).
- Label expressions are not supported.
- The `ADR` and `ADRL` pseudo-instructions are not supported.
- The `&` operator cannot be used to denote hexadecimal constants. Use the `0x` prefix instead. For example:

  ```
  __asm {AND x, y, 0xF00}
  ```
- The notation to specify the actual rotate of an 8-bit constant is not available in inline assembly language. This means that where an 8-bit shifted constant is used, the C flag must be regarded as corrupted if the NZCV flags are updated.
- Registers, such as r0-r3, ip, lr, and the NZCV flags in the CPSR must be used with caution. If you use C or C++ expressions, these might be used as temporary registers and NZCV flags might be corrupted by the compiler when evaluating the expression. See *Virtual and physical registers* on page 5-5.
- No virtual registers are created for the PC (r15), lr (r14), and sp (r13) registers, and they cannot be read or directly modified in inline assembly code. See *Virtual and physical registers* on page 5-5 for more details on virtual registers.
- You cannot write to the PC. The `BX`, `BXJ`, `BLX` and `BKPT` instructions are not supported.
- You must not modify the stack. This is not necessary because the compiler stacks and restores any working registers as required automatically. It is not permitted to explicitly stack and restore work registers.

- You can change processor modes, alter the ATPCS registers fp, sl, and sb, or alter the state of coprocessors, but the compiler is unaware of the change. If you change processor mode, you must not use C or C++ expressions until you change back to the original mode because the compiler corrupts the registers for the processor mode to which you have changed.

  Similarly, if you change the state of a floating-point coprocessor by executing floating-point instructions, you must not use floating-point expressions until the original state has been restored.

5.1.4 Usage

The following points apply to using inline assembly language:

- Comma is used as a separator in assembly language, so C expressions with the comma operator must be enclosed in parentheses to distinguish them:

  ```
  __asm {ADD x, y, (f(), z)}
  ```

- Register names in the inline assembler are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name (see *Virtual and physical registers* on page 5-5). If you do not declare the register as a C or C++ variable, then the compiler warns you that it should be decared as a variable.

- Do not save and restore registers in inline assembler. The compiler does this for you. Also, the inline assembler does not provide direct access to the physical registers (see *Virtual and physical registers* on page 5-5). If registers other than CPSR and SPSR are read without being written to, an error message is issued. For example:

  ```
  int f(int x)
  {
      __asm
      {
          STMFD sp!, {r0}    // save r0 - illegal: read before write
          ADD r0, x, 1
          EOR x, r0, x
          LDMFD sp!, {r0}    // restore r0 - not needed.
      }
      return x;
  }
  ```

  The function must be written as:

  ```
  int f(int x)
  {
      int r0;
      __asm
      {
          ADD r0, x, 1
  ```

```
            EOR x, r0, x
        }
        return x;
    }
```

5.1.5 Examples

Example 5-2 and Example 5-3 on page 5-10 demonstrates some of the ways that you can use inline assembly language effectively.

Enabling and disabling interrupts

Interrupts are enabled or disabled by reading the CPSR flags and updating bit 7. Example 5-2 shows how this can be done by using small functions that can be inlined.

This code is also in *Examples_directory*\inline\irqs.c.

These functions work only in a privileged mode, because the control bits of the CPSR and SPSR cannot be changed while in User mode.

Example 5-2 Interrupts

```
__inline void enable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        BIC tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}

__inline void disable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        ORR tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}

int main(void)
{
    disable_IRQ();
    enable_IRQ();
}
```

Dot product

Example 5-3 calculates the dot product of two integer arrays. It demonstrates how inline assembly language can interwork with C or C++ expressions and data types that are not directly supported by the inline assembler. The inline function `mlal()` is optimized to a single `SMLAL` instruction. Use the `-S -fs` compiler option to view the assembly language code generated by the compiler.

`long long` is the same as `__int64`.

This code is also in *`Examples_directory`*`\inline\dotprod.c`.

Example 5-3 Dot product

```
#include <stdio.h>
/* change word order if big-endian */
#define lo64(a) (((unsigned*) &a)[0])    /* low 32 bits of a long long */
#define hi64(a) (((int*) &a)[1])        /* high 32 bits of a long long */

__inline __int64 mlal(__int64 sum, int a, int b)
{
#if !defined(__thumb) && defined(__TARGET_FEATURE_MULTIPLY)
    __asm
    {
    SMLAL lo64(sum), hi64(sum), a, b
    }
#else
    sum += (__int64) a * (__int64) b;
#endif
    return sum;
}

__int64 dotprod(int *a, int *b, unsigned n)
{
    __int64 sum = 0;
    do
        sum = mlal(sum, *a++, *b++);
    while (--n != 0);
    return sum;
}
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int b[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
int main(void)
{
    printf("Dotproduct %lld (should be %d)\n", dotprod(a, b, 10), 220);
    return 0;
}
```

5.2 Using the embedded assembler

The embedded assembler that is built into the ARM compiler enables you to include assembly code out-of-line, in one or more C or C++ function definitions. Embedded assembler provides unrestricted, low-level access to the target processor, and enables you to use the C and C++ preprocessor directives and easy access to structure member offsets.

The embedded assembler enables you to use the full ARM assembler instruction set, including assembler directives. Embedded assembly code is assembled separately from the C and C++ code. A compiled object is produced that is then combined with the object from the compilation of the C and C++ source.

Embedded assembler is supported in both ARM and Thumb code. See the *RealView Compilation Tools v2.0 Assembler Guide* for details of the ARM instruction set.

For more details on the embedded assembler, see the chapter on inline and embedded assemblers in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide*.

5.2.1 Invoking the embedded assembler

An embedded assembly function definition is marked by the `__asm` function qualifier. Functions declared with `__asm` can have arguments, and return a type. They are called from C and C++ in the same way as normal C and C++ functions. The syntax of an embedded assembly function is:

```
__asm return-type function-name(parameter-list)
{
    // ARM/Thumb assembler code

    instruction
    ...
    [instruction]
}
```

Note

Argument names are permitted in the parameter list, but they cannot be used in the body of the embedded assembly function. For example, the following function uses integer `i` in the body of the function, but this is not valid in assembly:

```
__asm int f(int i) {
    ADD i, i, #1 // error
}
```

You can use, for example, `r0` instead of `i`.

Example addition with an embedded assembly function

Example 5-4 shows an example of using an embedded assembly function to add two values. Compare this with the inline assembly code in Example 5-1 on page 5-4.

Example 5-4 Addition with an embedded assembly function

```
#include <stdio.h>

__asm int add(int i, int j) {
    ADD  r0,r0,r1       // Value of i in r0 and j in r1, result in r0
    MOV  pc,lr
}

void main() {
    printf("12345 + 67890 = %d\n", add(12345, 67890));
}
```

5.2.2 Restrictions on embedded assembly

The following restrictions apply to embedded assembly functions:

- After preprocessing, `__asm` functions can only contain assembly code, with the exception of the following identifiers:

  ```
  __cpp(expr)
  __offsetof_base(D, B)
  __mcall_is_virtual(D, f)
  __mcall_is_in_vbase(D, f)
  __mcall_this_offset(D, f)
  __vcall_offsetof_vfunc(D, f)
  ```

- No return instructions are generated by the compiler for an `__asm` function. If you want to return from an `__asm` function, then you must include the return instructions, in assembly code, in the body of the function.

  Note

  This makes it possible to fall through to the next function, because the embedded assembler guarantees to emit the `__asm` functions in the order you have defined them. However, inlined and template functions behave differently.

- `__asm` functions do not change the ATPCS rules that apply. This means that all calls between an `__asm` function and a normal C or C++ function must adhere to the ATPCS, even though there are no restrictions on the assembly code that an `__asm` function can use (for example, change state).

See the chapter on inline and embedded assemblers in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide* for more details.

5.2.3 Differences between embedded assembler and C or C++

Be aware of the following differences between embedded assembler and C or C++:

- Assembler expressions are always unsigned. The same expression might have different values between assembler and C or C++. For example:

  ```
  MOV r0, #(-33554432 / 2)        // result is 0x7f000000
  MOV r0, #__cpp(-33554432 / 2)   // result is 0xff000000
  ```

- Assembler numbers with leading zeroes are still decimal. For example:

  ```
  MOV r0, #0700              // decimal 700
  MOV r0, #__cpp(0700)       // octal 0700 == decimal 448
  ```

- Assembler operator precedence differs from C and C++. For example:

  ```
  MOV r0, #(0x23 :AND: 0xf + 1)     // ((0x23 & 0xf) + 1) => 4
  MOV r0, #__cpp(0x23 & 0xf + 1)    // (0x23 & (0xf + 1)) => 0
  ```

- Assembler strings are not null-terminated:

  ```
  DCB "no trailing null"                // 16 bytes
  DCB __cpp("I have a trailing null!!")   // 25 bytes
  ```

Note

The assembler rules apply outside of `__cpp`, and the C or C++ rules apply inside `__cpp`. See the chapter on inline and embedded assemblers in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide* for details on the `__cpp` keyword.

5.3 Differences between inline and embedded assembly code

There are differences between the way inline and embedded assembly is compiled:

- Inline assembly code uses a high-level of processor abstraction, and is integrated with the C and C++ code during code generation. Therefore, the compiler optimizes the C and C++ code, and the assembly code together.
- Unlike inline assembly code, embedded assembly code is assembled separately from the C and C++ code to produce a compiled object that is then combined with the object from the compilation of the C or C++ source.
- Inline assembly code can be inlined by the compiler, but embedded assembly code cannot be inlined, either implicitly or explicitly.

Table 5-1 summarizes the main differences between inline assembler and embedded assembler.

Table 5-1 Differences between inline and embedded assembler

Feature	Embedded assembler	Inline assembler
Instruction set	ARM and Thumb	ARM only
ARM assembler directives	All supported	None supported
C/C++ expressions	Constant expressions only	Full C/C++ expressions
Optimization of assembly code	No optimization	Full optimization
Inlining	No	Possible
Register access	Specified physical registers are used. You can also use PC, LR and SP.	Uses virtual registers (see *Virtual and physical registers* on page 5-5)
Return instructions	You must add them in your code.	Generated automatically

Note

A list of differences between embedded assembler and C or C++ is provided in *Differences between embedded assembler and C or C++* on page 5-14.

5.4 Accessing C global variables from assembly code

Global variables can only be accessed indirectly, through their address. To access a global variable, use the `IMPORT` directive to import the global and then load the address into a register. You can access the variable with load and store instructions, depending on its type.

For **`unsigned`** variables use:

- `LDRB/STRB` for **`char`**
- `LDRH/STRH` for **`short`** (use two `LDRB/STRB` instructions for ARM architecture v3)
- `LDR/STR` for **`int`**.

For **`signed`** variables, use the equivalent signed instruction, such as `LDRSB` and `LDRSH`.

Small structures of less than eight words can be accessed as a whole using the `LDM` and `STM` instructions. Individual members of structures can be accessed by a load or store instruction of the appropriate type. You must know the offset of a member from the start of the structure in order to access it.

Example 5-5 loads the address of the integer global `globvar` into r1, loads the value contained in that address into r0, adds 2 to it, then stores the new value back into `globvar`.

Example 5-5 Address of global

```
    AREA     globals,CODE,READONLY

    EXPORT    asmsubroutine
    IMPORT    globvar

asmsubroutine
    LDR  r1, =globvar   ; read address of globvar into
                        ; r1 from literal pool
    LDR  r0, [r1]
    ADD  r0, r0, #2
    STR  r0, [r1]
    MOV  pc, lr
    END
```

5.5 Using C header files from C++

This section describes how to use C header files from your C++ code. C header files must be wrapped in `extern "C"` directives before they are called from C++.

5.5.1 Including system C header files

To include standard system C header files, such as `stdio.h`, you do not have to do anything special. The standard C header files already contain the appropriate `extern "C"` directives. For example:

```
#include <stdio.h>
int main()
{
    ...       // C++ code
    return 0;
}
```

If you include headers using this syntax, all library names are placed in the global namespace.

