ARM Developer Suite

Version 1.0.1

Developer Guide

ARM DUI 0056B

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
October 1999	A	Release 1.0
March 2000	B	Release 1.0.1

Proprietary Notice

Contents
Developer Guide

Preface

This preface introduces the *ARM Developer Suite* (ADS) Developer Guide. It contains the following sections:

- *About this book* on page Preface-viii
- *Feedback* on page Preface-xii.

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 as described in ADS *Getting Started*.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the *ARM Developer Suite* (ADS) and the differences between ADS and the ARM Software Development Toolkit version 2.50.

Chapter 2 *Assembly Language Programming*

Read this chapter for an introduction to the general principles of writing ARM and Thumb assembly language.

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 *Mixed Language Programming*

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 assemblers 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 *Writing Code for ROM*

Read this chapter for details on building ROM images. These can be used in, for example, embedded applications. There are also hints on how to avoid the most common errors in writing code for ROM.

Typographical conventions

The following typographical conventions are used in this book:

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

`typewriter` (with `type` underlined) Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.

`typewriter italic`

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

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

bold Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

`typewriter bold`

Denotes language keywords when used outside example code and 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.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

This book contains information that is specific to the version of the CodeWarrior IDE supplied with the *ARM Developer Suite* (ADS). Refer to the following books in the ADS document suite for information on other components:

- *Getting Started* (ARM DUI 0064A)
- *ADS Tools Guide* (ARM DUI 0067A)
- *ADS Debuggers Guide* (ARM DUI 0066A)
- *ADS Debug Target Guide* (ARM DUI 0058A)
- *CodeWarrior IDE Guide* (ARM DUI 0065A). The CodeWarrior IDE and guide is available only on Windows.

The following additional documentation is provided with the ARM Developer Suite:

- *ARM Architecture Reference Manual* (ARM DUI 0100). This is supplied in Dynatext format, and in PDF format in `install_directory\PDF\ARM-DDI0100B_armarm.pdf`.
- *ARM Applications Library Programmer's Guide* (ARM DUI 0081). This is supplied in Dynatext format, and in PDF format on the CD.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory\PDF\specs\ARM ELFA08.pdf`.
- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory\PDF\specs\TIS-DWARF2.pdf`.
- *Angel Debug Protocol*. This is supplied in PDF format in `install_directory\PDF\specs\ADP ARM-DUI0052C.pdf`
- *Angel Debug Protocol Messages*. This is supplied in PDF format in `install_directory\PDF\specs\ADP ARM-DUI0053D.pdf`

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

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

This book is not intended to be an introduction to the ARM assembly language, C, or C++ programming languages. It does not try to teach programming in C or C++, and it is not a reference manual for the C or C++ standards.

The following book gives general information about the ARM architecture:

- *ARM System Architecture*, Furber, S., (1996). Addison Wesley Longman, Harlow, England. ISBN 0-201-40352-8.

Feedback

ARM Limited welcomes feedback on both the ARM Developer Suite, and its documentation.

Feedback on the ARM Developer Suite

If you have any problems with this book, please contact your supplier. To help us provide a rapid and useful response, please give:

- 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 number of the tool, including the version number and build number.

Feedback on this book

If you have any problems with this book, please 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 *ARM Developer Suite* (ADS) and describes the differences between ADS and the *ARM Software Development Toolkit*. It contains the following sections:

- *About the ARM Developer Suite* on page 1-2
- *Supported platforms* on page 1-5
- *What is different?* on page 1-6.

1.1 About the ARM Developer Suite

The ARM Developer Suite (ADS) consists of a suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

You can use ADS to develop, build, and debug C, C++, or ARM assembly language programs.

1.1.1 Components of the ADS

ADS consists of the following major components:

- *Command-line development tools*
- *GUI development tools* on page 1-3
- *Utilities* on page 1-4
- *Supporting software* on page 1-4.

Command-line development tools

The following command-line development tools are provided:

armcc	The ARM C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 32-bit ARM code.
armcpp	This is the ARM C++ compiler. It compiles ISO C++ or EC++ source into 32-bit ARM code.
tcc	The Thumb C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 16-bit Thumb code.
tcpp	This is the Thumb C++ compiler. It compiles ISO C++ or EC++ source into 16-bit Thumb code.
armasm	The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source.
armlink	The ARM linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. The ARM linker creates ELF executable images.

armsd The ARM and Thumb symbolic debugger. This enables source level debugging of programs. You can single-step through C or assembly language source, set breakpoints and watchpoints, and examine program variables or memory.

Rogue Wave C++ library

The Rogue Wave library provides an implementation of the standard C++ library as defined in the *ISO/IEC 14822:1998 International Standard for C++*. For more information on Rogue Wave, see the online HTML documentation.

support library

The ARM C libraries provide additional components to enable support for C++ and to compile code for different architectures and processors.

GUI development tools

The following GUI development tools are provided:

AXD The new ARM Debugger for Windows and UNIX. This provides a full Windows environment for debugging your C, C++, and assembly language source.

ADW The old ARM Debugger for Windows. This provides a full Windows environment for debugging your C, C++, and assembly language source.

ADU The old ARM Debugger for UNIX. This provides a full GUI environment for debugging your C, C++, and assembly language source.

CodeWarrior IDE

The project manager for Windows. This is a graphical user interface tool that automates the routine operations of managing source files and building your software development projects. The CodeWarrior IDE helps you to construct the environment, and specify the procedures needed to build your software.

See the *ADS Debuggers Guide* and the *CodeWarrior IDE Guide* for more information on the development tools.

Utilities

The following utility tools are provided to support the main development tools:

fromELF	The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats, including AIF, plain binary, *Extended Intellec Hex* (IHF) format, Motorola 32-bit S record format, and Intel Hex 32 format.
armprof	The ARM profiler displays an execution profile of a program from a profile data file generated by an ARM debugger.
armar	The ARM librarian enables sets of ELF format object files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ELF files.

Supporting software

The following support software is provided to enable you to debug your programs, either under simulation, or on ARM-based hardware:

ARMulator	The ARM core simulator. This provides instruction-accurate simulation of ARM processors, and enables ARM and Thumb executable programs to be run on non-native hardware. The ARMulator is integrated with the ARM debuggers.
Angel	The ARM debug monitor. Angel runs on target development hardware and enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.

1.2 Supported platforms

This release of the ADS is supported on the following platforms:

- Sun workstations running Solaris 2.5.1 or 2.6
- Hewlett Packard workstations running HP-UX 10.20
- IBM-compatible PCs running Windows 95, Windows 98, or Windows NT 4.

The CodeWarrior IDE is supported on IBM-compatible PCs running Windows 95, Windows 98, and Windows NT 4.

1.3 What is different?

This section describes the major differences between ADS and ARM Software Development Toolkit version 2.50. The most important changes are:

- C and C++ libraries supplied as binaries with automatic selection of the appropriate library for the build option. See the *ADS Tools Guide*.
- The CodeWarrior IDE is used for project management instead of APM. See the *CodeWarrior IDE Guide*.
- AXD is a new debugger for Windows or UNIX (ADW and ADU are still supported). See the *ADS Debuggers Guide*.
- armar replaces armlib as library manager. See the *ADS Tools Guide*.
- The preferred and default executable image format is now ELF. Refer to the ELF description in `\PDF\specs` for details of the ARM implementation of standard ELF format.
- The preferred and default debug table format is now DWARF2.
- There are additional command-line options for the compilers, assembler, and linker. See the *ADS Tools Guide*.
- The default *Procedure Call Standard* (PCS) for both the ARM and Thumb compilers, and the assembler in ADS has changed. See Chapter 3 *Using the Procedure Call Standard* and the *ADS Tools Guide* for more details.
- The default options are different for ADS and ARM Software Development Toolkit version 2.50. See the *ADS Tools Guide*.

For a complete list of differences, see the differences chapter in *Getting Started* and the referenced manuals.

Chapter 2
Assembly Language Programming

This chapter provides an introduction to the general principles of writing ARM and Thumb assembly language. It contains the following sections:

2.1 Introduction

This chapter gives a basic, practical understanding of how to write ARM and Thumb assembly language modules. It also gives information on the facilities provided by the *ARM assembler* (armasm). Refer to the assembler chapter in *ADS Tools Guide* for further information.

This chapter does not give specific information about the inline assemblers in the ARM C and C++ compilers (see Chapter 5 *Mixed Language Programming*).

This chapter does not provide a detailed description of either the ARM instruction set or the Thumb instruction set. This information can be found in the *ARM Architecture Reference Manual*.

2.1.1 Code examples

There are a number of code examples in this chapter. Many of them are supplied in the `examples\asm` directory of the ADS.

Follow these steps to build, link, and execute an assembly language file:

1. Type `armasm -g` *`filename`*`.s` at the command prompt to assemble the file and generate debug tables.
2. Type `armlink` *`filename`*`.o -o` *`filename`* to link the object file and generate and ELF executable image.
3. Type `armsd` *`filename`* to load the image file into the debugger.
4. Type `go` at the `armsd:` prompt to execute it.
5. Type `quit` at the `armsd:` prompt to return to the command line.

To see how the assembler converts the source code, enter:

```
fromelf filename.o -text/c
```

or run the module in AXD, ADW, or ADU with interleaving on.

See:

- *ADS Debuggers Guide* for details on armsd, AXD, ADW and ADU.
- *ADS Tools Guide* for details on `armlink` and `fromelf`.

2.2 Overview of the ARM architecture

This section gives a brief overview of the ARM architecture. Refer to *ARM Architecture Reference Manual* for a detailed description of the points described here.

The ARM is typical of RISC processors in that it implements a load/store architecture. Only load and store instructions can access memory. Data processing instructions operate on register contents only.

2.2.1 Architecture versions

The information and examples in this book assume that you are using a processor that implements ARM architecture v3 or above. Refer to *ARM Architecture Reference Manual* for a summary of the different architecture versions.

All these processors have a 32-bit addressing range.

2.2.2 ARM and Thumb state

Versions 4T, 4TxM, and 5T of the ARM architecture define a 16-bit instruction set called the Thumb instruction set. The functionality of the Thumb instruction set is a subset of the functionality of the 32-bit ARM instruction set. Refer to *Thumb instruction set overview* on page 2-9 for more information.

A processor that is executing Thumb instructions is operating in *Thumb state*. A processor that is executing ARM instructions is operating in *ARM state*.

A processor in ARM state cannot execute Thumb instructions, and a processor in Thumb state cannot execute ARM instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state.

Each instruction set includes instructions to change processor state.

You must also switch the assembler mode to produce the correct opcodes using `CODE16` and `CODE32` directives. Refer to the assembler chapter in *ADS Tools Guide* for details.

ARM processors always start in ARM state.

2.2.3 Processor mode

The ARM supports up to seven processor modes, depending on the architecture version. These are:

- User
- FIQ - Fast Interrupt Request
- IRQ - Interrupt Request
- Supervisor
- Abort
- Undefined
- System (ARM architecture v4 and above).

Applications that require task protection usually execute in User mode. Some embedded applications may run entirely in Supervisor or System modes.

The other modes are entered to service exceptions, or to access privileged resources. Refer to Chapter 6 *Handling Processor Exceptions*, and the *ARM Architecture Reference Manual* for more information.

2.2.4 Registers

The ARM processor has 37 registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations. Refer to the *ARM Architecture Reference Manual* for a detailed description of how registers are banked.

The following registers are available in ARM architecture v3 and above:

- *30 general-purpose, 32-bit registers*
- *The program counter (pc)* on page 2-5
- *The Current Program Status Register (CPSR)* on page 2-5
- *Five Saved Program Status Registers (SPSRs)* on page 2-5.

30 general-purpose, 32-bit registers

Fifteen general-purpose registers are visible at any one time, depending on the current processor mode, as r0, r1, ... ,r13, r14.

By convention in ARM assembly language r13 is used as a *stack pointer* (sp). The C and C++ compilers always use r13 as the stack pointer.

In User mode, r14 is used as a *link register* (lr) to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack.

In the exception handling modes, r14 holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. r14 can be used as a general-purpose register if the return address is stored on the stack.

The *program counter* (pc)

The program counter is accessed as r15 (or pc). It is incremented by one word (four bytes) for each instruction in ARM state, or by two bytes in Thumb state. Branch instructions load the destination address into the program counter. You can also load the program counter directly using data operation instructions. For example, to return from a subroutine, you can copy the link register into the program counter using:

```
    MOV  pc,lr
```

The *Current Program Status Register* (CPSR)

The CPSR holds:

- copies of the *Arithmetic Logic Unit* (ALU) status flags
- the current processor mode
- interrupt disable flags.

On Thumb-capable processors, the CPSR also holds the current processor state (ARM or Thumb).

The ALU status flags in the CPSR are used to determine whether or not conditional instructions are executed. Refer to *Conditional execution* on page 2-19 for more information.

Five *Saved Program Status Registers* (SPSRs)

The SPSRs are used to store the CPSR when an exception is taken. One SPSR is accessible in each of the exception-handling modes. User mode and System mode do not have an SPSR because they are not exception handling modes. Refer to Chapter 6 *Handling Processor Exceptions*, and the *ARM Architecture Reference Manual* for more information.

2.2.5 ARM instruction set overview

All ARM instructions are 32 bits long. Instructions are stored word-aligned, so the least significant two bits of instruction addresses are always zero in ARM state. Some instructions use the least significant bit to determine whether the code being branched to is Thumb code or ARM code.

See the *ARM Architecture Reference Manual* for detailed information on the syntax of the ARM instruction set.

ARM instructions can be classified into a number of functional groups:

- *Branch instructions*
- *Data processing instructions*
- *Single register load and store instructions*
- *Multiple register load and store instructions* on page 2-7
- *Status register access instructions* on page 2-7
- *Semaphore instructions* on page 2-7
- *Coprocessor instructions* on page 2-7.

Branch instructions

These instructions are used to:

- branch backwards to form loops
- branch forward in conditional structures
- branch to subroutines
- change the processor from ARM state to Thumb state.

Data processing instructions

These instructions operate on the general-purpose registers. They perform operations such as addition, subtraction, or bitwise logic on the contents of two registers and place the result in a third register. Long multiply instructions (unavailable in some architectures) give a 64-bit result in two registers.

Single register load and store instructions

These instructions load or store the value of a single register from or to memory. They can load or store a 32-bit word or an 8-bit unsigned byte. In ARM architecture v4 and above they can also load or store a 16-bit unsigned halfword, or load and sign extend a 16-bit halfword or an 8-bit byte.

Multiple register load and store instructions

These instructions load or store any subset of the general-purpose registers from or to memory. Refer to *Load and store multiple register instructions* on page 2-39 for a detailed description of these instructions.

Status register access instructions

These instructions move the contents of the CPSR or an SPSR to or from a general-purpose register.

Semaphore instructions

These instructions load and alter a memory semaphore.

Coprocessor instructions

These instructions support a general way to extend the ARM architecture.

2.2.6 ARM instruction capabilities

The following general points apply to ARM instructions:

- *Conditional execution*
- *Register access*
- *Access to the inline barrel shifter*.

Conditional execution

All ARM instructions can be executed conditionally on the value of the ALU status flags in the CPSR. You do not need to use branches to skip conditional instructions, although it may be better to do so when a series of instructions depend on the same condition.

You can specify whether a data processing instruction sets the state of these flags or not. You can use the flags set by one instruction to control execution of other instructions even if there are many instructions in between.

Refer to *Conditional execution* on page 2-19 for a detailed description.

Register access

In ARM state, all instructions can access r0-r14, and most also allow access to r15 (pc). The `MRS` and `MSR` instructions can move the contents of the CPSR and SPSRs to a general-purpose register, where they can be manipulated by normal data processing operations. Refer to the *ARM Architecture Reference Manual* for more information.

Access to the inline barrel shifter

The ARM arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations. The second operand to all ARM data-processing and single register data-transfer instructions can be shifted, before the data-processing or data-transfer is executed, as part of the instruction. This supports, but is not limited to:

- scaled addressing
- multiplication by a constant
- constructing constants.

Refer to *Loading constants into registers* on page 2-24 for more information on using the barrel-shifter to generate constants.

2.2.7 Thumb instruction set overview

The functionality of the Thumb instruction set, with one exception, is a subset of the functionality of the ARM instruction set. The instruction set is optimized for production by a C or C++ compiler.

All Thumb instructions are 16 bits long and are stored halfword-aligned in memory. Because of this, the least significant bit of the address of an instruction is always zero in Thumb state. Some instructions use the least significant bit to determine whether the code being branched to is Thumb code or ARM code.

All Thumb data processing instructions:

- operate on full 32-bit values in registers
- use full 32-bit addresses for data access and for instruction fetches.

Refer to the *ARM Architecture Reference Manual* for detailed information on the syntax of the Thumb instruction set, and how Thumb instructions differ from their ARM counterparts.

In general, the Thumb instruction set differs from the ARM instruction set in the following ways:

- *Branch instructions*
- *Data processing instructions* on page 2-10
- *Single register load and store instructions* on page 2-10
- *Multiple register load and store instructions* on page 2-10.

There are no Thumb coprocessor instructions, no Thumb semaphore instructions, and no Thumb instructions to access the CPSR or SPSR.

Branch instructions

These instructions are used to:

- branch backwards to form loops
- branch forward in conditional structures
- branch to subroutines
- change the processor from Thumb state to ARM state.

Program-relative branches, particularly conditional branches, are more limited in range than in ARM code, and branches to subroutines can only be unconditional.

Data processing instructions

These operate on the general-purpose registers. The result of the operation is put in one of the operand registers, not in a third register. There are fewer data processing operations available than in ARM state. They have limited access to registers r8 to r15.

The ALU status flags in the CPSR are always updated by these instructions except when `MOV` or `ADD` instructions access registers r8 to r15. Thumb data processing instructions that access registers r8 to r15 cannot update the flags.

Single register load and store instructions

These instructions load or store the value of a single low register from or to memory. In Thumb state they cannot access registers r8 to r15.

Multiple register load and store instructions

These instructions load from memory or store to memory any subset of the registers in the range r0 to r7.

In addition, the `PUSH` and `POP` instructions implement a full descending stack using the stack pointer (r13) as the base. `PUSH` can stack the link register and `POP` can load the program counter.

2.2.8 Thumb instruction capabilities

The following general points apply to Thumb instructions:

- *Conditional execution*
- *Register access*
- *Access to the barrel shifter*.

Conditional execution

The conditional branch instruction is the only Thumb instruction that can be executed conditionally on the value of the ALU status flags in the CPSR. All data processing instructions update these flags, except when one or more high registers are specified as operands to the `MOV` or `ADD` instructions. In these cases the flags *cannot* be updated.

You cannot have any data processing instructions between an instruction that sets a condition and a conditional branch that depends on it. Use a conditional branch over any instruction that you wish to be conditional.

Register access

In Thumb state, most instructions can access only r0-r7. These are referred to as the low registers.

Registers r8 to r15 are limited access registers. In Thumb state these are referred to as high registers. They can be used, for example, as fast temporary storage.

Refer to the *ARM Architecture Reference Manual* for a complete list of the Thumb data processing instructions that can access the high registers.

Access to the barrel shifter

In Thumb state you can use the barrel shifter only in a separate operation, using an `LSL`, `LSR`, `ASR,` or `ROR` instruction.

2.3 Structure of assembly language modules

Assembly language is the language that the ARM assembler (`armasm`) parses and assembles to produce object code. This can be:

- ARM assembly language
- Thumb assembly language
- a mixture of both.

2.3.1 Layout of assembly language source files

The general form of source lines in assembly language is:

```
{label} {instruction|directive|pseudo-instruction} {;comment}
```

Note

Instructions, pseudo-instructions, and directives must be preceded by white space, such as a space or a tab, even if there is no label.

All three sections of the source line are optional. You can use blank lines to make your code more readable.

Case rules

Instruction mnemonics, directives, and symbolic register names can be written in uppercase or lowercase, but not mixed.

Line length

To make source files easier to read, a long line of source can be split onto several lines by placing a backslash character (\ ) at the end of the line. The backslash must not be followed by any other characters (including spaces and tabs). The backslash/end-of-line sequence is treated by the assembler as white space.

Note

Do not use the backslash/end-of-line sequence within quoted strings.

The exact limit on the length of lines, including any extensions using backslashes, depends on the contents of the line, but is generally between 128 and 255 characters.

Labels

Labels are symbols that represent addresses. The address given by a label is calculated during assembly.

The assembler calculates the address of a label relative to the origin of the section where the label is defined. A reference to a label within the same section can use the program counter plus or minus an offset. This is called *program-relative addressing*.

Labels can be defined in a map. See *Describing data structures with MAP and FIELD directives* on page 2-51. You can place the origin of the map in a specified register at runtime, and references to the label use the specified register plus an offset. This is called *register-relative addressing*.

Addresses of labels in other sections are calculated at link time, when the linker has allocated specific locations in memory for each section.

Local labels

Local labels are a subclass of label. A local label begins with a number in the range 0-99. Unlike other labels, a local label can be defined many times. Local labels are useful when you are generating labels with a macro. When the assembler finds a reference to a local label, it links it to a nearby instance of the local label.

The scope of local labels is limited by the `AREA` directive. You can use the `ROUT` directive to limit the scope more tightly.

Refer to the assembler chapter in *ADS Tools Guide* for details of:

- the syntax of local label declarations
- how the assembler associates references to local labels with their labels.

Comments

The first semicolon on a line marks the beginning of a comment, except where the semicolon appears inside a string constant. The end of the line is the end of the comment. A comment alone is a valid line. All comments are ignored by the assembler.

Constants

Numbers	Numeric constants are accepted in three forms: • Decimal, for example, 123 • Hexadecimal, for example, 0x7b • `n_xxx` where: *n* is a base between 2 and 9; *xxx* is a number in that base.
Boolean	The Boolean constants `TRUE` and `FALSE` must be written as `{TRUE}` and `{FALSE}`.
Characters	Character constants consist of opening and closing single quotes, enclosing either a single character or an escaped character, using the standard C escape characters.
Strings	Strings consist of opening and closing double quotes, enclosing characters and spaces. If double quotes or dollar signs are used within a string as literal text characters, they must be represented by a pair of the appropriate character. For example, you must use `$$` if you require a single `$` in the string. The standard C escape sequences can be used within string constants.

2.3.2 An example ARM assembly language module

Example 2-1 illustrates some of the core constituents of an assembly language module. The example is written in ARM assembly language. It is supplied as `armex.s` in the `examples\asm` subdirectory of ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The constituent parts of this example are described in more detail in the following sections.

Example 2-1

```
        AREA     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SWI      0x123456       ; ARM semihosting SWI

        END                     ; Mark end of file
```

The AREA directive

ELF *sections* are independent, named, indivisible sequences of code or data. A single code section is the minimum required to produce an application.

The output of an assembly or compilation usually consists of two or more sections:

- a code section that is usually a read-only section
- a data section that is usually a read-write section.

The linker places each section in a program image according to section placement rules. Sections that are adjacent in source files are not necessarily adjacent in the application image. Refer to the linker chapter in *ADS Tools Guide* for more information on how the linker places sections.

In an ARM assembly language source file, the start of a section is marked by the `AREA` directive. This directive names the section and sets its attributes. The attributes are placed after the name, separated by commas. Refer to the assembler chapter in *ADS Tools Guide* for a detailed description of the syntax of the `AREA` directive.

You can choose any name for your sections. However, names starting with any nonalphabetic character must be enclosed in bars, or an `AREA name missing` error is generated. For example: `|1_DataArea|`.

Example 2-1 defines a single section called `ARMex` that contains code and is marked as being `READONLY`.

The ENTRY directive

The `ENTRY` directive marks the first instruction to be executed. In applications containing C code, an entry point is also contained within the C library initialization code.

Application execution

The application code in Example 2-1 begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers r0 and r1. These registers are added together and the result placed in r0.

Application termination

After executing the main code, the application terminates by returning control to the debugger. This is done using the ARM semihosting SWI (by default this is `0x123456`), with the following parameters:

- r0 equal to `angel_SWIreason_ReportException` (by default `0x18`)
- r1 equal to `ADP_Stopped_ApplicationExit` (by default `0x20026`)

Refer to the Angel chapter in *ADS Debug Target Guide* for additional information.