The C++ standard specifies that the functionality of the C header files is available through C++ specific header files. These files are installed in *install_directory*\RVCT\Data\2.0\build_num\platform\include, together with the standard C header files, and can be referenced in the usual way. For example:

```
#include <cstdio>
int main()
{
    ...       // C++ code
    return 0;
}
```

In ARM C++, these headers `#include` the C headers. If you include headers using this syntax, all C++ standard library names are defined in the namespace `std`, including the C library names. This means that you must qualify all the library names by using one of the following methods:

- specify the standard namespace, for example:

  ```
  std::printf("example\n");
  ```

- use the C++ keyword **`using`** to import a name to the global namespace:

  ```
  using namespace std;
  printf("example\n");
  ```

- use the compiler option `--using_std`.

5.5.2 Including your own C header files

To include your own C header files, you must wrap the `#include` directive in an `extern "C"` statement. You can do this in the following ways:

- When the file is `#included`. This is shown in Example 5-6.
- By adding the `extern "C"` statement to the header file. This is shown in Example 5-7.

Example 5-6 Directive before include file

```
// C++ code

extern "C" {
#include "my-header1.h"
#include "my-header2.h"
}

int main()
{
    // ...
    return 0;
}
```

Example 5-7 Directive in file header

```
/* C header file */

#ifdef __cplusplus    /* Insert start of extern C construct */
extern "C" {
#endif

/* Body of header file */

#ifdef __cplusplus  /* Insert end of extern C construct */
}                   /* The C header file can now be */
#endif              /* included in either C or C++ code. */
```

5.6 Calling between C, C++, and ARM assembly language

This section provides examples that can help you to call C and assembly language code from C++, and to call C++ code from C and assembly language. It also describes calling conventions and data types.

You can mix calls between C and C++ and assembly language routines provided you follow the appropriate procedure ATPCS call standard. For more information on the ATPCS, see Chapter 3 *Using the Procedure Call Standard*.

Note

The information in this section is implementation dependent and might change in future toolkit releases.

5.6.1 General rules for calling between languages

The following general rules apply to calling between C, C++, and assembly language. For more details, see the *RealView Compilation Tools v2.0 Compiler and Libraries Guide*.

The embedded assembler and compliance with the ARM Embedded Application Binary Interface (EABI) makes mixed language programming easier to implement. These assist you with:

- name mangling, using the `__cpp` keyword
- the way the implicit **`this`** parameter is passed
- the way virtual functions are called
- the representation of references
- the layout of C++ class types that have base classes or virtual member functions
- the passing of class objects that are not *plain old data* (POD) structures.

The following general rules apply to mixed language programming:

- Use C calling conventions.
- In C++, nonmember functions can be declared as `extern "C"` to specify that they have C linkage. In this release of RVCT, having C linkage means that the symbol defining the function is not mangled. C linkage can be used to implement a function in one language and call it from another.

  Note

  Functions that are declared `extern "C"` cannot be overloaded.

- Assembly language modules must conform to the appropriate ATPCS call standard for the memory model used by the application.

The following rules apply to calling C++ functions from C and assembly language:

- To call a global (nonmember) C++ function, declare it `extern "C"` to give it C linkage.
- Member functions (both static and non-static) always have mangled names. Using the `__cpp` keyword of the embedded assembler means that you do not have to find the mangled names manually.
- C++ inline functions cannot be called from C unless you ensure that the C++ compiler generates an out-of-line copy of the function. For example, taking the address of the function results in an out-of-line copy.
- Nonstatic member functions receive the implicit **`this`** parameter as a first argument in r0, or as a second argument in r1 if the function returns a non `int`-like structure. Static member functions do not receive an implicit **`this`** parameter.

5.6.2 Information specific to C++

The following applies specifically to C++.

C++ calling conventions

ARM C++ uses the same calling conventions as ARM C with the following exceptions:

- Nonstatic member functions are called with the implicit **`this`** parameter as the first argument, or as the second argument if the called function returns a non **`int`**-like **`struct`**. This might change in future implementations.

C++ data types

ARM C++ uses the same data types as ARM C with the following exceptions and additions:

- C++ objects of type **`struct`** or **`class`** have the same layout that is expected from ARM C if they have no base classes or virtual functions. If such a **`struct`** has neither a user-defined copy assignment operator or a user-defined destructor, it is a POD structure.
- References are represented as pointers.
- No distinction is made between pointers to C functions and pointers to C++ (nonmember) functions.

Symbol name mangling

The linker unmangles symbol names in messages.

C names must be declared as `extern "C"` in C++ programs. This is done already for the ARM ISO C headers. Refer to *Using C header files from C++* on page 5-17 for more information.

5.6.3 Examples

The following sections contain code examples that demonstrate:

- *Calling assembly language from C*
- *Calling C from assembly language* on page 5-23
- *Calling C from C++* on page 5-23
- *Calling assembly language from C++* on page 5-24
- *Calling C++ from C* on page 5-25
- *Calling C++ from assembly language* on page 5-26
- *Calling C++ from C or assembly language* on page 5-28
- *Passing a reference between C and C++* on page 5-27.

The examples assume the default non software-stack checking ATPCS variant because they perform stack operations without checking for stack overflow.

Calling assembly language from C

Example 5-8 and Example 5-9 on page 5-22 show a C program that uses a call to an assembly language subroutine to copy one string over the top of another string.

Example 5-8 Calling assembly language from C

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{   const char *srcstr = "First string - source ";
    char dststr[] = "Second string - destination ";
/* dststr is an array since we're going to change it */
    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcopy(dststr,srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    return (0);
}
```

Example 5-9 Assembly language string copy subroutine

```
    AREA    SCopy, CODE, READONLY
    EXPORT strcopy
strcopy               ; r0 points to destination string.
                      ; r1 points to source string.
    LDRB r2, [r1],#1  ; Load byte and update address.
    STRB r2, [r0],#1  ; Store byte and update address.
    CMP r2, #0        ; Check for zero terminator.
    BNE strcopy       ; Keep going if not.
    MOV pc,lr         ; Return.
    END
```

Example 5-8 on page 5-21 is located in `Examples_directory\asm` as `strtest.c` and `scopy.s`. Follow these steps to build the example from the command line:

1. Type `armasm -g scopy.s` to build the assembly language source.
2. Type `armcc -c -g strtest.c` to build the C source.
3. Type `armlink strtest.o scopy.o -o strtest` to link the object files.
4. Run the image using an ELF/DWARF2 compatible debugger with an appropriate debug target.

Calling C from assembly language

Example 5-10 and Example 5-11 show how to call C from assembly language.

Example 5-10 Defining the function in C

```
int g(int a, int b, int c, int d, int e)
{
        return a + b + c + d + e;
}
```

Example 5-11 Assembly language call

```
    ; int f(int i) { return g(i, 2*i, 3*i, 4*i, 5*i); }

    EXPORT f
    AREA f, CODE, READONLY
    IMPORT g           ; i is in r0
    STR lr, [sp, #-4]! ; preserve lr
    ADD r1, r0, r0     ; compute 2*i (2nd param)
    ADD r2, r1, r0     ; compute 3*i (3rd param)
    ADD r3, r1, r2     ; compute 5*i
    STR r3, [sp, #-4]! ; 5th param on stack
    ADD r3, r1, r1     ; compute 4*i (4th param)
    BL g               ; branch to C function
    ADD sp, sp, #4     ; remove 5th param
    LDR pc, [sp], #4   ; return
    END
```

Calling C from C++

Example 5-12 and Example 5-13 on page 5-24 show how to call C from C++.

Example 5-12 Calling a C function from C++

```
struct S {              // has no base classes
                        // or virtual functions
    S(int s) : i(s) { }
    int i;
};
extern "C" void cfunc(S *);
// declare the C function to be called from C++
int f(){
    S s(2);          // initialize 's'
    cfunc(&s);       // call 'cfunc' so it can change 's'
   return s.i * 3;
}
```

Example 5-13 Defining the function in C

```
struct S {
    int i;
};
void cfunc(struct S *p) {
/* the definition of the C function to be called from C++ */
    p->i += 5;
}
```

Calling assembly language from C++

Example 5-14 and Example 5-15 on page 5-25 show how to call assembly language from C++.

Example 5-14 Calling assembly language from C++

```
struct S {        // has no base classes
                  // or virtual functions
    S(int s) : i(s) { }
    int i;
};

extern "C" void asmfunc(S *);   // declare the Asm function
                                // to be called
int f() {
    S s(2);                    // initialize 's'
    asmfunc(&s);               // call 'asmfunc' so it
                               // can change 's'
    return s.i * 3;
}
```

Example 5-15 Defining the assembly language function

```
    AREA Asm, CODE
    EXPORT asmfunc
asmfunc                 ; the definition of the Asm
    LDR r1, [r0]        ; function to be called from C++
    ADD r1, r1, #5
    STR r1, [r0]
    MOV pc, lr
    END
```

Calling C++ from C

Example 5-16 and Example 5-17 show how to call C++ from C.

Example 5-16 Defining the function to be called in C++

```
struct S {        // has no base classes or virtual functions
    S(int s) : i(s) { }
    int i;
};

extern "C" void cppfunc(S *p) {
// Definition of the C++ function to be called from C.
// The function is written in C++, only the linkage is C
    p->i += 5;                //
}
```

Example 5-17 Declaring and calling the function in C

```
struct S {
    int i;
};

extern void cppfunc(struct S *p);
/* Declaration of the C++ function to be called from C */

int f(void) {
    struct S s;
    s.i = 2;                /* initialize 's' */
    cppfunc(&s);            /* call 'cppfunc' so it */
```

```
                        /* can change 's' */
    return s.i * 3;
}
```

Calling C++ from assembly language

Example 5-18 and Example 5-19 show how to call C++ from assembly language.

Example 5-18 Defining the function to be called in C++

```
struct S {           // has no base classes or virtual functions
    S(int s) : i(s) { }
    int i;
};
extern "C" void cppfunc(S * p) {
// Definition of the C++ function to be called from ASM.
// The body is C++, only the linkage is C
    p->i += 5;
}
```

In ARM assembly language, import the name of the C++ function and use a Branch with link instruction to call it:

Example 5-19 Defining assembly language function

```
    AREA Asm, CODE
    IMPORT cppfunc    ; import the name of the C++
                      ; function to be called from Asm

    EXPORT   f
f
    STMFD  sp!,{lr}
    MOV    r0,#2
    STR    r0,[sp,#-4]! ; initialize struct
    MOV    r0,sp        ; argument is pointer to struct
    BL     cppfunc      ; call 'cppfunc' so it can change
                        ; the struct
    LDR    r0, [sp], #4
    ADD    r0, r0, r0,LSL #1
    LDMFD  sp!,{pc}
    END
```

Passing a reference between C and C++

Example 5-20 and Example 5-21 show how to pass a reference between C and C++.

Example 5-20 C++ function

```
extern "C" int cfunc(const int&);
// Declaration of the C function to be called from C++

extern "C" int cppfunc(const int& r) {
// Definition of the C++ to be called from C.
    return 7 * r;
}

int f() {
    int i = 3;
    return cfunc(i);    // passes a pointer to 'i'
}
```

Example 5-21 Defining the C function

```
extern int cppfunc(const int*);
/* declaration of the C++ to be called from C */

int cfunc(const int* p) {
/* definition of the C function to be called from C++ */
    int k = *p + 4;
    return cppfunc(&k);
}
```

Calling C++ from C or assembly language

The code in Example 5-22, Example 5-23 and Example 5-24 on page 5-29 demonstrates how to call a non-static, non-virtual C++ member function from C or assembly language. Use the assembler output from the compiler to locate the mangled name of the function.

Example 5-22 Calling a C++ member function

```
struct T {
    T(int i) : t(i) { }
    int t;
    int f(int i);
};

int T::f(int i) { return i + t; }
// Definition of the C++ function to be called from C.

extern "C" int cfunc(T*);
// declaration of the C function to be called from C++

int f() {
    T t(5);                    // create an object of type T
    return cfunc(&t);
}
```

Example 5-23 Defining the C function

```
struct T;

extern int _ZN1T1fEi(struct T*, int);
    /* the mangled name of the C++ */
    /* function to be called */

int cfunc(struct T* t) {
/* Definition of the C function to be called from C++. */
    return 3 * _ZN1T1fEi(t, 2);    /* like '3 * t->f(2)' */
}
```

Example 5-24 Implementing the function in assembly language