The END directive

This directive instructs the assembler to stop processing this source file. Every assembly language source module must finish with an `END` directive on a line by itself.

2.3.3 Calling subroutines

To call subroutines in assembly language, use a branch and link instruction. The syntax is:

```
BL  destination
```

where `destination` is usually the label on the first instruction of the subroutine.

`destination` could alternatively be a program-relative or register-relative expression. Refer to the assembler chapter in *ADS Tools Guide* for further information.

The `BL` instruction:

- places the return address in the *link register* (lr)
- sets pc to the address of the subroutine.

After the subroutine code is executed you can use a `MOV pc,lr` instruction to return. By convention, registers r0-r3 are used to pass parameters to subroutines, and to pass results back to the callers.

Note

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the procedure call standard. Refer to Chapter 3 *Using the Procedure Call Standard* for more information.

Example 2-2 shows a subroutine that adds the values of its two parameters and returns a result in r0. It is supplied as `subrout.s` in the `examples\asm` subdirectory of the ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-2

```
        AREA    subrout, CODE, READONLY
                                ; Name this block of code
        ENTRY                   ; Mark first instruction to execute
start   MOV     r0, #10         ; Set up parameters
        MOV     r1, #3
        BL      doadd           ; Call subroutine
stop    MOV     r0, #0x18       ; angel_SWIreason_ReportException
        LDR     r1, =0x20026    ; ADP_Stopped_ApplicationExit
        SWI     0x123456        ; ARM semihosting SWI

doadd   ADD     r0, r0, r1      ; Subroutine code
        MOV     pc, lr          ; Return from subroutine
        END                     ; Mark end of file
```

2.3.4 An example Thumb assembly language module

Example 2-3 illustrates some of the core constituents of a Thumb assembly language module. It is based on `subrout.s`. It is supplied as `thumbsub.s` in the `examples\asm` subdirectory of the ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-3

```
        AREA ThumbSub, CODE, READONLY   ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        CODE32                          ; Subsequent instructions are ARM
header  ADR     r0, start + 1           ; Processor starts in ARM state,
        BX      r0                      ; so small ARM code header used
                                        ; to call Thumb main program
        CODE16                          ; Subsequent instructions are Thumb
start
        MOV     r0, #10                 ; Set up parameters
        MOV     r1, #3
        BL      doadd                   ; Call subroutine
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SWI     0xAB                    ; Thumb semihosting SWI
doadd
        ADD     r0, r0, r1              ; Subroutine code
        MOV     pc, lr                  ; Return from subroutine
        END                             ; Mark end of file
```

CODE32 and CODE16 directives

These directives instruct the assembler to assemble subsequent instructions as ARM (`CODE32`) or Thumb (`CODE16`) instructions. They do not assemble to an instruction to change the processor state at runtime. They only change the assembler state.

The ARM assembler, `armasm`, starts in ARM mode by default. You can use the `-16` option in the command line if you want it to start in Thumb mode.

BX instruction

This instruction is a branch that can change processor state at runtime. The least significant bit of the target address specifies whether it is an ARM instruction (clear) or a Thumb instruction (set). In this example, this bit is set in the `ADR` pseudo-instruction.

2.4 Conditional execution

In ARM state, each data processing instruction has an option to update ALU status flags in the *Current Program Status Register* (CPSR) according to the result of the operation.

Add an S suffix to an ARM instruction to make it update the ALU status flags in the CPSR.

Do not use the S suffix with `CMP`, `CMN`, `TST`, or `TEQ`. These comparison instructions always update the flags. This is their only effect.

In Thumb state, there is no option. All data processing instructions update the ALU status flags in the CPSR, except when one or more high registers are used in `MOV` and `ADD` instructions. `MOV` and `ADD` cannot update the status flags in these cases.

Every ARM instruction can be executed conditionally on the state of the ALU status flags in the CPSR. Refer to Table 2-1 on page 2-20 for a list of the suffixes to add to instructions to make them conditional.

In ARM state, you can:

- update the ALU status flags in the CPSR on the result of a data operation
- execute several other data operations without updating the flags
- execute following instructions or not, according to the state of the flags updated in the first operation.

In Thumb state you cannot execute data operations without updating the flags, and conditional execution can only be achieved using conditional branches. The only Thumb instruction that can be conditional is the conditional branch instruction (`B`). The suffixes for this instruction are the same as in ARM state. The branch with link (`BL`) or branch and exchange instruction set (`BX`) instructions cannot be conditional.

2.4.1 The ALU status flags

The CPSR contains the following ALU status flags:

`N`	Set when the result of the operation was Negative.
`Z`	Set when the result of the operation was Zero.
`C`	Set when the operation resulted in a Carry.
`V`	Set when the operation caused oVerflow.
`Q`	Sticky flag. (ARM architecture v5E only.)

A carry occurs if the result of an add, subtract, or compare is greater than or equal to 2^{32}, or as the result of an inline barrel shifter operation in a move or logical instruction.

Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}.

2.4.2 Execution conditions

The relation of condition code suffixes to the `N`, `Z`, `C` and `V` flags is shown in Table 2-1.

Table 2-1 Condition code suffixes

Suffix	Flags	Meaning
`EQ`	`Z` set	Equal
`NE`	`Z` clear	Not equal
`CS/HS`	`C` set	Higher or same (unsigned >=)
`CC/LO`	`C` clear	Lower (unsigned <)
`MI`	`N` set	Negative
`PL`	`N` clear	Positive or zero
`VS`	`V` set	Overflow
`VC`	`V` clear	No overflow
`HI`	`C` set and `Z` clear	Higher (unsigned >)
`LS`	`C` clear and `Z` set	Lower or same (unsigned <=)
`GE`	`N` and `V` the same	Signed >=
`LT`	`N` and `V` differ	Signed <
`GT`	`Z` clear, `N` and `V` the same	Signed >
`LE`	`Z` set, `N` and `V` differ	Signed <=

Examples

```
    ADD     r0, r1, r2     ; r0 = r1 + r2, don't update flags

    ADDS    r0, r1, r2     ; r0 = r1 + r2 and update flags

    ADDEQS  r0, r1, r2     ; If Z flag set then r0 = r1 + r2,
                           ; and update flags

    CMP     r0, r1         ; update flags based on r0-r1.
```

2.4.3 Using conditional execution in ARM state

You can use conditional execution of ARM instructions to reduce the number of branch instructions in your code. This improves code density.

Branch instructions are also expensive in processor cycles. On ARM processors without branch prediction hardware, it typically takes three processor cycles to refill the processor pipeline each time a branch is taken.

Some ARM processors, for example ARM10 and StrongARM, have branch prediction hardware. In systems using these processors, the pipeline only needs to be flushed and refilled when there is a misprediction.

Example 2-4: Euclid's Greatest Common Divisor

This example uses two implementations of Euclid's *Greatest Common Divisor* (gcd) algorithm. It demonstrates how you can use conditional execution to improve code density and execution speed. The detailed analysis of execution speed only applies to an ARM7 processor. The code density calculations apply to all ARM processors.

In C the algorithm can be expressed as:

```
int gcd(int a, int b)
{
    while (a != b) do
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

You can implement the gcd function with conditional execution of branches only, in the following way:

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUB     r0, r0, r1
        B       gcd
less
        SUB     r1, r1, r0
        B       gcd
end
```

Because of the number of branches, the code is seven instructions long. Every time a branch is taken, the processor must refill the pipeline and continue from the new location. The other instructions and non-executed branches use a single cycle each.

By using the conditional execution feature of the ARM instruction set, you can implement the gcd function in only four instructions:

```
gcd
        CMP      r0, r1
        SUBGT    r0, r0, r1
        SUBLT    r1, r1, r0
        BNE      gcd
```

In addition to improving code size, this code executes faster in most cases. Table 2-2 and Table 2-3 show the number of cycles used by each implementation for the case where r0 equals 1 and r1 equals 2. In this case, replacing branches with conditional execution of all instructions saves three cycles.

The conditional version of the code executes in the same number of cycles for any case where r0 equals r1. In all other cases, the conditional version of the code executes in fewer cycles.

Table 2-2 Conditional branches only

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1 (not executed)
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			Total = 13

Table 2-3 All instructions conditional

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1 (not executed)
1	1	`BNE gcd`	1 (not executed)
			Total = 10

Converting to Thumb

Because `B` is the only Thumb instruction that can be executed conditionally, the gcd algorithm in Example 2-4 must be written with conditional branches in Thumb code.

Like the ARM conditional branch implementation, the Thumb code requires seven instructions. However, because Thumb instructions are only 16 bits long, the overall code size is 14 bytes, compared to 16 bytes for the smaller ARM implementation.

In addition, on a system using 16-bit memory the Thumb version runs *faster* than the second ARM implementation because only one memory access is required for each Thumb instruction, whereas each ARM instruction requires two fetches.

Branch prediction and caches

To optimize code for execution speed you need detailed knowledge of the instruction timings, branch prediction logic, and cache behavior of your target system. Refer to the *ARM Architecture Reference Manual* and data sheets for individual processors for full information.

2.5 Loading constants into registers

You cannot load an arbitrary 32-bit immediate constant into a register in a single instruction without performing a data load from memory. This is because ARM instructions are only 32 bits long.

Thumb instructions have the same limitation.

You can load any 32-bit value into a register with a data load, but there are more direct and efficient ways to load many commonly-used constants.

The following sections describe:

- how to use the `MOV` and `MVN` instructions to load a range of immediate values, see *Direct loading with MOV and MVN* on page 2-25
- how to use the `LDR` pseudo-instruction to load any 32-bit constant, see *Loading with LDR Rd, =const* on page 2-27
- how to load floating-point constants, see *Loading floating-point constants* on page 2-29.

2.5.1 Direct loading with MOV and MVN

In ARM state, you can use the `MOV` and `MVN` instructions to load a range of 8-bit constant values directly into a register:

- `MOV` loads any 8-bit constant value, giving a range of `0x0` to `0xff` (0-255)
- `MVN` loads the bitwise complement of these values, giving a range of `0xffffff00` to `0xffffffff`.

In addition, you can use either `MOV` or `MVN` in conjunction with the barrel shifter to generate a wider range of constants. The barrel shifter can right-rotate 8-bit values through any even number of positions from 2 to 30.

You can use `MOV` to load values that follow the pattern shown in Table 2-4, in a single instruction. Use `MVN` to load the bitwise complement of these values. Right-rotates by 2, 4, or 6 bits produce bit patterns with a few bits at each end of a 32-bit word.

Table 2-4 ARM-state immediate constants

Decimal values	Equivalent hexadecimal	Step between values	Rotate
0-255	`0-0xff`	1	No rotate
256, 260, 264, ... , 1020	`0x100-0x3fc`	4	Right by 30 bits
1024, 1040, 1056, ... , 4080	`0x400-0xff0`	16	Right by 28 bits
4096, 4160, 4224, ... , 16320	`0x1000-0x3fc0`	64	Right by 26 bits
...	...	...	...
64 x 2^{24}, 65 x 2^{24}, ... , 255 x 2^{24}	`0x40000000-0xff000000`	2^{24}	Right by 8 bits
4 x 2^{24}, ... , 252 x 2^{24} + 3	`0x04000000-0xfc000003`	2^{26}, 1	Right by 6 bits
16 x 2^{24}, ... , 240 x 2^{24} + 15	`0x10000000-0xf000000f`	2^{28}, 1	Right by 4 bits
64 x 2^{24}, ... , 192 x 2^{24} + 63	`0x40000000-0xc000003f`	2^{30}, 1	Right by 2 bits

Using MOV and MVN

You do not need to work out how to load a constant using `MOV` or `MVN`. The assembler attempts to convert any constant value to an acceptable form. This means that you can use `MOV` and `MVN` in two ways:

- Convert the value to an 8-bit constant, followed by the right-rotate value. For example:

  ```
  MOV     r0, #0xFF,ROR 30     ; r0 = 1020
  ```

- Allow the assembler to do the work of converting the value. If you specify the constant to be loaded, the assembler converts it to an acceptable form if possible. For example:

  ```
  MOV     r0, #0x3FC           ; r0 = 1020
  ```

  If the constant cannot be expressed as a right-rotated 8-bit value or its bitwise complement, the assembler reports the error, `Immediate` *n* `out of range for this operation`.

Table 2-5 gives an example of how the assembler converts constants. The left-hand column lists the ARM instructions input to the assembler. The right-hand column shows the instruction generated by the assembler.

Table 2-5 Assembler-generated constants

Input instruction	Assembled equivalent
`MOV r0, #0`	`MOV r0, #0`
`MOV r1, #0xFF000000`	`MOV r1, #0xFF, 8`
`MOV r2, #0xFFFFFFFF`	`MVN r2, #0`
`MVN r3, #1`	`MVN r3, #1`
`MOV r4, #0xFC000003`	`MOV r4, #0xFF, 6`
`MOV r5, #0x03FFFFFC`	`MVN r5, #0xFF, 6`
`MOV r6, #0x55555555`	`Error (cannot be constructed)`

Direct loading with MOV in Thumb state

In Thumb state you can use the `MOV` instruction to load constants in the range 0-255. You cannot generate constants outside this range because:

- The Thumb `MOV` instruction does not provide inline access to the barrel shifter. Constants cannot be right-rotated as they can in ARM state.
- The Thumb `MVN` instruction can act only on registers and not on constant values. Bitwise complements cannot be directly loaded as they can in ARM state.

If you attempt to use a `MOV` instruction with a value outside the range 0-255, the assembler reports the error, `Immediate` *`n`* `out of range for this operation`.

2.5.2 Loading with LDR Rd, =const

The `LDR Rd,=`*`const`* pseudo-instruction can construct any 32-bit numeric constant in a single instruction. Use this pseudo-instruction to generate constants that are out of range of the `MOV` and `MVN` instructions.

The `LDR` pseudo-instruction generates the most efficient code for a specific constant:

- If the constant can be constructed with a `MOV` or `MVN` instruction, the assembler generates the appropriate instruction.
- If the constant cannot be constructed with a `MOV` or `MVN` instruction, the assembler:
  - — places the value in a *literal pool* (a portion of memory embedded in the code to hold constant values)
  - — generates an `LDR` instruction with a program-relative address that reads the constant from the literal pool.

  For example:

  ```
  LDR      rn, [pc, #offset to literal pool]
                                ; load register n with one word
                                ; from the address [pc + offset]
  ```

  You must ensure that there is a literal pool within range of the LDR instruction generated by the assembler. Refer to *Placing literal pools* for more information.

Refer to the assembler chapter in *ADS Tools Guide* for a description of the syntax of the `LDR` pseudo-instruction.

Placing literal pools

The assembler places a literal pool at the end of each section. These are defined by the `AREA` directive at the start of the following section, or by the `END` directive at the end of the assembly. The `END` directives at the ends of included files do not signal the end of sections.

In large sections the default literal pool may be out of range of one or more `LDR` instructions. The offset from the pc to the constant must be:

- less than 4KB in ARM state, but may be in either direction
- forward and less than 1KB in Thumb state.

When an `LDR Rd,=const` pseudo-instruction requires the constant to be placed in a literal pool, the assembler:

- Checks if the constant is available and addressable in any previous literal pools. If so, it addresses the existing constant.
- Attempts to place the constant in the next literal pool if it is not already available.

If the next literal pool is out of range, the assembler generates an error message. In this case you must use the `LTORG` directive to place an additional literal pool in the code. Place the `LTORG` directive after the failed `LDR` pseudo-instruction, and within 4KB (ARM) or 1KB (Thumb). Refer to the assembler chapter in *ADS Tools Guide* for a detailed description of the `LTORG` directive.

You must place literal pools where the processor does not attempt to execute them as instructions. Place them after unconditional branch instructions, or after the return instruction at the end of a subroutine.

Example 2-5 shows how this works in practice. It is supplied as `loadcon.s` in the `examples\asm` subdirectory of the ADS. The instructions listed as comments are the ARM instructions that are generated by the assembler. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-5

```
        AREA     Loadcon, CODE, READONLY
        ENTRY                              ; Mark first instruction to execute
start   BL       func1                     ; Branch to first subroutine
        BL       func2                     ; Branch to second subroutine
stop    MOV      r0, #0x18                 ; angel_SWIreason_ReportException
        LDR      r1, =0x20026              ; ADP_Stopped_ApplicationExit
        SWI      0x123456                  ; ARM semihosting SWI
func1
        LDR      r0, =42                   ; => MOV R0, #42
        LDR      r1, =0x55555555           ; => LDR R1, [PC, #offset to
                                           ; Literal Pool 1]
        LDR      r2, =0xFFFFFFFF           ; => MVN R2, #0
        MOV      pc, lr
        LTORG                              ; Literal Pool 1 contains
                                           ; literal Ox55555555
```

```
func2
        LDR      r3, =0x55555555            ; => LDR R3, [PC, #offset to
                                            ; Literal Pool 1]
        ; LDR r4, =0x66666666               ; If this is uncommented it
                                            ; fails, because Literal Pool 2
                                            ; is out of reach
        MOV      pc, lr
LargeTable
        SPACE    4200                       ; Starting at the current location,
                                            ; clears a 4200 byte area of memory
                                            ; to zero
        END                                 ; Literal Pool 2 is empty
```

2.5.3 Loading floating-point constants

You can load any single-precision or double-precision floating-point constant in a single instruction, using the following pseudo-instructions:

- `LDFS` *`fp-register,=fp-literal`*
- `LDFD` *`fp-register,=fp-literal`*
- `FLDS` *`fp-register,=fp-literal`*
- `FLDD` *`fp-register,=fp-literal`*

Refer to the assembler chapter in *ADS Tools Guide* for details.

2.6 Loading addresses into registers

It is often necessary to load an address into a register. You may need to load the address of a variable, a string constant, or the start location of a jump table.

Addresses are normally expressed as offsets from the current pc or other register.

This section describes two methods for loading an address into a register:

- load the register directly, see *Direct loading with ADR and ADRL* below.
- load the address from a literal pool, see *Loading addresses with LDR Rd, = label* on page 2-35.

2.6.1 Direct loading with ADR and ADRL

The `ADR` and `ADRL` pseudo-instructions enable you to load a range of addresses without performing a data load. `ADR` and `ADRL` accept either of the following:

- A program-relative expression, which is a label with an optional offset, where the address of the label is relative to the current pc.
- A register-relative expression, which is a label with an optional offset, where the address of the label is relative to an address held in a specified general-purpose register. Refer to *Describing data structures with MAP and FIELD directives* on page 2-51 for information on specifying register-relative expressions.

The assembler converts an `ADR` *`rn,label`* pseudo-instruction by generating:

- a single `ADD` or `SUB` instruction that loads the address, if it is in range
- an error message if the address cannot be reached in a single instruction.

The offset range is 255 bytes for an offset to a non word-aligned address, and 1020 bytes (255 words) for an offset to a word-aligned address.

The assembler converts an `ADRL` *`rn,label`* pseudo-instruction by generating:

- two data-processing instructions that load the address, if it is in range
- an error message if the address cannot be constructed in two instructions.

The range of an `ADRL` pseudo-instruction is 64KB for a non word-aligned address and 256KB for a word-aligned address.

`ADRL` assembles to two instructions, if successful. The assembler generates two instructions even if the address could be loaded in a single instruction.

Refer to *Loading addresses with LDR Rd, = label* on page 2-35 for information on loading addresses that are outside the range of the `ADRL` pseudo-instruction.

Note

The label used with ADR or ADRL must be within the same code section. The assembler faults references to labels that are out of range in the same section. The linker faults references to labels that are out of range in other code sections.

In Thumb state, ADR can generate word-aligned addresses only.

ADRL is not available in Thumb code. Use it only in ARM code.

Example 2-6 shows the type of code generated by the assembler when assembling ADR and ADRL pseudo-instructions. It is supplied as adrlabel.s in the examples\asm subdirectory of the ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The instructions listed in the comments are the ARM instructions generated by the assembler.

Example 2-6

```
            AREA    adrlabel, CODE,READONLY
            ENTRY                           ; Mark first instruction to execute
Start
            BL      func                    ; Branch to subroutine
stop        MOV     r0, #0x18               ; angel_SWIreason_ReportException
            LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
            SWI     0x123456                ; ARM semihosting SWI
            LTORG                           ; Create a literal pool
func        ADR     r0, Start               ; => SUB r0, PC, #offset to Start
            ADR     r1, DataArea            ; => ADD r1, PC, #offset to DataArea
            ; ADR   r2, DataArea+4300       ; This would fail because the offset
                                            ; cannot be expressed by operand2
                                            ; of an ADD
            ADRL    r3, DataArea+4300       ; => ADD r2, PC, #offset1
                                            ;  ADD r2, r2, #offset2
            MOV     pc, lr                  ; Return
DataArea    SPACE   8000                    ; Starting at the current location,
                                            ; clears a 8000 byte area of memory
                                            ; to zero
            END
```

Implementing a jump table with ADR

Example 2-7 on page 2-32 shows ARM code that implements a jump table. It is supplied as `jump.s` in the `examples\asm` subdirectory of the ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The `ADR` pseudo-instruction loads the address of the jump table.

In the example, the function `arithfunc` takes three arguments and returns a result in r0. The first argument determines which operation is carried out on the second and third arguments:

argument1=0	Result = argument2 + argument3
argument1=1	Result = argument2 – argument3

The jump table is implemented with the following instructions and assembler directives:

`EQU` Is an assembler directive. It is used to give a value to a symbol. In this example it assigns the value 2 to `num`. When `num` is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

`DCD` Declares one or more words of store. In this example each `DCD` stores the address of a routine that handles a particular clause of the jump table.

`LDR` The `LDR  pc,[r3,r0,LSL#2]` instruction loads the address of the required clause of the jump table into the pc. It:

- multiplies the clause number in r0 by 4 to give a word offset
- adds the result to the address of the jump table
- loads the contents of the combined address into the program counter.

Example 2-7 ARM code jump table

```
        AREA    Jump, CODE, READONLY      ; Name this block of code
        CODE32                            ; Following code is ARM code
num     EQU     2                         ; Number of entries in jump table
        ENTRY                             ; Mark first instruction to execute
start                                     ; First instruction to call
        MOV     r0, #0                    ; Set up the three parameters
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc                   ; Call the function
stop    MOV     r0, #0x18                   ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
        SWI     0x123456                    ; ARM semihosting SWI
arithfunc                                   ; Label the function
        CMP     r0, #num                    ; Treat function code as unsigned integer
        MOVHS   pc, lr                      ; If code is >= num then simply return
        ADR     r3, JumpTable               ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]           ; Jump to the appropriate routine
JumpTable
        DCD     DoAdd
        DCD     DoSub

DoAdd   ADD     r0, r1, r2                  ; Operation 0
        MOV     pc, lr                      ; Return
DoSub   SUB     r0, r1, r2                  ; Operation 1
        MOV     pc,lr                       ; Return
        END                                 ; Mark the end of this file
```

Converting to Thumb

Example 2-8 on page 2-33 shows the implementation of the jump table converted to Thumb code.

Most of the Thumb version is the same as the ARM code. The differences are commented in the Thumb version.

In Thumb state, you cannot:

- increment the base register of `LDR` and `STR` instructions
- load a value into the pc using an `LDR` instruction
- do an inline shift of a value held in a register.

Example 2-8 Thumb code jump table

```
        AREA    Jump, CODE, READONLY
        CODE16                              ; Following code is Thumb code
num     EQU     2
        ENTRY
start
        MOV     r0, #0
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc
stop    MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0xAB                    ; Thumb semihosting SWI
arithfunc
        CMP     r0, #num
        BHS     exit                    ; MOV pc, lr cannot be conditional
        ADR     r3, JumpTable
        LSL     r0, r0, #2              ; 3 instructions needed to replace
        LDR     r0, [r3,r0]             ; LDR pc, [r3,r0,LSL#2]
        MOV     pc, r3
        ALIGN                           ; Ensure that the table is aligned on a
                                        ; 4-byte boundary
JumpTable
        DCD     DoAdd
        DCD     DoSub

DoAdd   ADD     r0, r1, r2
exit    MOV     pc, lr
DoSub   SUB     r0, r1, r2
        MOV     pc,lr
        END
```

2.6.2 Loading addresses with LDR Rd, = label

The `LDR Rd,=` pseudo-instruction can load any 32-bit constant into a register. See *Loading with LDR Rd, =const* on page 2-27. It also accepts program-relative expressions such as labels, and labels with offsets.