```
   EXPORT cfunc
   AREA cfunc, CODE
   IMPORT  _ZN1T1fEi
   STMFD   sp!,{lr}  ; r0 already contains the object pointer
   MOV r1, #2
   BL _ZN1T1fEi
   ADD r0, r0, r0, LSL #1   ; multiply by 3
   LDMFD sp!,{pc}
   END
```

Alternatively, you can implement Example 5-22 on page 5-28 and Example 5-24 using embedded assembly, as shown in Example 5-25. In this example, the `__cpp` keyword is used to reference the function. Therefore, you do not have to know the mangled name of the function.

Example 5-25 Implementing the function in embedded assembly

```
struct T {
   T(int i) : t(i) { }
   int t;
   int f(int i);
};
int T::f(int i) { return i + t; }

// Definition of asm function called from C++
__asm int asm_func(T*) {
   STMFD sp!, {lr}
   MOV r1, #2;
   BL __cpp(T::f);
   ADD r0, r0, r0, LSL #1 ; multiply by 3
   LDMFD sp!, {pc}
}

int f() {
   T t(5); // create an object of type T
   return asm_func(&t);
}
```

Chapter 6
Handling Processor Exceptions

This chapter describes how to handle the various types of exception supported by ARM processors. It contains the following sections:

6.1 About processor exceptions

During the normal flow of execution through a program, the program counter increases sequentially through the address space, with branches to nearby labels or branch and links to subroutines.

Processor exceptions occur when this normal flow of execution is diverted, to enable the processor to handle events generated by internal or external sources. Examples of such events are:

- externally generated interrupts
- an attempt by the processor to execute an undefined instruction
- accessing privileged operating system functions.

It is necessary to preserve the previous processor status when handling such exceptions, so that execution of the program that was running when the exception occurred can resume when the appropriate exception routine has completed.

Table 6-1 shows the different types of exception recognized by ARM processors.

Table 6-1 Exception types

Exception	Description
Reset	Occurs when the processor reset pin is asserted. This exception is only expected to occur for signalling power-up, or for resetting as if the processor has powered up. A soft reset can be done by branching to the reset vector (`0x0000`).
Undefined Instruction	Occurs if neither the processor, or any attached coprocessor, recognizes the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction.It enables a program running in User mode, for example, to request privileged operations that run in Supervisor mode, such as an RTOS function.
Prefetch Abort	Occurs when the processor attempts to execute an instruction that was not fetched, because the address was illegal[a].
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address[a].
IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

a. An illegal virtual address is one that does not currently correspond to an address in physical memory, or one that the memory management subsystem has determined is inaccessible to the processor in its current mode.

6.1.1 The vector table

Processor exception handling is controlled by a *vector table*. The vector table is a reserved area of 32 bytes, usually at the bottom of the memory map. It has one word of space allocated to each exception type, and one word that is currently reserved.

This is not enough space to contain the full code for a handler, so the vector entry for each exception type typically contains a branch instruction or load PC instruction to continue execution with the appropriate handler.

6.1.2 Use of modes and registers by exceptions

Typically, an application runs in *User mode*, but servicing exceptions requires privileged (that is, non-User mode) operation. An exception changes the processor mode, and this in turn means that each exception handler has access to a certain subset of the banked registers:

- its own `r13` or *Stack Pointer* (sp_*mode*)
- its own `r14` or *Link Register* (lr_*mode*)
- its own *Saved Program Status Register* (spsr_ *mode*).

In the case of a FIQ, each exception handler has access to five more general purpose registers (`r8_FIQ` to `r12_FIQ`).

Each exception handler must ensure that other registers are restored to their original contents on exit. You can do this by saving the contents of any registers that the handler has to use onto its stack and restoring them before returning. If you are using Angel™ or ARMulator®, the required stacks are set up for you. Otherwise, you must set them up yourself.

Note

As supplied, the assembler does *not* predeclare symbolic register names of the form *`register_mode`*. To use this form, you must declare the appropriate symbolic names with the `RN` assembler directive. For example, `lr_FIQ RN r14` declares the symbolic register name `lr_FIQ` for r14. See the directives chapter in *RealView Compilation Tools v2.0 Assembler Guide* for more information on the `RN` directive.

6.1.3 Exception priorities

When several exceptions occur simultaneously, they are serviced in a fixed order of priority. Each exception is handled in turn before execution of the user program continues. It is not possible for all exceptions to occur concurrently. For example, the Undefined Instruction and SWI exceptions are mutually exclusive because they are both triggered by executing an instruction.

Table 6-2 shows the exceptions, their corresponding processor modes and their handling priorities.

Because the Data Abort exception has a higher priority than the FIQ exception, the Data Abort is actually registered before the FIQ is handled. The Data Abort handler is entered, but control is then passed immediately to the FIQ handler. When the FIQ has been handled, control returns to the Data Abort handler. This means that the data transfer error does not escape detection as it would if the FIQ were handled first.

Table 6-2 Exception priorities

Vector address	Exception type	Exception mode	Priority (1=high, 6=low)
`0x0`	Reset	Supervisor (SVC)	1
`0x4`	Undefined Instruction	Undef	6
`0x8`	Software Interrupt (SWI)	Supervisor (SVC)	6
`0xC`	Prefetch Abort	Abort	5
`0x10`	Data Abort	Abort	2
`0x14`	*Reserved*	*Not applicable*	*Not applicable*
`0x18`	Interrupt (IRQ)	Interrupt (IRQ)	4
`0x1C`	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

6.2 Determining the Processor State

An exception handler might have to determine whether the processor was in ARM or Thumb state when the exception occurred. SWI handlers, especially, might have to read the processor state. This is done by examining the SPSR T-bit. This bit is set for Thumb state and clear for ARM state.

Both ARM and Thumb instruction sets have the `SWI` instruction. When calling SWIs from Thumb state, you must consider:

- the address of the instruction is at (lr – 2), rather than (lr – 4)
- the instruction itself is 16-bit, and so requires a halfword load (see Figure 6-1)
- the SWI number is held in 8 bits instead of the 24 bits in ARM state.

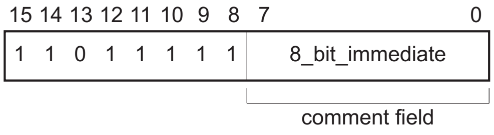


Figure 6-1 Thumb SWI instruction

Example 6-1 shows ARM code that handles a `SWI` from both sources. Consider the following points:

- Each of the `do_swi_x` routines could carry out a switch to Thumb state and back again to improve code density if required.
- You can replace the jump table by a call to a C function containing a `switch()` statement to implement the SWIs.
- It is possible for a SWI number to be handled differently depending on the state it is called from.
- The range of SWI numbers accessible from Thumb state can be increased by calling SWIs dynamically (as described in *SWI handlers* on page 6-18).

Example 6-1

```
T_bit   EQU    0x20                     ; Thumb bit of CPSR/SPSR, that is, bit 5.
        :
        :
SWIHandler
        STMFD   sp!, {r0-r3,r12,lr}     ; Store registers.
        MRS     r0, spsr                ; Move SPSR into general purpose register.
        TST     r0, #T_bit              ; Occurred in Thumb state?
        LDRNEH  r0,[lr,#-2]             ; Yes: load halfword and...
        BICNE   r0,r0,#0xFF00           ; ...extract comment field.
        LDREQ   r0,[lr,#-4]             ; No: load word and...
        BICEQ   r0,r0,#0xFF000000       ; ...extract comment field.

            ; r0 now contains SWI number

        CMP     r0, #MaxSWI             ; Rangecheck
        LDRLS   pc, [pc, r0, LSL#2]     ; Jump to the appropriate routine.
        B       SWIOutOfRange
switable
        DCD     do_swi_1
        DCD     do_swi_2
        :
        :
do_swi_1
        ; Handle the SWI.
        LDMFD   sp!, {r0-r3,r12,pc}^   ; Restore the registers and return.
do_swi_2
        :
```

6.3 Entering and leaving an exception

This section describes the processor response to an exception, and how to return to the place where an exception occurred after the exception has been handled. The method for returning is different depending on the exception type.

Thumb-capable processors use the same basic exception handling mechanism as processors that are not Thumb-capable. An exception causes the next instruction to be fetched from the appropriate vector table entry.

The same vector table is used for both Thumb-state and ARM-state exceptions. An initial step that switches to ARM state is added to the exception handling procedure described in *The processor response to an exception*.

Where there are additional considerations that you must take into account when writing exception handlers suitable for use on Thumb-capable processors, these are clearly marked.

6.3.1 The processor response to an exception

When an exception is generated, the processor takes the following actions:

1. Copies the *Current Program Status Register* (CPSR) into the *Saved Program Status Register* (SPSR) for the mode in which the exception is to be handled. This saves the current mode, interrupt mask, and condition flags.
2. *For Thumb-capable processors only,* switches to ARM state.
3. Changes the appropriate CPSR mode bits in order to:
    - Change to the appropriate mode, and map in the appropriate banked registers for that mode.
    - Disable interrupts. IRQs are disabled when any exception occurs. FIQs are disabled when a FIQ occurs, and on reset.
4. Sets lr_*mode* to the return address, as defined in *The return address and return instruction* on page 6-9.
5. Sets the program counter to the vector address for the exception.

    For ARM processors that are not Thumb-capable, this forces a branch to the appropriate exception handler.

    For Thumb-capable processors, the switch from Thumb state to ARM state in step 2 ensures that the ARM instruction installed at this vector address (either a branch or a PC-relative load) is correctly fetched, decoded, and executed. This forces a branch to a top-level veneer that you must write in ARM code.

6.3.2 Returning from an exception handler

The method used to return from an exception depends on whether the exception handler uses stack operations or not. In both cases, to return execution to the place where the exception occurred an exception handler must:

- restore the CPSR from spsr_*mode*
- restore the program counter using the return address stored in lr_*mode*.

For a simple return that does not require the destination mode registers to be restored from the stack, the exception handler carries out these operations by performing a data processing instruction with:

- the S flag set
- the program counter as the destination register.

The return instruction required depends on the type of exception. See *The return address and return instruction* on page 6-9 for instructions on how to return from each exception type.

Note

You do not have to return from the reset handler because the reset handler executes your main code directly.

If the exception handler entry code uses the stack to store registers that must be preserved while it handles the exception, it can return using a load multiple instruction with the ^ qualifier. For example, an exception handler can return in one instruction using:

```
LDMFD sp!,{r0-r12,pc}^
```

if it saves the following onto the stack:

- all the work registers in use when the handler is invoked
- the link register, modified to produce the same effect as the data processing instructions described below.

The ^ qualifier specifies that the CPSR is restored from the SPSR. It must be used only from a privileged mode. See the description of Implementing stacks with `LDM` and `STM` in the *RealView Compilation Tools v2.0 Assembler Guide* for more general information on stack operations.

6.3.3 The return address and return instruction

The actual location pointed to by the program counter when an exception is taken depends on the exception type. The return address might not necessarily be the next instruction pointed to by the program counter.

If an exception occurs in ARM state, the processor stores (PC – 4) in lr_ *mode*. However, for exceptions that occur in Thumb state, the processor automatically stores a different value for each of the exception types. This adjustment is required because Thumb instructions take up only a halfword, rather than the full word that ARM instructions occupy.

If this correction were not made by the processor, the handler would have to determine the original state of the processor, and use a different instruction to return to Thumb code rather than ARM code. By making this adjustment, however, the processor enables the handler to have a single return instruction that returns correctly, regardless of the processor state (ARM or Thumb) at the time the exception occurred.

The following sections detail the instructions to return correctly from handling code for each type of exception.

Returning from SWI and Undefined Instruction handlers

The SWI and Undefined Instruction exceptions are generated by the instruction itself, so the program counter is not updated when the exception is taken. The processor stores (PC – 4) in lr_ *mode*. This makes lr_*mode* point to the next instruction to be executed. Restoring the program counter from the lr with:

```
    MOVS        pc, lr
```

returns control from the handler.

The handler entry and exit code to stack the return address and pop it on return is:

```
    STMFD       sp!,{reglist,lr}
    ;...
    LDMFD       sp!,{reglist,pc}^
```

For exceptions that occur in Thumb state, the handler return instruction (`MOVS pc,lr`) changes the program counter to the address of the next instruction to execute. This is at (PC – 2), so the value stored by the processor in lr_*mode* is (PC – 2).

Returning from FIQ and IRQ handlers

After executing each instruction, the processor checks to see whether the interrupt pins are LOW and whether the interrupt disable bits in the CPSR are clear. As a result, IRQ or FIQ exceptions are generated only after the program counter has been updated. The processor stores (PC – 4) in lr_*mode*. This makes lr_*mode* point one instruction beyond the end of the instruction in which the exception occurred. When the handler has finished, execution must continue from the instruction prior to the one pointed to by lr_*mode*. The address to continue from is one word (four bytes) less than that in lr_*mode*, so the return instruction is:

```
SUBS        pc, lr, #4
```

The handler entry and exit code to stack the return address and pop it on return is:

```
SUB       lr,lr,#4
STMFD     sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

For exceptions that occur in Thumb state, the handler return instruction (`SUBS pc,lr,#4`) changes the program counter to the address of the next instruction to execute. Because the program counter is updated before the exception is taken, the next instruction is at (PC – 4). The value stored by the processor in lr_*mode* is therefore PC.