The assembler converts an `LDR r0,=label` pseudo-instruction by:

- placing the address of `label` in a literal pool (a portion of memory embedded in the code to hold constant values).
- generating a program-relative `LDR` instruction that reads the address from the literal pool, for example:

  ```
  LDR      rn [pc, #offset to literal pool]
                            ; load register n with one word
                            ; from the address [pc + offset]
  ```

  You must ensure that there is a literal pool within range. Refer to *Placing literal pools* on page 2-27 for more information.

Unlike the `ADR` and `ADRL` pseudo-instructions, you can use `LDR` with labels that are outside the current section. If the label is outside the current section, the assembler places a relocation directive in the object code when the source file is assembled. The relocation directive instructs the linker to resolve the address at link time. The address remains valid wherever the linker places the section containing the `LDR` and the literal pool.

Example 2-9 on page 2-35 shows how this works. It is supplied as `ldrlabel.s` in the `examples\asm` subdirectory of the ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The instructions listed in the comments are the ARM instructions that are generated by the assembler.

Example 2-9

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                               ; Mark first instruction to execute
start
        BL      func1                       ; Branch to first subroutine
        BL      func2                       ; Branch to second subroutine
stop    MOV     r0, #0x18                   ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
```

```
        SWI     0x123456                        ; ARM semihosting SWI
func1
        LDR     r0, =start                      ; => LDR R0,[PC, #offset to
                                                ; Litpool 1]
        LDR     r1, =Darea + 12                 ; => LDR R1,[PC, #offset to
                                                ; Litpool 1]
        LDR     r2, =Darea + 6000               ; => LDR R2, [PC, #offset to
                                                ; Litpool 1]
        MOV     pc,lr                           ; Return
        LTORG                                   ; Literal Pool 1
func2
        LDR     r3, =Darea + 6000               ; => LDR r3, [PC, #offset to
                                                ; Litpool 1]
                                                ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004               ; If uncommented produces an
                                                ; error as Litpool 2 is out of range
        MOV     pc, lr                          ; Return
Darea   SPACE   8000                            ; Starting at the current location,
                                                ; clears a 8000 byte area of memory
                                                ; to zero
        END                                     ; Literal Pool 2 is out of range of
                                                ; the LDR instructions above
```

An LDR Rd, =label example: string copying

Example 2-10 on page 2-37 shows an ARM code routine that overwrites one string with another string. It uses the LDR pseudo-instruction to load the addresses of the two strings from a data section. The following are particularly significant:

DCB	The DCB directive defines one or more bytes of store. In addition to integer values, DCB accepts quoted strings. Each character of the string is placed in a consecutive byte. Refer to the assembler chapter in *ADS Tools Guide* for more information.
LDR/STR	The LDR and STR instructions use post-indexed addressing to update their address registers. For example, the instruction: `LDRB    r2,[r1],#1` loads r2 with the contents of the address pointed to by r1 and then increments r1 by 1.

Example 2-10 String copy

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                              ; Mark first instruction to execute
start   LDR     r1, =srcstr                ; Pointer to first string
        LDR     r0, =dststr                ; Pointer to second string
        BL      strcopy                    ; Call subroutine to do copy
stop    MOV     r0, #0x18                  ; angel_SWIreason_ReportException
        LDR     r1, =0x20026               ; ADP_Stopped_ApplicationExit
        SWI     0x123456                   ; ARM semihosting SWI
strcopy
        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

        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```

Converting to Thumb

There is no post-indexed addressing mode for Thumb `LDR` and `STR` instructions. Because of this, you must use an `ADD` instruction to increment the address register after the `LDR` and `STR` instructions. For example:

```
LDRB  r2, [r1]        ; load register 2
ADD   r1, #1          ; increment the address in
                      ; register 1.
```

2.7 Load and store multiple register instructions

The ARM and Thumb instruction sets include instructions that load and store multiple registers to and from memory.

Multiple register transfer instructions provide an efficient way of moving the contents of several registers to and from memory. They are most often used for block copy and for stack operations for context changing at subroutine entry and exit. The advantages of using a multiple register transfer instruction instead of a series of single data transfer instructions include:

- Smaller code size.
- A single instruction fetch overhead, rather than many instruction fetches.
- Only one register writeback cycle is required for a multiple register load or store, as opposed to one for each register.
- On uncached ARM processors, the first word of data transferred by a load or store multiple is always a nonsequential memory cycle, but all subsequent words transferred can be sequential memory cycles. Sequential memory cycles are faster in most systems.

Note

The lowest numbered register is transferred to or from the lowest memory address accessed, and the highest numbered register to or from the highest address accessed. The order of the registers in the register list in the instructions makes no difference.

Use the `-checkreglist` assembler command line option to check that registers in register lists are specified in increasing order. Refer to the assembler chapter in *ADS Tools Guide* for further information.

2.7.1 ARM LDM and STM instructions

The load (or store) multiple instruction loads (stores) any subset of the 16 general-purpose registers from (to) memory, using a single instruction.

Syntax

The syntax of the `LDM` instructions is:

`LDM{`*`cond`*`}`*`address-mode`* `R`*`n`*`{!},`*`reg-list`*`{^}`

where:

`cond` is an optional condition code. Refer to *Conditional execution* on page 2-19 for more information.

`address-mode`
: specifies the addressing mode of the instruction. Refer to *LDM and STM addressing modes* on page 2-41 for details.

`R`*`n`* is the base register for the load operation. The address stored in this register is the starting address for the load operation. Do not specify r15 (pc) as the base register.

`!` specifies base register write back. If this is specified, the address in the base register is updated after the transfer. It is decremented or incremented by one word for each register in the register list.

`register-list`
: is a comma-delimited list of symbolic register names and register ranges enclosed in braces. There must be at least one register in the list. Register ranges are specified with a dash. For example:

```
{r0,r1,r4-r6,pc}
```

Do not specify writeback if the base register `R`*`n`* is in *`register-list`*.

`^` Do not use this option in User or System mode. For details of its use in privileged modes, see Chapter 6 *Handling Processor Exceptions* and the *ARM Architecture Reference Manual*.

The syntax of the `STM` instruction corresponds exactly, except for some details in the effect of the `^` option.

Usage

See *Implementing stacks with LDM and STM* on page 2-42 and *Block copy with LDM and STM* on page 2-44.

2.7.2 LDM and STM addressing modes

There are four different addressing modes. The base register can be incremented or decremented by one word for each register in the operation, and the increment or decrement can occur before or after the operation. The suffixes for these options are:

`IA`	Increment after.
`IB`	Increment before.
`DA`	Decrement after.
`DB`	Decrement before.

There are alternative addressing mode suffixes that are easier to use for stack operations. See *Implementing stacks with LDM and STM* on page 2-42.

2.7.3 Implementing stacks with LDM and STM

The load and store multiple instructions can update the base register. For stack operations, the base register is usually the stack pointer, r13. This means that you can use load and store multiple instructions to implement push and pop operations for any number of registers in a single instruction.

The load and store multiple instructions can be used with several types of stack:

Descending or ascending

The stack grows downwards, starting with a high address and progressing to a lower one (a descending stack), or upwards, starting from a low address and progressing to a higher address (an ascending stack).

Full or empty

The stack pointer can either point to the last item in the stack (a full stack), or the next free space on the stack (an empty stack).

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment/decrement and before/after suffixes. Refer to Table 2-6 for a list of stack-oriented suffixes.

Table 2-6 Suffixes for load and store multiple instructions

Stack type	Push	Pop
Full descending	`STMFD (DB)`	`LDMFD (IA)`
Full ascending	`STMFA (IB)`	`LDMFA (DA)`
Empty descending	`STMED (DA)`	`LDMED (IB)`
Empty ascending	`STMEA (IA)`	`LDMEA (DB)`

For example:

```
    STMFD    r13!, {r0-r5}  ; Push onto a Full Descending Stack
    LDMFD    r13!, {r0-r5}  ; Pop from a Full Descending Stack.
```

Note

The *ARM/Thumb Procedure Call Standard* (ATPCS), and ARM C and C++ compilers always use a full descending stack.

Stacking registers for nested subroutines

Stack operations are very useful at subroutine entry and exit. At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be popped off again.

In addition, if the link register is pushed onto the stack at entry, additional subroutine calls can safely be made without causing the return address to be lost. If you do this, you can also return from a subroutine by popping the pc off the stack at exit, instead of popping lr and then moving that value into the pc. For example:

```
subroutine  STMFD   sp!, {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
             LDMFD   sp!, {r5-r7,pc} ; Pop work registers and pc
```

Note

Use this with care in mixed ARM/Thumb systems. In ARM architecture v4T systems, you cannot change state by popping directly into the program counter.

In ARM architecture v5T and above, you can change state in this way.

2.7.4 Block copy with LDM and STM

Example 2-11 is an ARM code routine that copies a set of words from a source location to a destination by copying a single word at a time. It is supplied as `word.s` in the `examples\asm` subdirectory of the ADS. Refer to *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-11: Block copy

```
            AREA    Word, CODE, READONLY      ; name this block of code
num         EQU     20                        ; set number of words to be copied
            ENTRY                             ; mark the first instruction to call
start
            LDR     r0, =src                  ; r0 = pointer to source block
            LDR     r1, =dst                  ; r1 = pointer to destination block
            MOV     r2, #num                  ; r2 = number of words to copy
wordcopy    LDR     r3, [r0], #4              ; load a word from the source and
            STR     r3, [r1], #4              ; store it to the destination
            SUBS    r2, r2, #1                ; decrement the counter
            BNE     wordcopy                  ; ... copy more
stop        MOV     r0, #0x18                 ; angel_SWIreason_ReportException
            LDR     r1, =0x20026              ; ADP_Stopped_ApplicationExit
            SWI     0x123456                  ; ARM semihosting SWI

            AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

This module can be made more efficient by using `LDM` and `STM` for as much of the copying as possible. Eight is a sensible number of words to transfer at a time, given the number of registers that the ARM has. The number of eight-word multiples in the block to be copied can be found (if r2 = number of words to be copied) using:

```
    MOVS   r3, r2, LSR #3    ; number of eight word multiples
```

This value can be used to control the number of iterations through a loop that copies eight words per iteration. When there are less than eight words left, the number of words left can be found (assuming that r2 has not been corrupted) using:

```
    ANDS   r2, r2, #7
```

Example 2-12 on page 2-45 lists the block copy module rewritten to use `LDM` and `STM` for copying.

Example 2-12

```
            AREA    Block, CODE, READONLY      ; name this block of code
num         EQU     20                         ; set number of words to be copied
            ENTRY                              ; mark the first instruction to call
start
            LDR     r0, =src                   ; r0 = pointer to source block
            LDR     r1, =dst                   ; r1 = pointer to destination block
            MOV     r2, #num                   ; r2 = number of words to copy
            MOV     sp, #0x400                 ; Set up stack pointer (r13)
blockcopy   MOVS    r3,r2, LSR #3              ; Number of eight word multiples
            BEQ     copywords                  ; Less than eight words to move?
            STMFD   sp!, {r4-r11}              ; Save some working registers
octcopy     LDMIA   r0!, {r4-r11}              ; Load 8 words from the source
            STMIA   r1!, {r4-r11}              ; and put them at the destination
            SUBS    r3, r3, #1                 ; Decrement the counter
            BNE     octcopy                    ; ... copy more
            LDMFD   sp!, {r4-r11}              ; Don't need these now - restore
                                               ; originals
copywords   ANDS    r2, r2, #7                 ; Number of odd words to copy
            BEQ     stop                       ; No words left to copy?
wordcopy    LDR     r3, [r0], #4               ; Load a word from the source and
            STR     r3, [r1], #4               ; store it to the destination
            SUBS    r2, r2, #1                 ; Decrement the counter
            BNE     wordcopy                   ; ... copy more
stop        MOV     r0, #0x18                  ; angel_SWIreason_ReportException
            LDR     r1, =0x20026               ; ADP_Stopped_ApplicationExit
            SWI     0x123456                   ; ARM semihosting SWI

            AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

2.7.5 Thumb LDM and STM instructions

The Thumb instruction set contains two pairs of multiple-register transfer instructions:

- `LDM` and `STM` for block memory transfers
- `PUSH` and `POP` for stack operations.

LDM and STM

These instructions can be used to load or store any subset of the low registers from or to memory. The base register is always updated at the end of the multiple register transfer instruction. You must specify the `!` character. The only valid suffix for these instructions is `IA`.

Examples of these instructions are:

```
LDMIA   r1!, {r0,r2-r7}
STMIA   r4!, {r0-r3}
```

PUSH and POP

These instructions can be used to push any subset of the low registers and (optionally) the link register onto the stack, and to pop any subset of the low registers and (optionally) the pc off the stack. The base address of the stack is held in r13. Examples of these instructions are:

```
PUSH   {r0-r3}
POP    {r0-r3}
PUSH   {r4-r7,lr}
POP    {r4-r7,pc}
```

The optional addition of the lr/pc to the register list provides support for subroutine entry and exit.

The stack is always full descending.

Thumb-state block copy example

The block copy example, Example 2-11 on page 2-44, can be converted into Thumb instructions. An example conversion can be found as `tblock.s` in the `examples\asm` subdirectory of the ADS.

Because the Thumb `LDM` and `STM` instructions can access only the low registers, the number of words copied per iteration is reduced from eight to four. In addition, the `LDM`/`STM` instructions can be used to carry out the single word at a time copy, because they update the base pointer after each access. If `LDR`/`STR` were used for this, separate `ADD` instructions would be required to update each base pointer.

Example 2-13

```
        AREA    Tblock, CODE, READONLY   ; Name this block of code
num     EQU     20                       ; Set number of words to be copied
        ENTRY                            ; Mark first instruction to execute
header                                   ; The first instruction to call
        MOV     sp, #0x400               ; Set up stack pointer (r13)
        ADR     r0, start + 1            ; Processor starts in ARM state,
        BX      r0                       ; so small ARM code header used
                                         ; to call Thumb main program
        CODE16                           ; Subsequent instructions are Thumb
start
        LDR     r0, =src                 ; r0 =pointer to source block
        LDR     r1, =dst                 ; r1 =pointer to destination block
        MOV     r2, #num                 ; r2 =number of words to copy
blockcopy
        LSR     r3,r2, #2                ; Number of four word multiples
        BEQ     copywords                ; Less than four words to move?
        PUSH    {r4-r7}                  ; Save some working registers
quadcopy
        LDMIA   r0!, {r4-r7}             ; Load 4 words from the source
        STMIA   r1!, {r4-r7}             ; and put them at the destination
        SUB     r3, #1                   ; Decrement the counter
        BNE     quadcopy                 ; ... copy more
        POP     {r4-r7}                  ; Don’t need these now-restore originals
copywords
        MOV     r3, #3                   ; Bottom two bits represent number
        AND     r2, r3                   ; ...of odd words left to copy
        BEQ     stop                     ; No words left to copy?
wordcopy
        LDMIA   r0!, {r3}                ; load a word from the source and
        STMIA   r1!, {r3}                ; store it to the destination
        SUB     r2, #1                   ; Decrement the counter
        BNE     wordcopy                 ; ... copy more
stop    MOV     r0, #0x18                ; angel_SWIreason_ReportException
        LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
        SWI     0xAB                     ; Thumb semihosting SWI

        AREA    BlockData, DATA, READWRITE
src     DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        END
```

2.8 Using macros

A macro definition is a block of code enclosed between `MACRO` and `MEND` directives. It defines a name that can be used instead of repeating the whole block of code. This has two main uses:

- to make it easier to follow the logic of the source code, by replacing a block of code with a single, meaningful name
- to avoid repeating a block of code several times.

Refer to the assembler chapter in *ADS Tools Guide* for more details.

2.8.1 Test-and-branch macro example

A test-and-branch operation requires two ARM instructions to implement.

You can define a macro definition such as this:

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc

$label  CMP     $reg, #0
        B$cc    $dest
        MEND
```

The line after the `MACRO` directive is the *macro prototype statement*. The macro prototype statement defines the name (TestAndBranch) you use to invoke the macro. It also defines *parameters* ($label, $dest, $reg, and $cc). You must give values to the parameters when you invoke the macro. The assembler substitutes the values you give into the code.

This macro can be invoked as follows:

```
test    TestAndBranch    NonZero, r0, NE
         ...
         ...
NonZero
```

After substitution this becomes:

```
test    CMP     r0, #0
        BNE     NonZero
         ...
         ...
NonZero
```

2.8.2 Unsigned integer division macro example

Example 2-14 shows a macro that performs an unsigned integer division. It takes four parameters:

`$Bot`	The register that holds the divisor.
`$Top`	The register that holds the dividend before the instructions are executed. After the instructions are executed, it holds the remainder.
`$Div`	The register where the quotient of the division is placed. It may be `NULL` ("") if only the remainder is required.
`$Temp`	A temporary register used during the calculation.

Example 2-14

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot            ; Produce an error message if the
        ASSERT  $Top <> $Temp           ; registers supplied are
        ASSERT  $Bot <> $Temp           ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top        ; These three only matter if $Div
            ASSERT  $Div <> $Bot        ; is not null ("")
            ASSERT  $Div <> $Temp       ;
        ENDIF
$Lab
        MOV     $Temp, $Bot             ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1     ; double it until
90      MOVLS   $Temp, $Temp, LSL #1    ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                    ; The b means search backwards
        IF      "$Div" <> ""            ; Omit next instruction if $Div is null
            MOV     $Div, #0            ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp             ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp        ; If we can, do so
        IF      "$Div" <> ""            ; Omit next instruction if $Div is null
            ADC     $Div, $Div, $Div    ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1    ; Halve $Temp,
        CMP     $Temp, $Bot             ; and loop until
        BHS     %b91                    ; less than divisor
        MEND
```

The macro checks that no two parameters use the same register. It also optimizes the code produced if only the remainder is required.

To avoid multiple definitions of labels if `DivMod` is used more than once in the assembler source, the macro uses local labels (90, 91). Refer to the assembler chapter in *ADS Tools Guide* for more information.

Example 2-15 shows the code that this macro produces if it is invoked as follows:

```
ratio  DivMod  r0,r5,r4,r2
```

Example 2-15

```
        ASSERT  r5 <> r4                    ; Produce an error if the
        ASSERT  r5 <> r2                    ; registers supplied are
        ASSERT  r4 <> r2                    ; not all different
        ASSERT  r0 <> r5                    ; These three only matter if $Div
        ASSERT  r0 <> r4                    ; is not null ("")
        ASSERT  r0 <> r2                    ;
ratio
        MOV     r2, r4                      ; Put divisor in $Temp
        CMP     r2, r5, LSR #1              ; double it until
90      MOVLS   r2, r2, LSL #1              ; 2 * r2 > r5
        CMP     r2, r5, LSR #1
        BLS     %b90                        ; The b means search backwards
        MOV     r0, #0                      ; Initialize quotient
91      CMP     r5, r2                      ; Can we subtract r2?
        SUBCS   r5, r5, r2                  ; If we can, do so
        ADC     r0, r0, r0                  ; Double r0

        MOV     r2, r2, LSR #1              ; Halve r2,
        CMP     r2, r4                      ; and loop until
        BHS     %b91                        ; less than divisor
```

2.9 Describing data structures with MAP and FIELD directives

You can use the MAP and FIELD directives to describe data structures. These directives are always used together.

Data structures defined using MAP and FIELD:

- are easily maintainable
- can be used to describe multiple instances of the same structure
- make it easy to access data efficiently.

The MAP directive specifies the base address of the data structure. Refer to the assembler chapter in *ADS Tools Guide* for further information.

The FIELD directive specifies the amount of memory required for a data item, and can give the data item a label. It is repeated for each data item in the structure. Refer to the assembler chapter in *ADS Tools Guide* for further information.

Note

No space in memory is allocated when a map is defined. Use define constant directives (for example, DCD) to allocate space in memory.

2.9.1 Absolute maps

Example 2-16 shows a data structure described using MAP and FIELD. It is located at an absolute (fixed) address, 4096 (0x1000) in this case.

Example 2-16

```
        MAP     4096
consta  FIELD   4         ; consta uses four bytes, and is located at 4096
constb  FIELD   4         ; constb uses four bytes, and is located at 5000
x       FIELD   8         ; x uses eight bytes, and is located at 5004
y       FIELD   8         ; y uses eight bytes, and is located at 5012
string  FIELD   256       ; string can be up to 256 bytes long, starting at 5020
```

You can access data at these locations with LDR or STR instructions, such as:

```
    LDR     r4,constb
```

You can only do this if each instruction is within 4KB (in either direction) of the data item it accesses. Refer to the *ARM Architecture Reference Manual* for details of the LDR and STR instructions.

2.9.2 Relative maps

If you need to access data from more than 4KB away, you can use a register-relative instruction, such as:

```
    LDR     r4,[r9,offset]
```

`offset` is limited to 4096, so r9 must already contain a value within 4KB of the address of the data.

You can access data in the structure described in Example 2-16 from an instruction at any address. This program fragment shows how:

```
    MOV     r9,#4096       ; or #0x1000
    LDR     r4,[r9,constb - 4096]
```

The assembler calculates (`constb - 4096`) for you. However, it is better to redesign the map description as in Example 2-17.

Example 2-17

```
        MAP     0
consta  FIELD   4       ; consta uses four bytes, located at offset 0
constb  FIELD   4       ; constb uses four bytes, located at offset 4
x       FIELD   8       ; x uses eight bytes, located at offset 8
y       FIELD   8       ; y uses eight bytes, located at offset 16
string  FIELD   256     ; string is up to 256 bytes long, starting at offset 24
```

Using the map in Example 2-17, you can access the data structure at the same location as before:

```
    MOV     r9,#4096
    LDR      r4,[r9,constb]
```

This program fragment assembles to exactly the same machine instructions as before. The value of each label is 4096 less than before, so the assembler does not need to subtract 4096 from each label to find the offset. The labels are *relative* to the start of the data structure, instead of being absolute. The register used to hold the start address of the map (r9 in this case) is called the *base register*.

There are likely to be many `LDR` or `STR` instructions accessing data in this data structure. You avoid typing `-4096` repeatedly by using this method. The code is also easier to follow.

This map does not contain the location of the data structure. The location of the structure is determined by the value loaded into the base register at runtime.

The same map can be used to describe many instances of the data structure. These may be located anywhere in memory.

There are restrictions on what addresses can be loaded into a register using the `MOV` instruction. Refer to *Loading addresses into registers* on page 2-30 for details of how to load arbitrary addresses.

Note

r9 is the *static base register* (sb) in the ARM/Thumb Procedure Call Standard. Refer to Chapter 3 *Using the Procedure Call Standard* for further information.

2.9.3 Register-based maps

In many cases, you can use the same register as the base register every time you access a data structure. You can include the name of the register in the base address of the map. Example 2-18 shows such a *register-based map*. The labels defined in the map include the register.

Example 2-18

```
        MAP     0,r9
consta  FIELD   4       ; consta uses four bytes, located at offset 0 (from r9)
constb  FIELD   4       ; constb uses four bytes, located at offset 4
x       FIELD   8       ; x uses eight bytes, located at offset 8
y       FIELD   8       ; y uses eight bytes, located at offset 16
string  FIELD   256     ; string is up to 256 bytes long, starting at offset 24
```

Using the map in Example 2-18, you can access the data structure wherever it is:

```
    ADR     r9,datastart
    LDR     r4,constb      ; => LDR r4,[r9,#4]
```

`constb` contains the offset of the data item from the start of the data structure, and also includes the base register. In this case the base register is r9, defined in the `MAP` directive.

2.9.4 Program-relative maps

You can use the program counter (r15) as the base register for a map. In this case, each `STM` or `LDM` instruction must be within 4KB of the data item it addresses, because the offset is limited to 4KB. The data structure must be in the same section as the instructions, because otherwise there is no guarantee that the data items will be within range after linking.

Example 2-19 shows a program fragment with such a map. It includes a directive which allocates space in memory for the data structure, and an instruction which accesses it.

Example 2-19