Returning from Prefetch Abort handlers

If the processor attempts to fetch an instruction from an illegal address, the instruction is flagged as invalid. Instructions already in the pipeline continue to execute until the invalid instruction is reached, at which point a Prefetch Abort is generated.

The exception handler loads the unmapped instruction into physical memory and uses the MMU, if there is one, to map the virtual memory location into the physical one. The handler must then return to retry the instruction that caused the exception. The instruction now loads and executes.

Because the program counter is not updated at the time the prefetch abort is issued, `lr_ABT` points to the instruction following the one that caused the exception. The handler must return to `lr_ABT - 4` with:

```
SUBS        pc,lr, #4
```

The handler entry and exit code to stack the return address and pop it on return is:

```
SUB       lr,lr,#4
STMFD     sp!,{reglist,lr}
;...
LDMFD     sp!,{reglist,pc}^
```

For exceptions that occur in Thumb state, the handler return instruction (`SUBS pc,lr,#4`) changes the program counter to the address of the aborted instruction. Because the program counter is not updated before the exception is taken, the aborted instruction is at (PC – 4). The value stored by the processor in lr_*mode* is therefore PC.

Returning from Data Abort handlers

When a load or store instruction tries to access memory, the program counter has been updated. The stored value of (PC – 4) in `lr_ABT` points to the second instruction beyond the address where the exception occurred. When the MMU, if present, has mapped the appropriate address into physical memory, the handler must return to the original, aborted instruction so that a second attempt can be made to execute it. The return address is therefore two words (eight bytes) less than that in `lr_ABT`, making the return instruction:

```
    SUBS        pc, lr, #8
```

The handler entry and exit code to stack the return address and pop it on return is:

```
    SUB         lr,lr,#8
    STMFD       sp!,{reglist,lr}
    ;...
    LDMFD       sp!,{reglist,pc}^
```

For exceptions that occur in Thumb state, the handler return instruction (`SUBS pc,lr,#8`) changes the program counter to the address of the aborted instruction. Because the program counter is updated before the exception is taken, the aborted instruction is at (PC – 6). The value stored by the processor in lr_*mode* is therefore (PC + 2).

6.4 Handling the Exception

Your top-level veneer routine must save the processor status and any required registers on the stack. You then have the following options for writing the exception handler:

- Write the whole exception handler in ARM code.
- Perform a `BX` (Branch and eXchange) to a Thumb code routine that handles the exception. The routine must return to an ARM code veneer in order to return from the exception, because the Thumb instruction set does not have the instructions required to restore cpsr from spsr.

This second strategy is shown in Figure 6-2. See Chapter 4 *Interworking ARM and Thumb* for details of how to combine ARM and Thumb code in this way.

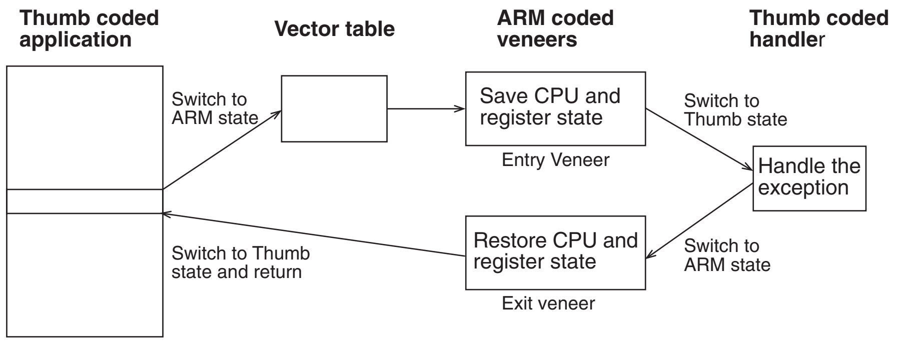


Figure 6-2 Handling an exception in ARM or Thumb state

6.5 Installing an exception handler

Any new exception handler must be installed in the vector table. When installation is complete, the new handler executes whenever the corresponding exception occurs.

Exception handlers can be installed in the following ways:

Branch instruction

This is the simplest way to reach the exception handler. Each entry in the vector table contains a branch to the required handler routine. However, this method does have a limitation. Because the branch instruction only has a range of 32MB relative to the PC, with some memory organizations the branch might be unable to reach the handler.

Load PC instruction

With this method, the program counter is forced directly to the handler address by:

1. Storing the absolute address of the handler in a suitable memory location (within 4KB of the vector address).
2. Placing an instruction in the vector that loads the program counter with the contents of the chosen memory location.

6.5.1 Installing the handlers at reset

If your application does not rely on the debugger or debug monitor to start program execution, you can load the vector table directly from your assembly language reset (or startup) code.

If your ROM is at location `0x0` in memory, you can have a branch statement for each vector at the start of your code. This could also include the FIQ handler if it is running directly from `0x1C` (see *Interrupt handlers* on page 6-27).

Example 6-2 shows code that sets up the vectors if they are located in ROM at address zero. You can substitute branch statements for the loads.

Example 6-2

```
Vector_Init_Block
                LDR    pc, Reset_Addr
                LDR    pc, Undefined_Addr
                LDR    pc, SWI_Addr
                LDR    pc, Prefetch_Addr
                LDR    pc, Abort_Addr
                NOP                     ;Reserved vector
                LDR    pc, IRQ_Addr
                LDR    pc, FIQ_Addr

Reset_Addr      DCD    Start_Boot
Undefined_Addr  DCD    Undefined_Handler
SWI_Addr        DCD    SWI_Handler
Prefetch_Addr   DCD    Prefetch_Handler
Abort_Addr      DCD    Abort_Handler
                DCD    0                 ;Reserved vector
IRQ_Addr        DCD    IRQ_Handler
FIQ_Addr        DCD    FIQ_Handler
```

You must have ROM at location `0x0` on reset. Your reset code can remap RAM to location `0x0`. Before doing this, it must copy the vectors (plus the FIQ handler if required) down from an area in ROM into the RAM.

In this case, you must use an `LDR pc` instruction to address the reset handler, so that the reset vector code can be position independent.

Example 6-3 copies down the vectors given in Example 6-2 on page 6-13 to the vector table in RAM.

Example 6-3

```
    MOV       r8, #0
    ADR       r9, Vector_Init_Block
    LDMIA     r9!,{r0-r7}          ;Copy the vectors (8 words)
    STMIA     r8!,{r0-r7}
    LDMIA     r9!,{r0-r7}          ;Copy the DCD'ed addresses
    STMIA     r8!,{r0-r7}          ;(8 words again)
```

Alternatively, you can use the scatter-loading mechanism to define the load and execution address of the vector table. In that case, the C library copies the vector table for you (see Chapter 2 *Embedded Software Development*).

6.5.2 Installing the handlers from C

Sometimes during development work it is necessary to install exception handlers into the vectors directly from the main application. As a result, the required instruction encoding must be written to the appropriate vector address. This can be done for both the branch and the load PC method of reaching the handler.

Branch method

The required instruction can be constructed as follows:

1. Take the address of the exception handler.
2. Subtract the address of the corresponding vector.
3. Subtract 0x8 to provide for prefetching.
4. Shift the result to the right by two to give a word offset, rather than a byte offset.
5. Test that the top eight bits of this are clear, to ensure that the result is only 24 bits long (because the offset for the branch is limited to this).
6. Logically OR this with `0xEA000000` (the opcode for the Branch instruction) to produce the value to be placed in the vector.

Example 6-4 shows a C function that implements this algorithm. It takes the following arguments:

- the address of the handler
- the address of the vector in which the handler is to be to installed.

The function can install the handler and return the original contents of the vector. This result can be used to create a chain of handlers for a particular exception. See *Chaining exception handlers* on page 6-41 for more details.

Example 6-4

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
/* Updates contents of 'vector' to contain branch instruction */
/* to reach 'routine' from 'vector'. Function return value is */
/* original contents of 'vector'.*/
/* NB: 'Routine' must be within range of 32MB from 'vector'.*/

{   unsigned vec, oldvec;
    vec = ((routine - (unsigned)vector - 0x8)>>2);
    if ((vec & 0xFF000000))
    {
        /* diagnose the fault */
        prinf ("Installation of Handler failed");
        exit (1);
    }
    vec = 0xEA000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

The following code calls this to install an IRQ handler:

```
unsigned *irqvec = (unsigned *)0x18;
Install_Handler ((unsigned)IRQHandler, irqvec);
```

In this case, the returned, original contents of the IRQ vector are discarded.

Load PC method

The required instruction can be constructed as follows:

1. Take the address of the word containing the address of the exception handler.
2. Subtract the address of the corresponding vector.
3. Subtract `0x8` to provide for prefetching.
4. Check that the result can be represented in 12 bits.
5. Logically OR this with `0xe59FF000` (the opcode for `LDR pc, [pc,#offset]`) to produce the value to be placed in the vector.
6. Put the address of the handler into the storage location.

Example 6-5 shows a C routine that implements this method.

Example 6-5

```
unsigned Install_Handler (unsigned location, unsigned *vector)

/* Updates contents of 'vector' to contain LDR pc, [pc, #offset] */
/* instruction to cause long branch to address in 'location'. */
/* Function return value is original contents of 'vector'. */

{   unsigned vec, oldvec;
    vec = ((unsigned)location - (unsigned)vector - 0x8) | 0xe59ff000;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

The following code calls this to install an IRQ handler:

```
unsigned *irqvec = (unsigned *)0x18;
static unsigned pIRQ_Handler = (unsigned)IRQ_handler
Install_Handler (&pIRQHandler, irqvec);
```

Again in this example the returned, original contents of the IRQ vector are discarded, but they could be used to create a chain of handlers. See *Chaining exception handlers* on page 6-41 for more information.

Note

If you are using a processor with separate instruction and data caches, such as StrongARM®, or ARM940T, you must ensure that cache coherence problems do not prevent the new contents of the vectors from being used.

The data cache (or at least the entries containing the modified vectors) must be cleaned to ensure the new vector contents are written to main memory. You must then flush the instruction cache to ensure that the new vector contents are read from main memory.

For details of cache clean and flush operations, see the technical reference manual for your target processor.

6.6 SWI handlers

When the SWI handler is entered, it must establish which SWI is being called. This information can be stored in bits 0-23 of the instruction itself, as shown in Figure 6-3, or passed in an integer register, usually one of r0-r3.

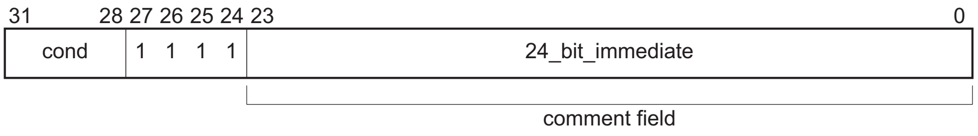


Figure 6-3 ARM SWI instruction

The top-level SWI handler can load the `SWI` instruction relative to the link register (`LDR swi, [lr, #-4]`). Do this in assembly language, or C/C++ inline or embedded assembler.

The handler must first load the `SWI` instruction that caused the exception into a register. At this point, `lr_SVC` holds the address of the instruction that follows the `SWI` instruction, so the `SWI` is loaded into the register (in this case r0) using:

```
LDR r0, [lr,#-4]
```

The handler can then examine the comment field bits, to determine the required operation. The SWI number is extracted by clearing the top eight bits of the opcode:

```
BIC r0, r0, #0xFF000000
```

Example 6-6 shows how you can put these instructions together to form a top-level SWI handler.

See *Determining the Processor State* on page 6-5 for an example of a handler that deals with both ARM-state and Thumb-state `SWI` instructions.

Example 6-6

```
    AREA TopLevelSwi, CODE, READONLY  ; Name this block of code.
    EXPORT     SWI_Handler
SWI_Handler
    STMFD      sp!,{r0-r12,lr}        ; Store registers.
    LDR        r0,[lr,#-4]            ; Calculate address of SWI instruction and load it into r0.
    BIC        r0,r0,#0xff000000      ; Mask off top 8 bits of instruction to give SWI number.
    ;
    ; Use value in r0 to determine which SWI routine to execute.
```

```
;
LDMFD       sp!, {r0-r12,pc}^    ; Restore registers and return.
END                              ; Mark end of this file.
```

6.6.1 SWI handlers in assembly language

The easiest way to call the handler for the requested SWI number is to use a jump table. If r0 contains the SWI number, the code in Example 6-7 can be inserted into the top-level handler given in Example 6-6 on page 6-18, following on from the `BIC` instruction.

Example 6-7 SWI jump table

```
    CMP    r0,#MaxSWI          ; Range check
    LDRLS  pc, [pc,r0,LSL #2]
    B      SWIOutOfRange
SWIJumpTable
    DCD    SWInum0
    DCD    SWInum1
                    ; DCD for each of other SWI routines
SWInum0             ; SWI number 0 code
    B    EndofSWI
SWInum1             ; SWI number 1 code
    B    EndofSWI
                    ; Rest of SWI handling code
                    ;
EndofSWI
                    ; Return execution to top level
                    ; SWI handler so as to restore
                    ; registers and return to program.
```

6.6.2 SWI handlers in C and assembly language

Although the top-level handler must always be written in ARM assembly language, the routines that handle each SWI can be written in either assembly language or in C. See *Using SWIs in Supervisor mode* on page 6-21 for a description of restrictions.

The top-level handler uses a `BL` (Branch with Link) instruction to jump to the appropriate C function. Because the SWI number is loaded into r0 by the assembly routine, this is passed to the C function as the first parameter (in accordance with the ARM Procedure Call Standard). The function can use this value in, for example, a `switch()` statement.

You can add the following line to the `SWI_Handler` routine in Example 6-6 on page 6-18:

```
    BL    C_SWI_Handler     ; Call C routine to handle the SWI
```

Example 6-8 shows how the C function can be implemented.

Example 6-8

```
void C_SWI_handler (unsigned number)
{ switch (number)
    {case 0 :                /* SWI number 0 code */
        break;
    case 1 :                 /* SWI number 1 code */
        break;
    :
    :
    default :                /* Unknown SWI - report error */
    }
}
```

The supervisor stack space might be limited, so avoid using functions that require a large amount of stack space.

You can pass values in and out of a SWI handler written in C, provided that the top-level handler passes the stack pointer value into the C function as the second parameter (in r1):

```
    MOV     r1, sp       ; Second parameter to C routine...
                         ; ...is pointer to register values.
    BL    C_SWI_Handler   ; Call C routine to handle the SWI
```

and the C function is updated to access it:

```
void C_SWI_handler(unsigned number, unsigned *reg)
```

The C function can now access the values contained in the registers at the time the `SWI` instruction was encountered in the main application code (see Figure 6-4 on page 6-21). It can read from them:

```
    value_in_reg_0 = reg [0];
    value_in_reg_1 = reg [1];
    value_in_reg_2 = reg [2];
    value_in_reg_3 = reg [3];
```

and also write back to them:

```
    reg [0] = updated_value_0;
    reg [1] = updated_value_1;
    reg [2] = updated_value_2;
    reg [3] = updated_value_3;
```

This causes the updated value to be written into the appropriate stack position, and then restored into the register by the top-level handler.

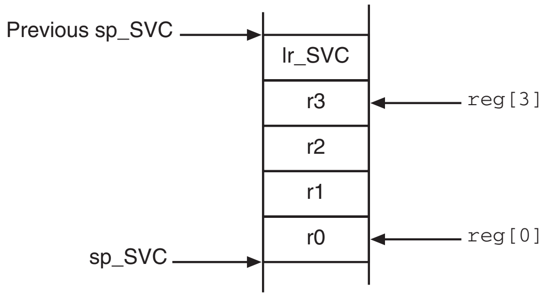


Figure 6-4 Accessing the supervisor stack

6.6.3 Using SWIs in Supervisor mode

When a `SWI` instruction is executed:

1. The processor enters Supervisor mode.
2. The CPSR is stored into `spsr_SVC`.
3. The return address is stored in `lr_SVC` (see *The processor response to an exception* on page 6-7).

If the processor is already in Supervisor mode, `lr_SVC` and `spsr_SVC` are corrupted.

If you call a SWI while in Supervisor mode you must store `lr_SVC` and `spsr_SVC` to ensure that the original values of the link register and the SPSR are not lost. For example, if the handler routine for a particular SWI number calls another SWI, you must ensure that the handler routine stores both `lr_SVC` and `spsr_SVC` on the stack. This ensures that each invocation of the handler saves the information required to return to the instruction following the `SWI` that invoked it. Example 6-9 shows how to do this.

Example 6-9 SWI Handler

```
    STMFD    sp!,{r0-r3,r12,lr}   ; Store registers.
    MOV      r1, sp               ; Set pointer to parameters.
    MRS      r0, spsr             ; Get spsr.
    STMFD    sp!, {r0}            ; Store spsr onto stack. This is only really needed in case of
                                  ; nested SWIs.

        ; the next two instructions only work for SWI calls from ARM state.
        ; See Example 6-18 on page 6-33 for a version that works for calls from either ARM or Thumb.

    LDR      r0,[lr,#-4]          ; Calculate address of SWI instruction and load it into r0.
    BIC      r0,r0,#0xFF000000    ; Mask off top 8 bits of instruction to give SWI number.

        ; r0 now contains SWI number
        ; r1 now contains pointer to stacked registers

    BL       C_SWI_Handler        ; Call C routine to handle the SWI.
    LDMFD    sp!, {r0}            ; Get spsr from stack.
    MSR      spsr_cf, r0          ; Restore spsr.
    LDMFD    sp!, {r0-r3,r12,pc}^ ; Restore registers and return.
```

Nested SWIs in C and C++

You can write nested SWIs in C or C++. Code generated by the ARM compiler stores and reloads `lr_SVC` as necessary.

6.6.4 Calling SWIs from an application

You can call a SWI from assembly language or C/C++.

In assembly language, set up any required register values and issue the relevant SWI. For example:

```
MOV    r0, #65    ; load r0 with the value 65
SWI    0x0        ; Call SWI 0x0 with parameter value in r0
```

The `SWI` instruction can be conditionally executed, as can almost all ARM instructions.

From C/C++, declare the SWI as an `__SWI` function, and call it. For example:

```
__swi(0) void my_swi(int);
.
.
.
my_swi(65);
```

This enables a SWI to be compiled inline, without additional calling overhead, provided that:

- any arguments are passed in r0-r3 only
- any results are returned in r0-r3 only.

The parameters are passed to the SWI as if the SWI were a real function call. However, if there are between two and four return values, you must tell the compiler that the return values are being returned in a structure, and use the `__value_in_regs` directive. This is because a **`struct`**-valued function is usually treated as if it were a **`void`** function whose first argument is the address where the result structure must be placed.

Example 6-10 and Example 6-11 show a SWI handler that provides SWI numbers `0x0`, `0x1`, `0x2` and `0x3`. SWIs `0x0` and `0x1` each take two integer parameters and return a single result. SWI `0x2` takes four parameters and returns a single result. SWI `0x3` takes four parameters and returns four results. This example is in *`Examples_directory`*`\SWI\main.c`. and *`Examples_directory`*`\SWI\swi.h`.

Example 6-10 main.c

```
#include <stdio.h>
#include "swi.h"

unsigned *swi_vec = (unsigned *)0x08;
extern void SWI_Handler(void);

int main( void )
{
    int result1, result2;
    struct four_results res_3;
    Install_Handler( (unsigned) SWI_Handler, swi_vec );
    printf("result1 = multiply_two(2,4) = %d\n", result1 = multiply_two(2,4));
    printf("result2 = multiply_two(3,6) = %d\n", result2 = multiply_two(3,6));
    printf("add_two( result1, result2 ) = %d\n", add_two( result1, result2 ));
    printf("add_multiply_two(2,4,3,6) = %d\n", add_multiply_two(2,4,3,6));
    res_3 = many_operations( 12, 4, 3, 1 );
    printf("res_3.a = %d\n", res_3.a );
    printf("res_3.b = %d\n", res_3.b );
    printf("res_3.c = %d\n", res_3.c );
    printf("res_3.d = %d\n", res_3.d );
    return 0;
}
```

Example 6-11 swi.h

```
__swi(0) int multiply_two(int, int);
__swi(1) int add_two(int, int);
__swi(2) int add_multiply_two(int, int, int, int);

struct four_results
```

```
{
    int a;
    int b;
    int c;
    int d;
};

__swi(3) __value_in_regs struct four_results
    many_operations(int, int, int, int);
```

6.6.5 Calling SWIs dynamically from an application

In some circumstances it can be necessary to call a SWI whose number is not known until runtime. This situation can occur, for example, when there are a number of related operations that can be performed on an object, and each operation has its own SWI. In such a case, the methods described above are not appropriate.

There are several ways of dealing with this, for example, you can:

- Construct the `SWI` instruction from the SWI number, store it somewhere, then execute it.
- Use a generic SWI that takes, as an extra argument, a code for the actual operation to be performed on its arguments. The generic SWI decodes the operation and performs it.

The second mechanism can be implemented in assembly language by passing the required operation number in a register, typically r0 or r12. You can then rewrite the SWI handler to act on the value in the appropriate register. Because some value has to be passed to the SWI in the comment field, it is possible for a combination of these two methods to be used.

For example, an operating system might make use of only a single `SWI` instruction and employ a register to pass the number of the required operation. This leaves the rest of the SWI space available for application-specific SWIs. You can use this method if the overhead of extracting the SWI number from the instruction is too great in a particular application. This is how the ARM (`0x123456`) and Thumb (`0xAB`) semihosted SWIs are implemented.

Example 6-12 on page 6-25 shows how `__swi` can be used to map a C function call onto a semihosting SWI. It is derived from *Examples_directory*`\embedded\embed\retarget.c`.

Example 6-12 Mapping a C function onto a semihosting SWI

```
#ifdef __thumb
/* Thumb Semihosting SWI */
#define SemiSWI 0xAB
#else
/* ARM Semihosting SWI */
#define SemiSWI 0x123456
#endif

/* Semihosting SWI to write a character */
__swi(SemiSWI) void Semihosting(unsigned op, char *c);
#define WriteC(c) Semihosting (0x3,c)

void write_a_character(int ch)
{
    char tempch = ch;
    WriteC( &tempch );
}
```

A mechanism is included in the compiler to support the use of r12 to pass the value of the required operation. Under the ARM Procedure Call Standard, r12 is the ip register and has a dedicated role only during function call. At other times, you can use it as a scratch register. The arguments to the generic SWI are passed in registers r0-r3 and values are optionally returned in r0-r3 as described earlier. The operation number passed in r12 can be, but is not require to be, the number of the SWI to be called by the generic SWI.

Example 6-13 shows a C fragment that uses a generic, or *indirect* SWI.

Example 6-13

```
__swi_indirect(0x80)
    unsigned SWI_ManipulateObject(unsigned operationNumber,
                                  unsigned object,unsigned parameter);

unsigned DoSelectedManipulation(unsigned object,
                                unsigned parameter, unsigned operation)
{ return SWI_ManipulateObject(operation, object, parameter);
}
```

This produces the following code:

```
DoSelectedManipulation PROC
        STMFD    sp!,{r3,lr}
        MOV      r12,r2
        SWI      0x80
        LDMFD    sp!,{r3,pc}
        ENDP
```

It is also possible to pass the SWI number in r0 from C using the `__swi` mechanism. For example, if `SWI 0x0` is used as the generic SWI and operation 0 is a character read and operation 1 a character write, you can set up the following:

```
__swi (0) char __ReadCharacter (unsigned op);
__swi (0) void __WriteCharacter (unsigned op, char c);
```

These can be used in a more reader-friendly fashion by defining the following:

```
#define ReadCharacter () __ReadCharacter (0);
#define WriteCharacter (c) __WriteCharacter (1, c);
```

However, if you use r0 in this way, only three registers are available for passing parameters to the SWI. Usually, if you have to pass more parameters to a subroutine in addition to r0-r3, you can do this using the stack. However, stacked parameters are not easily accessible to a SWI handler, because they typically exist on the User mode stack rather than the supervisor stack employed by the SWI handler.

Alternatively, one of the registers (typically r1) can be used to point to a block of memory storing the other parameters.

6.7 Interrupt handlers

The ARM processor has two levels of external interrupt, FIQ and IRQ, both of which are level-sensitive active LOW signals into the core. For an interrupt to be taken, the appropriate disable bit in the CPSR must be clear.

FIQs have higher priority than IRQs in the following ways:

- FIQs are serviced first when multiple interrupts occur.
- Servicing a FIQ causes IRQs to be disabled, preventing them from being serviced until after the FIQ handler has re-enabled them. This is usually done by restoring the CPSR from the SPSR at the end of the handler.

The FIQ vector is the last entry in the vector table (at address `0x1C`) so that the FIQ handler can be placed directly at the vector location and run sequentially from that address. This removes the requirement for a branch and its associated delays, and also means that if the system has a cache, the vector table and FIQ handler might all be locked down in one block within it. This is important because FIQs are designed to service interrupts as quickly as possible. The five extra FIQ mode banked registers enable status to be held between calls to the handler, again increasing execution speed.

Note

An interrupt handler must contain code to clear the source of the interrupt.

6.7.1 Simple interrupt handlers in C

You can write simple C interrupt handlers by using the `__irq` function declaration keyword. You can use the `__irq` keyword both for simple one-level interrupt handlers, and interrupt handlers that call subroutines. However, you cannot use the `__irq` keyword for *reentrant* interrupt handlers, because it does not cause the SPSR to be saved or restored. In this context, reentrant means that the handler re-enables interrupts, and can itself be interrupted. See *Reentrant interrupt handlers* on page 6-29 for more information.

The `__irq` keyword:

- preserves all ATPCS corruptible registers
- preserves all other registers (excluding the floating-point registers) used by the function
- exits the function by setting the program counter to (lr – 4) and restoring the CPSR to its original value.

If the function calls a subroutine, `__irq` preserves the link register for the interrupt mode in addition to preserving the other corruptible registers. See *Calling subroutines from interrupt handlers* for more information.

Note

C interrupt handlers cannot be produced in this way using when compiling Thumb C code. The `__irq` keyword is faulted when compiling in Thumb mode, and the processor is always switched to ARM state when an interrupt, or any other exception, occurs.

However, the subroutine called by an `__irq` function can be compiled for Thumb, with interworking enabled. See Chapter 4 *Interworking ARM and Thumb* for more information on interworking.

Calling subroutines from interrupt handlers

If you call subroutines from your top-level interrupt handler, the `__irq` keyword also restores the value of `lr_IRQ` from the stack so that it can be used by a `SUBS` instruction to return to the correct address after the interrupt has been handled.

Example 6-14 shows how this works. The top level interrupt handler reads the value of a memory-mapped interrupt controller base address at `0x80000000`. If the value of the address is 1, the top-level handler branches to a handler written in C.

Example 6-14

```
__irq void IRQHandler (void)
{
    volatile unsigned int *base = (unsigned int *) 0x80000000;

    if (*base == 1)          // which interrupt was it?
    {
        C_int_handler();     // process the interrupt
    }
    *(base+1) = 0;           // clear the interrupt
}
```

Compiled with `armcc`, Example 6-14 produces the following code:

```
IRQHandler PROC
        STMFD    sp!,{r0-r4,r12,lr}
        MOV      r4,#0x80000000
        LDR      r0,[r4,#0]
        SUB      sp,sp,#4
        CMP      r0,#1
        BLEQ     C_int_handler
        MOV      r0,#0
        STR      r0,[r4,#4]
        ADD      sp,sp,#4
        LDMFD    sp!,{r0-r4,r12,lr}
        SUBS     pc,lr,#4
        ENDP
```

Compare this with the result when the `__irq` keyword is not used:

```
IRQHandler PROC
        STMFD    sp!,{r4,lr}
        MOV      r4,#0x80000000
        LDR      r0,[r4,#0]
        CMP      r0,#1
        BLEQ     C_int_handler
        MOV      r0,#0
        STR      r0,[r4,#4]
        LDMFD    sp!,{r4,pc}
        ENDP
```

6.7.2 Reentrant interrupt handlers

If an interrupt handler re-enables interrupts, then calls a subroutine, and another interrupt occurs, the return address of the subroutine (stored in `lr_IRQ`) is corrupted when the second IRQ is taken. Using the `__irq` keyword in C does not cause the SPSR to be saved and restored, as required by reentrant interrupt handlers, so you must write your top level interrupt handler in assembly language.

A reentrant interrupt handler must save the IRQ state, switch processor modes, and save the state for the new processor mode before branching to a nested subroutine or C function.

In ARMv4 or later you can switch to System mode. System mode uses the User mode registers, and enables privileged access that might be required by your exception handler. See *System mode* on page 6-43 for more information. In ARM architectures prior to ARM architecture v4 you must switch to Supervisor mode instead.

Note

This method works for both IRQ and FIQ interrupts. However, because FIQ interrupts are meant to be serviced as quickly as possible there is normally only one interrupt source, so it might not be necessary to provide for reentrancy.

The steps required to safely re-enable interrupts in an IRQ handler are:

1. Construct return address and save on the IRQ stack.
2. Save the work registers and `spsr_IRQ`.

3. Clear the source of the interrupt.
4. Switch to System mode and re-enable interrupts.
5. Save User mode link register and non callee-saved registers.
6. Call the C interrupt handler function.
7. When the C interrupt handler returns, restore User mode registers and disable interrupts.
8. Switch to IRQ mode, disabling interrupts.
9. Restore work registers and `spsr_IRQ`.
10. Return from the IRQ.

Example 6-15 shows how this works for System mode. Registers r12 and r14 are used as temporary work registers after `lr_IRQ` is pushed on the stack.

Example 6-15

```
    AREA INTERRUPT, CODE, READONLY
    IMPORT C_irq_handler
IRQ
    SUB     lr, lr, #4        ; construct the return address
    STMFD   sp!, {lr}         ; and push the adjusted lr_IRQ
    MRS     r14, SPSR         ; copy spsr_IRQ to r14
    STMFD   sp!, {r12, r14}   ; save work regs and spsr_IRQ

    ; Add instructions to clear the interrupt here
    ; then re-enable interrupts.

    MSR     CPSR_c, #0x1F     ; switch to SYS mode, FIQ and IRQ
                              ; enabled. USR mode registers
                              ; are now current.
    STMFD  sp!, {r0-r3, lr}   ; save lr_USR and non-callee
                              ; saved registers
    BL      C_irq_handler     ; branch to C IRQ handler.
    LDMFD   sp!, {r0-r3, lr}  ; restore registers
    MSR     CPSR_c, #0x92     ; switch to IRQ mode and disable
                              ; IRQs. FIQ is still enabled.

    LDMFD   sp!, {r12, r14}   ; restore work regs and spsr_IRQ
    MSR     SPSR_cf, r14
    LDMFD   sp!, {pc}^        ; return from IRQ.
    END
```

This example assumes that FIQ remains permanently enables.

6.7.3 Example interrupt handlers in assembly language

Interrupt handlers are often written in assembly language to ensure that they execute quickly. The following sections give some examples:

- *Single-channel DMA transfer*
- *Dual-channel DMA transfer* on page 6-32
- *Interrupt prioritization* on page 6-33
- *Context switch* on page 6-34.

Single-channel DMA transfer

Example 6-16 shows an interrupt handler that performs interrupt driven I/O to memory transfers (soft DMA). The code is an FIQ handler. It uses the banked FIQ registers to maintain state between interrupts. This code is best situated at location `0x1C`.

In the example code:

r8	Points to the base address of the I/O device that data is read from.
IOData	Is the offset from the base address to the 32-bit data register that is read. Reading this register clears the interrupt.
r9	Points to the memory location to where that data is being transferred.
r10	Points to the last address to transfer to.

The entire sequence for handling a normal transfer is four instructions. Code situated after the conditional return is used to signal that the transfer is complete.

Example 6-16

```
    LDR     r11, [r8, #IOData]      ; Load port data from the IO
                                    ; device.
    STR     r11, [r9], #4           ; Store it to memory: update
                                    ; the pointer.
    CMP     r9, r10                 ; Reached the end ?
    SUBLSS  pc, lr, #4              ; No, so return.
                                    ; Insert transfer complete
                                    ; code here.
```

Byte transfers can be made by replacing the load instructions with load byte instructions. Transfers from memory to an I/O device are made by swapping the addressing modes between the load instruction and the store instruction.

Dual-channel DMA transfer

Example 6-17 is similar to Example 6-16 on page 6-31, except that there are two channels being handled. The code is an FIQ handler. It uses the banked FIQ registers to maintain state between interrupts. It is best situated at location 0x1c.

In the example code:

r8	Points to the base address of the I/O device from which data is read.
IOStat	Is the offset from the base address to a register indicating which of two ports caused the interrupt.
IOPort1Active	Is a bit mask indicating if the first port caused the interrupt (otherwise it is assumed that the second port caused the interrupt).
IOPort1, IOPort2	Are offsets to the two data registers to be read. Reading a data register clears the interrupt for the corresponding port.
r9	Points to the memory location to which data from the first port is being transferred.
r10	Points to the memory location to which data from the second port is being transferred.
r11, r12	Point to the last address to transfer to (r11 for the first port, r12 for the second).

The entire sequence to handle a normal transfer takes nine instructions. Code situated after the conditional return is used to signal that the transfer is complete.

Example 6-17

```
    LDR     r13, [r8, #IOStat]      ; Load status register to find which port
                                    ; caused the interrupt.
    TST     r13, #IOPort1Active
    LDREQ   r13, [r8, #IOPort1]     ; Load port 1 data.
    LDRNE   r13, [r8, #IOPort2]     ; Load port 2 data.
    STREQ   r13, [r9], #4           ; Store to buffer 1.
    STRNE   r13, [r10], #4          ; Store to buffer 2.
    CMP     r9, r11                 ; Reached the end?
    CMPLE   r10, r12                ; On either channel?
    SUBNES  pc, lr, #4              ; Return
                            ; Insert transfer complete code here.
```

Byte transfers can be made by replacing the load instructions with load byte instructions. Transfers from memory to an I/O device are made by swapping the addressing modes between the conditional load instructions and the conditional store instructions.

Interrupt prioritization

Example 6-18 dispatches up to 32 interrupt sources to their appropriate handler routines. Because it is designed for use with the normal interrupt vector (IRQ), it is branched to from location `0x18`.

External hardware is used to prioritize the interrupt and present the high-priority active interrupt in an I/O register.

In the example code:

IntBase	Holds the base address of the interrupt controller.
IntLevel	Holds the offset of the register containing the highest-priority active interrupt.
r13	Is assumed to point to a small full descending stack.

Interrupts are enabled after ten instructions, including the branch to this code.

The specific handler for each interrupt is entered after a further two instructions (with all registers preserved on the stack).

In addition, the last three instructions of each handler are executed with interrupts turned off again, so that the SPSR can be safely recovered from the stack.

Note

Application Note 30: *Software Prioritization of Interrupts* describes multiple-source prioritization of interrupts using software, as opposed to using hardware as described here.

Example 6-18

```
    ; first save the critical state
    SUB     lr, lr, #4              ; Adjust the return address
                                    ; before we save it.
    STMFD   sp!, {lr}               ; Stack return address
    MRS     r14, SPSR               ; get the SPSR ...
    STMFD   sp!, {r12, r14}         ; ... and stack that plus a
                                    ; working register too.
                                    ; Now get the priority level of the
                                    ; highest priority active interrupt.
    MOV     r12, #IntBase           ; Get the interrupt controller's
                                    ; base address.
    LDR     r12, [r12, #IntLevel]   ; Get the interrupt level (0 to 31).