```
datastruc   SPACE   280         ; reserves 280 bytes of memory for datastruc
            MAP     datastruc
consta      FIELD   4
constb      FIELD   4
x           FIELD   8
y           FIELD   8
string      FIELD   256

code        LDR     r2,constb   ; => LDR r2,[pc,offset]
```

In this case, there is no need to load the base register before loading the data as the program counter already holds the correct address. (This is not actually the same as the address of the `LDR` instruction, because of pipelining in the processor. However, the assembler takes care of this for you.)

2.9.5 Finding the end of the allocated data

You can use the `FIELD` directive with an operand of 0 to label a location within a structure. The location is labeled, but the location counter is not incremented.

The size of the data structure defined in Example 2-20 depends on the values of `MaxStrLen` and `ArrayLen`. If these values are too large, the structure overruns the end of available memory.

Example 2-20 uses:

- an `EQU` directive to define the end of available memory
- a `FIELD` directive with an operand of 0 to label the end of the data structure.

An `ASSERT` directive checks that the end of the data structure does not overrun the available memory.

Example 2-20

```
StartOfData     EQU     0x1000
EndOfData       EQU     0x2000
                MAP     StartOfData
Integer         FIELD   4
Integer2        FIELD   4
String          FIELD   MaxStrLen
Array           FIELD   ArrayLen*8
BitMask         FIELD   4
EndOfUsedData   FIELD   0
                ASSERT  EndOfUsedData <= EndOfData
```

2.9.6 Forcing correct alignment

You are likely to have problems if you include some character variables in the data structure, as in Example 2-21. This is because a lot of words are misaligned.

Example 2-21

```
StartOfData     EQU     0x1000
EndOfData       EQU     0x2000
                MAP     StartOfData
Char            FIELD   1
Char2           FIELD   1
Char3           FIELD   1
Integer         FIELD   4        ; alignment = 3
Integer2        FIELD   4
String          FIELD   MaxStrLen
Array           FIELD   ArrayLen*8
BitMask         FIELD   4
EndOfUsedData   FIELD   0
                ASSERT  EndOfUsedData <= EndOfData
```

You cannot use the `ALIGN` directive, because the `ALIGN` directive aligns the current location within memory. `MAP` and `FIELD` directives do not allocate any memory for the structures they define.

You could insert a dummy `FIELD 1` after `Char3 FIELD 1`. However, this makes maintenance difficult if you change the number of character variables. You must recalculate the right amount of padding each time.

Example 2-22 on page 2-57 shows a better way of adjusting the padding. The example uses a `FIELD` directive with a 0 operand to label the end of the character data. A second `FIELD` directive inserts the correct amount of padding based on the value of the label. An `:AND:` operator is used to calculate the correct value.

The `(-EndOfChars):AND:3` expression calculates the correct amount of padding:

```
0 if EndOfChars is 0 mod 4;
3 if EndOfChars is 1 mod 4;
2 if EndOfChars is 2 mod 4;
1 if EndOfChars is 3 mod 4.
```

This automatically adjusts the amount of padding used whenever character variables are added or removed.

Example 2-22

```
StartOfData     EQU     0x1000
EndOfData       EQU     0x2000
                MAP     StartOfData
Char            FIELD   1
Char2           FIELD   1
Char3           FIELD   1
EndOfChars      FIELD   0
Padding         FIELD   (-EndOfChars):AND:3
Integer         FIELD   4
Integer2        FIELD   4
String          FIELD   MaxStrLen
Array           FIELD   ArrayLen*8
BitMask         FIELD   4
EndOfUsedData   FIELD   0
                ASSERT  EndOfUsedData <= EndOfData
```

2.9.7 Using register-based MAP and FIELD directives

Register-based `MAP` and `FIELD` directives define register-based symbols. There are two main uses for register-based symbols:

- defining structures similar to C structures
- gaining faster access to memory sections described by non-register-based `MAP` and `FIELD` directives.

Defining register-based symbols

Register-based symbols can be very useful, but you must be careful when using them. As a general rule, use them only in the following ways:

- As the location for a load or store instruction to load from or store to. If *`Location`* is a register-based symbol based on the register Rb and with numeric offset, the assembler automatically translates, for example, `LDR Rn,`*`Location`* into `LDR Rn,[Rb,#offset]`.

  In an `ADR` or `ADRL` instruction, `ADR Rn,`*`Location`* is converted by the assembler into `ADD Rn,Rb,#offset`.

- Adding an ordinary numeric expression to a register-based symbol to get another register-based symbol.

- Subtracting an ordinary numeric expression from a register-based symbol to get another register-based symbol.

- Subtracting a register-based symbol from another register-based symbol to get an ordinary numeric expression. Do not do this unless the two register-based symbols are based on the same register. Otherwise, you have a combination of two registers and a numeric value. This results in an assembler error.

- As the operand of a `:BASE:` or `:INDEX:` operator. These operators are mainly of use in macros.

Other uses usually result in assembler error messages. For example, if you write `LDR Rn,=`*`Location`*, where *`Location`* is register-based, you are asking the assembler to load `Rn` from a memory location that always has the current value of the register `Rb` plus offset in it. It cannot do this, because there is no such memory location.

Similarly, if you write `ADD Rd,Rn,#`*`expression`*, and *`expression`* is register-based, you are asking for a single `ADD` instruction that adds both the base register of the expression and its offset to `Rn`. Again, the assembler cannot do this. You must use two `ADD` instructions to perform these two additions.

Setting up a C-type structure

There are two stages to using structures in C:

1. Declaring the fields that the structure contains.
2. Generating the structure in memory and using it.

For example, the following **`typedef`** statement defines a point structure that contains three **`float`** fields named x, y and z, but it does not allocate any memory. The second statement allocates three structures of type `Point` in memory, named origin, oldloc, and newloc:

```
typedef struct Point
{
    float x,y,z;
} Point;

Point origin,oldloc,newloc;
```

The following assembly language code is equivalent to the **`typedef`** statement above:

```
PointBase   RN      r11
            MAP     0,PointBase
Point_x     FIELD   4
Point_y     FIELD   4
Point_z     FIELD   4
```

The following assembly language code allocates space in memory. This is equivalent to the last line of C code:

```
origin  SPACE   12
oldloc  SPACE   12
newloc  SPACE   12
```

You must load the base address of the data structure into the base register before you can use the labels defined in the map. For example:

```
        LDR     PointBase,=origin
        MOV     r0,#0
        STR     r0,Point_x
        MOV     r0,#2
        STR     r0,Point_y
        MOV     r0,#3
        STR     r0,Point_z
```

is equivalent to the C code:

```
origin.x = 0;
origin.y = 2;
origin.z = 3;
```

Making faster access possible

To gain faster access to a section of memory:

1. Describe the memory section as a structure.
2. Use a register to address the structure.

For example, consider the definitions in Example 2-23.

Example 2-23

```
StartOfData     EQU     0x1000
EndOfData       EQU     0x2000
                MAP     StartOfData
Integer         FIELD   4
String          FIELD   MaxStrLen
Array           FIELD   ArrayLen*8
BitMask         FIELD   4
EndOfUsedData   FIELD   0
                ASSERT  EndOfUsedData <= EndOfData
```

If you want the equivalent of the C code:

```
Integer = 1;
String = "";
BitMask = 0xA000000A;
```

With the definitions from Example 2-23, the assembly language code could be as shown in Example 2-24.

Example 2-24

```
        MOV     r0,#1
        LDR     r1,=Integer
        STR     r0,[r1]
        MOV     r0,#0
        LDR     r1,=String
        STRB    r0,[r1]
        MOV     r0,#0xA000000A
        LDR     r1,=BitMask
        STRB    r0,[r1]
```

Example 2-24 uses `LDR` *pseudo-instructions*. Refer to *Loading with LDR Rd, =const* on page 2-27 for an explanation of these.

Example 2-24 contains separate `LDR` pseudo-instructions to load the address of each of the data items. Each `LDR` pseudo-instruction is converted to a separate instruction by the assembler. However, it is possible to access the entire data section with a single `LDR` pseudo-instruction. Example 2-25 shows how to do this. Both speed and code size are improved.

Example 2-25

```
                AREA    data, DATA
StartOfData     EQU     0x1000
EndOfData       EQU     0x2000
DataAreaBase    RN      r11
                MAP     0,DataAreaBase
StartOfUsedData FIELD   0
Integer         FIELD   4
String          FIELD   MaxStrLen
Array           FIELD   ArrayLen*8
BitMask         FIELD   4
EndOfUsedData   FIELD   0
UsedDataLen     EQU     EndOfUsedData - StartOfUsedData
                ASSERT  UsedDataLen <= (EndOfData - StartOfData)

                AREA    code, CODE
                LDR     DataAreaBase,=StartOfData
                MOV     r0,#1
                STR     r0,Integer
                MOV     r0,#0
                STRB    r0,String
                MOV     r0,#0xA000000A
                STRB    r0,BitMask
```

Note

In this example, the `MAP` directive is:

```
MAP 0, DataAreaBase
```

not:

```
MAP StartOfData,DataAreaBase
```

The `MAP` and `FIELD` directives give the position of the data relative to the DataAreaBase register, not the absolute position. The `LDR DataAreaBase,=StartOfData` statement provides the absolute position of the entire data section.

If you use the same technique for a section of memory containing memory-mapped I/O (or whose absolute addresses must not change for other reasons), you must take care to keep the code maintainable.

One method is to add comments to the code warning maintainers to take care when modifying the definitions. A better method is to use definitions of the absolute addresses to control the register-based definitions.

Using `MAP` *`offset`*`,`*`reg`* followed by *`label`* `FIELD 0` makes *`label`* into a register-based symbol with register part *`reg`* and numeric part *`offset`*. Example 2-26 shows this.

Example 2-26

```
StartOfIOArea   EQU     0x1000000
SendFlag_Abs    EQU     0x1000000
SendData_Abs    EQU     0x1000004
RcvFlag_Abs     EQU     0x1000008
RcvData_Abs     EQU     0x100000C
IOAreaBase      RN      r11
                MAP     (SendFlag_Abs-StartOfIOArea),IOAreaBase
SendFlag        FIELD   0
                MAP     (SendData_Abs-StartOfIOArea),IOAreaBase
SendData        FIELD   0
                MAP     (RcvFlag_Abs-StartOfIOArea),IOAreaBase
RcvFlag         FIELD   0
                MAP     (RcvData_Abs-StartOfIOArea),IOAreaBase
RcvData         FIELD   0
```

Load the base address with `LDR IOAreaBase,=StartOfIOArea`. This allows the individual locations to be accessed with statements like `LDR R0,RcvFlag` and `STR R4,SendData`.

2.9.8 Using two register-based structures

Sometimes you need to operate on two structures of the same type at the same time. For example, if you want the equivalent of the pseudo-code:

```
newloc.x = oldloc.x + (value in r0);
newloc.y = oldloc.y + (value in r1);
newloc.z = oldloc.z + (value in r2);
```

The base register needs to point alternately to the `oldloc` structure and to the `newloc` one. Repeatedly changing the base register would be inefficient. Instead, use a non register-based map, and set up two pointers in two different registers as in Example 2-27:

Example 2-27

```
        MAP     0               ; Non-register based relative map used twice, for
Pointx  FIELD   4               ; old and new data at oldloc and newloc
Pointy  FIELD   4               ; oldloc and newloc are labels for
Pointz  FIELD   4               ; memory allocated in other sections

        ; code

        ADR     r8,oldloc
        ADR     r9,newloc
        LDR     r3,[r8,Pointx]  ; load from oldloc (r8)
        ADD     r3,r3,r0
        STR     r3,[r9,Pointx]  ; store to newloc (r9)
        LDR     r3,[r8,Pointy]
        ADD     r3,r3,r1
        STR     r3,[r9,Pointy]
        LDR     r3,[r8,Pointz]
        ADD     r3,r3,r2
        STR     r3,[r9,Pointz]
```

2.9.9 Avoiding problems with MAP and FIELD directives

Using `MAP` and `FIELD` directives can help you to produce maintainable data structures. However, this is only true if the order the elements are placed in memory is not important to either the programmer or the program.

You can have problems if you load or store multiple elements of a structure in a single instruction. These problems arise in operations such as:

- loading several single-byte elements into one register
- using a store multiple or load multiple instruction (`STM` and `LDM`) to store or load multiple words from or to multiple registers.

These operations require the data elements in the structure to be contiguous in memory, and to be in a specific order. If the order of the elements is changed, or a new element is added, the program is broken in a way that cannot be detected by the assembler.

There are several methods for avoiding problems such as this.

Example 2-28 shows a sample structure.

Example 2-28