    ; Now read-modify-write the CPSR to enable interrupts.

    MRS     r14, CPSR               ; Read the status register.
    BIC     r14, r14, #0x80         ; Clear the I bit
                                    ; (use 0x40 for the F bit).
    MSR     CPSR_c, r14             ; Write it back to re-enable
                                    ; interrupts and
    LDR     pc, [pc, r12, LSL #2]   ; jump to the correct handler.
                                    ; PC base address points to this
                                    ; instruction + 8
    NOP                             ; pad so the PC indexes this table.

                                    ; Table of handler start addresses
    DCD     Priority0Handler
    DCD     Priority1Handler
    DCD     Priority2Handler
; ...
    Priority0Handler
    STMFD   sp!, {r0 - r11}         ; Save other working registers.
                                    ; Insert handler code here.
; ...
    LDMFD   sp!, {r0 - r11}         ; Restore working registers (not r12).

    ; Now read-modify-write the CPSR to disable interrupts.
    MRS     r12, CPSR               ; Read the status register.
    ORR     r12, r12, #0x80         ; Set the I bit
                                    ; (use 0x40 for the F bit).
    MSR     CPSR_c, r12             ; Write it back to disable interrupts.

    ; Now that interrupt disabled, can safely restore SPSR then return.
    LDMFD   sp!, {r12, r14}         ; Restore r12 and get SPSR.
    MSR     SPSR_csxf, r14          ; Restore status register from r14.
    LDMFD   sp!, {pc}^              ; Return from handler.
Priority1Handler
; ...
```

Context switch

Example 6-19 on page 6-35 performs a context switch on the User mode process. The code is based around a list of pointers to *Process Control Blocks* (PCBs) of processes that are ready to run.

Figure 6-5 shows the layout of the PCBs that the example expects.

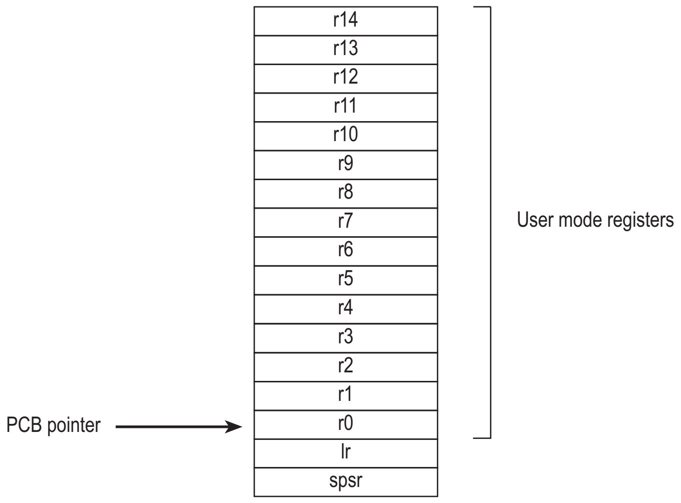


Figure 6-5 PCB layout

The pointer to the PCB of the next process to run is pointed to by r12, and the end of the list has a zero pointer. Register r13 is a pointer to the PCB, and is preserved between time slices, so that on entry it points to the PCB of the currently running process.

Example 6-19

```
STMIA   r13, {r0 - r14}^       ; Dump user registers above r13.
MRS     r0, SPSR               ; Pick up the user status
STMDB   r13, {r0, lr}          ; and dump with return address below.
LDR     r13, [r12], #4         ; Load next process info pointer.
CMP     r13, #0                ; If it is zero, it is invalid
LDMNEDB r13, {r0, lr}          ; Pick up status and return address.
MSRNE   SPSR_cxsf, r0          ; Restore the status.
LDMNEIA r13, {r0 - r14}^       ; Get the rest of the registers
NOP
SUBNES  pc, lr, #4             ; and return and restore CPSR.
                ; Insert "no next process code" here.
```

6.8 Reset handlers

The operations carried out by the Reset handler depend on the system for which the software is being developed. For example, it might:

- Set up exception vectors. See *Installing an exception handler* on page 6-13 for details.
- Initialize stacks and registers.
- Initialize the memory system, if using an MMU.
- Initialize any critical I/O devices.
- Enable interrupts.
- Change processor mode and/or state.
- Initialize variables required by C and call the main application.

See Chapter 2 *Embedded Software Development* for more information.

6.9 Undefined Instruction handlers

Instructions that are not recognized by the processor are offered to any coprocessors attached to the system. If the instruction remains unrecognized, an Undefined Instruction exception is generated. It might be the case that the instruction is intended for a coprocessor, but that the relevant coprocessor, for example a Floating-Point Accelerator, is not attached to the system. However, a software emulator for such a coprocessor might be available.

Such an emulator must:

1. Attach itself to the Undefined Instruction vector and store the old contents.
2. Examine the undefined instruction to see if it has to be emulated. This is similar to the way in which a SWI handler extracts the number of a SWI, but rather than extracting the bottom 24 bits, the emulator must extract bits 27-24.

   These bits determine whether the instruction is a coprocessor operation in the following way:

   - If bits 27 to 24 = b1110 or b110x, the instruction is a coprocessor instruction.
   - If bits 8-11 show that this coprocessor emulator has to handle the instruction, the emulator must process the instruction and return to the user program.
3. Otherwise the emulator must pass the exception onto the original handler (or the next emulator in the chain) using the vector stored when the emulator was installed.

When a chain of emulators is exhausted, no further processing of the instruction can take place, so the Undefined Instruction handler must report an error and quit. See *Chaining exception handlers* on page 6-41 for more information.

Note

The Thumb instruction set does not have coprocessor instructions, so there is no requirement for the Undefined Instruction handler to emulate such instructions.

6.10 Prefetch Abort handler

If the system has no MMU, the Prefetch Abort handler can report the error and quit. Otherwise the address that caused the abort must be restored into physical memory. `lr_ABT` points to the instruction at the address following the one that caused the abort, so the address to be restored is at `lr_ABT - 4`. The virtual memory fault for that address can be dealt with and the instruction fetch retried. The handler therefore returns to the same instruction rather than the following one, for example:

```
    SUBS    pc,lr,#4
```

6.11 Data Abort handler

If there is no MMU, the Data Abort handler must report the error and quit. If there is an MMU, the handler must deal with the virtual memory fault.

The instruction that caused the abort is at `lr_ABT - 8` because `lr_ABT` points two instructions beyond the instruction that caused the abort.

The following types of instruction can cause this abort:

Single Register Load or Store (LDR or STR)

The response depends on the processor type:

- If the abort takes place on an ARM6-based processor:
  - If the processor is in early abort mode and writeback was requested, the address register has not been updated.
  - If the processor is in late abort mode and writeback was requested, the address register has been updated. The change must be undone.
- If the abort takes place on an ARM7-based processor, including the ARM7TDMI, the address register has been updated and the change must be undone.
- If the abort takes place on an ARM9™, ARM10™, or StrongARM-based processor, the address is restored by the processor to the value it had before the instruction started. No further action is required to undo the change.

Swap (SWP) There is no address register update involved with this instruction.

Load Multiple or Store Multiple (LDM or STM)

The response depends on the processor type:

- If the abort takes place on an ARM6-based processor or ARM7-based processor, and writeback is enabled, the base register is updated as if the whole transfer had taken place.

  In the case of an `LDM` with the base register in the register list, the processor replaces the overwritten value with the modified base value so that recovery is possible. The original base address can then be recalculated using the number of registers involved.
- If the abort takes place on an ARM9, ARM10, or StrongARM-based processor and writeback is enabled, the base register is restored to the value it had before the instruction started.

In each of the three cases the MMU can load the required virtual memory into physical memory. The MMU *Fault Address Register* (FAR) contains the address that caused the abort. When this is done, the handler can return and try to execute the instruction again.

You can find example Data Abort handler code in `Examples_directory\databort`.

6.12 Chaining exception handlers

In some situations there can be several different sources of a particular exception. For example:

- Angel uses an Undefined Instruction to implement breakpoints. However, Undefined Instruction exceptions also occur when a coprocessor instruction is executed, and no coprocessor is present.
- Angel uses a SWI for various purposes, such as entering Supervisor mode from User mode, and supporting semihosting requests during development. However, an RTOS or an application might also implement some SWIs.

In such situations the following approaches that can be taken to extend the exception handling code:

- *A single extended handler*
- *Several chained handlers*.

6.12.1 A single extended handler

In some circumstances it is possible to extend the code in the exception handler to work out what the source of the exception was, and then directly call the appropriate code. In this case, you are modifying the source code for the exception handler.

Angel has been written to make this approach simple. Angel decodes SWIs and Undefined Instructions, and the Angel exception handlers can be extended to deal with non-Angel SWIs and Undefined Instructions.

However, this approach is only useful if all the sources of an exception are known when the single exception handler is written.

6.12.2 Several chained handlers

Some circumstances require more than a single handler. Consider the situation in which a standard Angel debugger is executing, and a standalone user application (or RTOS) which wants to support some additional SWIs is then downloaded. The newly loaded application might have its own entirely independent exception handler that it wants to install, but which cannot replace the Angel handler.

In this case the address of the old handler must be noted so that the new handler is able to call the old handler if it discovers that the source of the exception is not a source it can deal with. For example, an RTOS SWI handler would call the Angel SWI handler on discovering that the SWI was not an RTOS SWI, so that the Angel SWI handler gets a chance to process it.

This approach can be extended to any number of levels to build a chain of handlers. Although code that takes this approach enables each handler to be entirely independent, it is less efficient than code that uses a single handler, or at least it becomes less efficient the further down the chain of handlers it has to go.

Both routines given in *Installing the handlers from C* on page 6-15 return the old contents of the vector. This value can be decoded to give:

The offset for a branch instruction

This can be used to calculate the location of the original handler and enable a new branch instruction to be constructed and stored at a suitable place in memory. If the replacement handler fails to handle the exception, it can branch to the constructed branch instruction, which in turn branches to the original handler.

The location used to store the address of the original handler

If the application handler fails to handle the exception, it has to load the program counter from that location.

In most cases, such calculations are not necessary because information on the debug monitor or RTOS handlers is available to you. If so, the instructions required to chain in the next handler can be hard-coded into the application. The last section of the handler must check that the cause of the exception has been handled. If it has, the handler can return to the application. If not, it must call the next handler in the chain.

Note

When chaining in a handler before a debug monitor handler, you must remove the chain when the monitor is removed from the system, then directly install the application handler.

6.13 System mode

The ARM Architecture defines a User mode that has 15 general purpose registers, a PC, and a CPSR. In addition to this mode there are five privileged processor modes, each of which have an SPSR and a number of registers that replace some of the 15 User mode general purpose registers.

Note

This section only applies to processors that implement ARM architectures v4, v4T and later.

When a processor exception occurs, the current program counter is copied into the link register for the exception mode, and the CPSR is copied into the SPSR for the exception mode. The CPSR is then altered in an exception-dependent way, and the program counter is set to an exception-defined address to start the exception handler.

The ARM subroutine call instruction (`BL`) copies the return address into r14 before changing the program counter, so the subroutine return instruction moves r14 to PC (`MOV pc,lr`).

Together these actions imply that ARM modes that handle exceptions must ensure that another exception of the same type cannot occur if they call subroutines, because the subroutine return address is overwritten with the exception return address.

(In earlier versions of the ARM architecture, this problem has been solved by either carefully avoiding subroutine calls in exception code, or changing from the privileged mode to User mode. The first solution is often too restrictive, and the second means the task might not have the privileged access it requires to run correctly.)

ARM architecture v4 and later provide a processor mode called *system* mode, to overcome this problem. System mode is a privileged processor mode that shares the User mode registers. Privileged mode tasks can run in this mode, and exceptions no longer overwrite the link register.

Note

System mode cannot be entered by an exception. The exception handlers modify the CPSR to enter System mode. See *Reentrant interrupt handlers* on page 6-29 for an example.

Chapter 7
Debug Communications Channel

This chapter explains how to use the *Debug Communications Channel* (DCC). It contains the following sections:

7.1 About the Debug Communications Channel

The EmbeddedICE logic in ARM cores such as ARM7TDMI and ARM9TDMI® contains a debug communications channel. This enables data to be passed between the target and the host debugger using the JTAG port and a protocol converter such as Multi-ICE®, without stopping the program flow or entering debug state. This chapter describes how the debug communications channel can be accessed by a program running on the target, and by the host debugger.

RVCT provides access to the debug communications channel through Multi-ICE semihosting.

7.2 Target transfer of data

The debug communications channel is accessed by the target as coprocessor 14 on the ARM core using the ARM instructions `MCR` and `MRC`.

Two registers are provided to transfer data:

Comms data read register

A 32-bit wide register used to receive data from the debugger. The following instruction returns the read register value in `Rd`:

```
MRC p14, 0, Rd, c1, c0
```

Comms data write register

A 32-bit wide register used to send data to the debugger. The following instruction writes the value in `Rn` to the write register:

```
MCR p14, 0, Rn, c1, c0
```

Note

Refer to the *ARM10 Technical Reference Manual* for information on accessing DCC registers for the ARM 10 core. The instructions used, positions of the status bits, and interpretation of the status bits are different for processors later than ARM9.

7.3 Polled debug communications

In addition to the comms data read and write registers, a comms data control register is provided by the debug communications channel.

The following instruction returns the control register value in Rd:

```
MRC p14, 0, Rd, c0, c0
```

Two bits in this control register provide synchronized handshaking between the target and the host debugger:

Bit 1 (W bit) Denotes whether the comms data write register is free (from the target point of view):

W = 0	New data can be written by the target application.
W = 1	The host debugger can scan new data out of the write register.

Bit 0 (R bit) Denotes whether there is new data in the comms data read register (from the target point of view):

R = 1	New data is available to be read by the target application.
R = 0	The host debugger can scan new data into the read register.

Note

The debugger cannot use coprocessor 14 to access the debug communications channel directly, because this has no meaning to the debugger. Instead, the debugger can read from and write to the debug communications channel registers using the scan chain. The debug communications channel data and control registers are mapped into addresses in the EmbeddedICE logic, see *Viewing EmbeddedICE logic registers*.

7.3.1 Viewing EmbeddedICE logic registers

To view the EmbeddedICE logic registers, see the *Multi-ICE v2.2 User Guide*, or later version.

7.3.2 Target to debugger communication

This is the sequence of events for an application running on the ARM core to communicate with a debugger running on the host:

1. The target application checks if the debug communications channel write register is free for use. It does this using the `MRC` instruction to read the debug communications channel control register to check that the W bit is clear.
2. If the W bit is clear, the debug communication write register is clear and the application writes a word to it using the `MCR` instruction to coprocessor 14. The action of writing to the register automatically sets the W bit. If the W bit is set, the debug communication write register has not been emptied by the debugger. If the application has to send another word, it must poll the W bit until it is clear.
3. The debugger polls the debug communication control register through scan chain 2. If the debugger sees that the W bit is set, it can read the debug communications channel data register to read the message sent by the application. The process of reading the data automatically clears the W bit in the debug communication control register.

Example 7-1 shows how this works. The example code is available in *`Examples_directory`*`\dcc\outchan.s`.

Example 7-1

```
     AREA  OutChannel, CODE, READONLY
     ENTRY
     MOV   r1,#3          ; Number of words to send
     ADR   r2, outdata    ; Address of data to send
pollout
     MRC   p14,0,r0,c0,c0 ; Read control register
     TST   r0, #2
     BNE   pollout        ; if W set, register still full
write
     LDR   r3,[r2],#4     ; Read word from outdata
                          ; into r3 and update the pointer
     MCR   p14,0,r3,c1,c0 ; Write word from r3
     SUBS  r1,r1,#1       ; Update counter
     BNE   pollout        ; Loop if more words to be written
     MOV   r0, #0x18      ; Angel_SWIreason_ReportException
     LDR   r1, =0x20026   ; ADP_Stopped_ApplicationExit
     SWI   0x123456       ; ARM semihosting SWI
outdata
     DCB "Hello there!"
     END
```

To execute the example:

1. Assemble `outchan.s`:

   `armasm -g outchan.s`

2. Link the output object:

   `armlink outchan.o -o outchan.axf`

   The link step creates the executable file `outchan.axf`

3. Load the and execute the image. See your debugger documentation for details.

7.3.3 Debugger to target communication

This is the sequence of events for message transfer from a debugger running on the host to the application running on the core:

1. The debugger polls the debug communication control register `R` bit. If the `R` bit is clear, the debug communication read register is clear and data can be written there for the target application to read.

2. The debugger scans the data into the debug communication read register via scan chain 2. The `R` bit in the debug communication control register is automatically set by this.

3. The target application polls the `R` bit in the debug communication control register. If it is set, there is data in the debug communication read register that can be read by the application, using the `MRC` instruction to read from coprocessor 14. The `R` bit is cleared as part of the read instruction.

   The following piece of target application code, supplied in file *`Examples_directory`*`\dcc\inchan.s`, shows this in action:

```
     AREA  InChannel, CODE, READONLY
     ENTRY
     MOV   r1,#3          ; Number of words to read
     LDR   r2, =indata    ; Address to store data read
pollin
     MRC   p14,0,r0,c0,c0 ; Read control register
     TST   r0, #1
     BEQ   pollin         ; If R bit clear then loop
read
     MRC   p14,0,r3,c1,c0 ; read word into r3
     STR   r3,[r2],#4     ; Store to memory and
                          ; update pointer
     SUBS  r1,r1,#1       ; Update counter
     BNE   pollin         ; Loop if more words to read
     MOV   r0, #0x18      ; Angel_SWIreason_ReportException
     LDR   r1, =0x20026   ; ADP_Stopped_ApplicationExit
```

```
        SWI    0x123456       ; ARM semihosting SWI

        AREA  Storage, DATA, READWRITE
indata
        DCB   "Duffmessage#"
        END
```

4. Create an input file on the host containing, for example, `And goodbye!`.
5. Assemble and link this code using the following commands:

```
armasm -g inchan.s
armlink inchan.o -o inchan.axf
```

You have created an executable image in a file called `inchan`. To load and execute the image see your debugger documentation.

7.4 Interrupt-driven debug communications

The examples given in *Polled debug communications* on page 7-4 demonstrate polling the DCC. You can convert these to interrupt-driven examples by connecting up `COMMRX` and `COMMTX` signals from the Embedded ICE logic to your interrupt controller.

The read and write code given above could then be moved into an interrupt handler.

See *Interrupt handlers* on page 6-27 for information on writing interrupt handlers.

7.5 Access from Thumb state

Because the Thumb instruction set does not contain coprocessor instructions, you cannot use the debug communications channel while the core is in Thumb state.

There are three possible ways around this:

- You can write each polling routine in a SWI handler, which can then be executed while in either ARM or Thumb state. Entering the SWI handler immediately puts the core into ARM state where the coprocessor instructions are available. See Chapter 6 *Handling Processor Exceptions* for more information on SWIs.
- Thumb code can make interworking calls to ARM subroutines which implement the polling. See Chapter 4 *Interworking ARM and Thumb* for more information on mixing ARM and Thumb code.
- Use interrupt-driven communication rather than polled communication. The interrupt handler would be written in ARM instructions, so the coprocessor instructions can be accessed directly.

7.6 Semihosting

You can use the debug communications channel for semihosting if you are using Multi-ICE with `$semihosting_enabled=2`. See the *Multi-ICE v2.2 User Guide* for more information.

Glossary

American National Standards Institute (ANSI)	An organization that specifies standards for, among other things, computer software.
ANSI	*See* American National Standards Institute.
Angel	Angel is a debug monitor that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.
ARMulator	RealView ARMulator ISS is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
ARM-Thumb Procedure Call Standard (ATPCS)	*ARM-Thumb Procedure Call Standard* defines how registers and the stack is used for subroutine calls.
ATPCS	*See* ARM-Thumb Procedure Call Standard.
Big-Endian	Memory organization where the least significant byte of a word is at a higher address than the most significant byte.
Canonical Frame Address (CFA)	In DWARF 2, this is an address on the stack specifying where the call frame of an interrupted function is located.
CFA	*See* Canonical Frame Address.

Coprocessor	An additional processor which is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.
Double word	A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
DWARF	Debug With Arbitrary Record Format
EC++	A variant of C++ designed to be used for embedded applications.
ELF	Executable Linkable Format.
Embedded	Applications that are developed as firmware. Assembler functions placed out-of-line in a C or C++ program. *See also* Inline.
Execution view	The address of regions and sections after the image has been loaded into memory and started execution.
Flash memory	Non-volatile memory that is often used to hold application code.
Halfword	A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
Heap	The portion of computer memory that can be used for creating new variables.
ICE	In Circuit Emulator.
Image	An executable file which has been loaded onto a processor for execution. A binary execution file loaded onto a processor and given a thread of execution. An image might have multiple threads. An image is related to the processor on which its default thread runs.
Inline	Functions that are repeated in code each time they are used rather than having a common subroutine. Assembler code placed within a C or C++ program. *See also* Output section, and Embedded.
Input section	Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts. *See also* Output sections.
International Standards Organization (ISO)	An organization that specifies standards for, among other things, computer software.
Interworking	Producing an application that uses both ARM and Thumb code.
ISO	*See* International Standards Organization.

Library	A collection of assembler or compiler output objects grouped together into a single repository.
Linker	Software which produces a single image from one or more source assembler or compiler output objects.
Little-endian	Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also *Big-endian*.
Local	An object that is only accessible to the subroutine that created it.
Load view	The address of regions and sections when the image has been loaded into memory but has not yet started execution.
Memory Management Unit (MMU)	Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.
MMU	*See* Memory Management Unit.
Multi-ICE	A multi-processor JTAG-based debug tool for embedded systems. Multi-ICE is an ARM registered trademark.
Output section	Is a contiguous sequence of input sections that have the same RO, RW, or ZI attributes. The sections are grouped together in larger fragments called regions. The regions are grouped together into the final executable image. *See also* Region.
PCS	Procedure Call Standard. *See also* ATPCS.
PIC	Position Independent Code. *See also* ROPI.
PID	Position Independent Data *or* the ARM Platform-Independent Development (PID) board. *See also* RWPI
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.
Program image	See Image.
Read Only Position Independent (ROPI)	Code and read-only data addresses can be changed at run-time.

Read Write Position Independent (RWPI)	Read/write data addresses can be changed at run-time.
RealView Compilation Tools (RVCT)	RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enable you to write and build applications for the ARM family of RISC processors.
Reentrancy	The ability of a subroutine to have more that one instance of the code active. Each instance of the subroutine call has its own copy of any required static data.
Remapping	Changing the address of physical memory or devices after the application has started executing. This is typically done to enable RAM to replace ROM when the initialization has been done.
Regions	In an Image, a region is a contiguous sequence of one to three output sections (RO, RW, and ZI).
Retargeting	The process of moving code designed for one execution environment to a new execution environment.
ROPI	*See* Read Only Position Independent.
RTOS	Real Time Operating System.
RVCT	*See* RealView Compilation Tools.
RWPI	*See* Read Write Position Independent.
Scatter-loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.
Section	A block of software code or data for an Image. *See also* Input sections.
Semihosting	A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.
Software Interrupt (SWI)	An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.
SWI	*See* Software Interrupt.
Target	The actual target processor, (real or simulated), on which the application is running.
TCM	*See* Tightly Coupled Memory.

Thread	A context of execution on a processor. A thread is always related to a processor and might or might not be associated with an image.
Tightly Coupled Memory (TCM)	Tightly Coupled Memory replaces an area of off-chip memory when enabled.
Veneer	A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.
Word	A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.