```
MiscBase        RN      r10
                MAP     0,MiscBase
MiscStart       FIELD   0
Misc_a          FIELD   1
Misc_b          FIELD   1
Misc_c          FIELD   1
Misc_d          FIELD   1
MiscEndOfChars  FIELD   0
MiscPadding     FIELD   (-:INDEX:MiscEndOfChars) :AND: 3
Misc_I          FIELD   4
Misc_J          FIELD   4
Misc_K          FIELD   4
Misc_data       FIELD   4*20
MiscEnd         FIELD   0
MiscLen         EQU     MiscEnd-MiscStart
```

There is no problem in using `LDM`/`STM` instructions for accessing single data elements that are larger than a word (for example, arrays). An example of this is the 20-word element `Misc_data`. It could be accessed as follows:

```
ArrayBase   RN      R9
            ADR     ArrayBase, MiscBase
            LDMIA   ArrayBase, {R0-R5}
```

Example 2-28 on page 2-64 loads the first six items in the array Misc_data. The array is a single element and therefore covers contiguous memory locations. It is unlikely that in the future anyone will split it into separate arrays.

However, for loading `Misc_I`, `Misc_J`, and `Misc_K` into registers r0, r1, and r2 the following code would work, but could cause problems in the future:

```
ArrayBase   RN      R9

            ADR     ArrayBase, Misc_I
            LDMIA   ArrayBase, {R0-R2}
```

Problems arise if the order of `Misc_I`, `Misc_J`, and `Misc_K` is changed, or if a new element `Misc_New` is added in the middle. Either of these small changes breaks the code.

If these elements need to be accessed separately elsewhere, you must not amalgamate them into a single array element. In this case, you must amend the code. The first remedy is to comment the structure to prevent changes affecting this section:

```
Misc_I      FIELD   4    ;  ==} Do not split/reorder
Misc_J      FIELD   4    ;    } these 3 elements, STM
Misc_K      FIELD   4    ;  ==} and LDM instructions used.
```

If the code is strongly commented, no deliberate changes are likely to be made that would affect the workings of the program. Unfortunately, mistakes can still occur. A second method of catching these problems would be to add `ASSERT` directives just before the `STM`/`LDM` instructions to check that the labels are consecutive and in the correct order:

```
ArrayBase   RN      R9

                            ; Check that the structure elements
                            ; are correctly ordered for LDM
    ASSERT  (((Misc_J-Misc_I) = 4) :LAND: ((Misc_K-Misc_J) = 4))
            ADR     ArrayBase, Misc_I
            LDMIA   ArrayBase, {R0-R2}
```

This `ASSERT` directive stops assembly at this point if the structure is not in the correct order to be loaded with an `LDM`. Remember that the element with the lowest address is always loaded from, or stored to, the lowest numbered register.

2.10 Using frame directives

If you are using the *ARM/Thumb Procedure Call Standard* (ATPCS), you must use frame directives to describe the way that your code uses the stack. Refer to the assembler chapter in *ADS Tools Guide* for details of these directives.

The assembler uses these directives to insert debug frame information into the object file in ELF format that it produces. This information is required by the debuggers for stack unwinding. Refer to Chapter 3 *Using the Procedure Call Standard* for further information about stack unwinding.

Frame directives do not affect the code produced by `armasm`.

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 3-8.

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
- 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 3-10
- *Read-only position independence* on page 3-13
- *Read-write position independence* on page 3-14
- *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. 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 *ADS Tools Guide*.

3.1.3 Conformance to the ATPCS

Routines compiled using the ADS compilers 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 `-apcs` option 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 machine registers and contents of the memory that it can address.

A process can address some or all of these 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 3-8. Between subroutine calls you can use r0-r3 for any purpose. A called routine need not restore r0-r3 before returning.
- 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 v1-v4 for local variables.

  A called routine must restore the values of these registers before returning, if it has used them. It is not necessary to restore any registers that have not been altered.
- Register r12 is the intra-procedure-call scratch register, ip. It is used in this role in procedure linkage veneers. Between procedure calls you can use it for any purpose.
- 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.

Frame pointers

If you use a frame pointer, use r11 in ARM state, or any one of r4-r7 in Thumb state. You can refer to r11 as fp to make this usage apparent.

If you use a frame pointer in a routine, you cannot use the same register for any other purpose in that routine. In other routines, that do not use the frame pointer, you may use the register for any purpose.

3.2.2 Register names

Table 3-1 lists the defined roles of the processor registers, and associated names. These names are predefined in both the compilers and the assembler.

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	fp	ARM-state variable register 8. ARM-state frame pointer.
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	wr	Variable register 4. Thumb-state work register.
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 coprocessors, see *The VFP architecture* on page 3-18 and *The FPA architecture* on page 3-20.

3.3 The stack

This section describes how to use the stack in the base standard. See also *Stack limit checking* on page 3-10.

ATPCS specifies a full, descending stack.

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*	is 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*	is the lowest address on the stack that the current process is allowed to use.
The *used stack*	is 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*	is the region of memory between the stack pointer and the stack limit. It includes the stack limit but not the stack pointer.
Activation records	are regions of memory allocated on the stack by routines for saving registers and holding local variables.

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

An interrupt handler may 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.

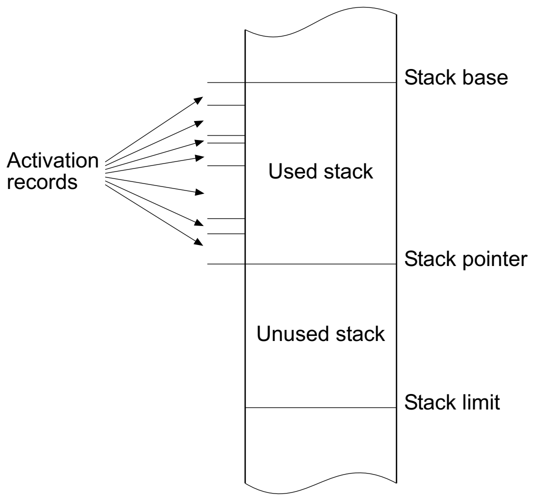

Figure 3-1 Stack memory layout

3.3.2 Stack unwinding

Object files generated by the compilers contain debug frame information. The debuggers use this information to unwind the stack when necessary during debug.

In assembly language it is the responsibility of the programmer to include debug frame information in source code. See the assembly language chapter in *ADS Tools 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 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, in memory, or split between integer registers and memory.

3.4.2 Nonvariadic routines

Machine-level parameter values are passed to a nonvariadic routine as if:

1. The first four integer values are assigned to a1-a4.
2. The first N floating-point values are assigned to floating-point registers of the appropriate precision. The details depend on the selected floating-point architecture (see *Floating-point options* on page 3-17).
3. Remaining values are pushed onto the stack in reverse order.

Note

A machine-level floating-point value is passed in a floating-point register or in memory, never in integer registers.

3.4.3 Result return

A procedure does not return a result.

A function may 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 or d0.
- A compound floating-point value (such as **`complex`**) in f0-fN, or d0-dN. 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* (`/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* (`/noswst`) option only if you can accurately calculate, at the design stage, the maximum amount of stack memory that your complete program requires.

It is possible for stack limit checking to be irrelevant. The code in a file may not require stack limit checking, but be compatible with other code assembled either `/swst` or `/noswst`. Use the *software stack limit checking not applicable* (`/swstna`) option in this case. This is the default.

3.5.1 Rules for stack limit checked code

In the stack limit checked variants of the ATPCS:

- 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 allow 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 sp must always be greater than or equal to the value in sl.

3.5.2 Register usage with stack limit checking

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

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 (`/swst`) option, the compilers generate object code that performs stack checking.

3.5.4 Stack checking in assembly language

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

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

There are three cases to consider:

- *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 3-12.

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 need to check the stack limit. This is a consequence of the rules above (see *Rules for stack limit checked code* on page 3-10).

For this purpose, a leaf routine may 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
    BLO     __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                  ; ip is the intraprocedure
                                ; call register
BLLO    __ARM_stack_overflow
```

or in Thumb code:

```
LDR     wr, #-size              ; Thumb code version
ADD     wr, sp                  ; wr is the Thumb-state
                                ; work register
CMP     wr, sl
BLO     __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.

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 in an ROPI segment cannot be stored in an 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 3-14).
- 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.

After you have used an RWPI segment, you must re-initialize it before repositioning it.

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.

3.8 Interworking between ARM and Thumb states

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

- you want to call ARM routines from Thumb routines
- you want to call Thumb routines from ARM routines
- you want the linker to provide the code to handle the changes of state.

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.

`/nointerwork` is the default.

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.

The linker cannot insert interworking call veneers, or patch the call site, for calls to routines in the *same* file. If you include both ARM and Thumb code in the same assembler source file, you must write the code to switch state as necessary. For example code, see the `CODE16` and `CODE32` sections in the assembler chapter of *ADS Tools Guide*.

You cannot include both ARM and Thumb code in the same C or C++ source file.

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 two different 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).

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

The ADS compilers and assembler have five floating-point options:

- `-fpu VFP`
- `-fpu FPA`
- `-fpu softVFP`
- `-fpu softFPA`
- `-fpu none`.

If your target system has floating-point hardware, you must choose either VFP or FPA.

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

- if you require compatibility with an FPA system, 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`.

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 two 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 is 1, and the vector stride is 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.

For example, in passing:

1.0 (double) 2.0 (double) 3.0 (single) 4.0 (double) 5.0 (single) 6.0 (single)

the assignment of parameter values to registers looks like:

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			

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 `FSTM` instruction and restore them using a single `FLDM` 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 are pure little-endian. This is different from FPA architecture.

Big-endian double-precision values are the same, pure big-endian, in both 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-precison, 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.

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 3-8.

A single-precision floating-point result is returned as a bit pattern in a1.

A double-precision floating-point result is returned in a1 and a2. a1 contains the word corresponding to the lower-addressed word of the representation of the value in memory.

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 wish, provided that the code conforms to the requirements of the ARM/Thumb Procedure Call Standard. The ARM compilers always create code that conforms to this standard. If you are writing ARM assembly language modules you must ensure that your code conforms. See Chapter 3 *Using the Procedure Call Standard* for detailed information.

The ARM linker detects when ARM and Thumb modules are being mixed and generates small code sections called *veneers*. A call to a function in the other instruction set is made through a veneer that changes the instruction set state. No veneer is needed on return.

If you are linking several source files together, all your files must use compatible atpcs options. If obviously incompatible options are detected, the linker will produce an error message.

4.1.1 When to use interworking

When you write code for a Thumb-capable ARM processor, you will 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 ARM equivalents. Some operations are not possible in Thumb state. For example, you cannot enable or disable interrupts. A state change is required in order 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 4-7 for an example.

4.1.2 Using the /interwork option

The option `-apcs /interwork` is available for all compilers and assemblers. 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.
- In C or C++, the compiler creates function exit code that returns to the instruction set state of the caller.

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

- Thumb subroutines that might need to return to ARM code
- ARM subroutines that might need to return to Thumb code.

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

- Thumb code that may be interrupted by an exception. The exception forces the processor into ARM state so no veneer is needed.
- Exception handling code that may handle exceptions from Thumb code. No veneer is needed for the return.
- Thumb code that calls ARM subroutines in other files (the veneers belong to the callee, not the caller).
- ARM code that calls Thumb subroutines in other files (the veneers 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 compiled for interworking. You must recompile the called routine for interworking.

For example, Figure 4-1 shows the error that is produced if the ARM routine in Example 4-2 on page 4-11 is compiled and linked without the `-apcs /interwork` option.

```
Error: Invalid call from Thumb code in thumb.o(.text) to ARM symbol arm_function
```

Figure 4-1 Interworking errors

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 Basic 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.

If you use both instruction sets within the same section, it is your responsibility to ensure that instruction set changes and processor state changes coincide in that section. Otherwise, you can use the linker to provide interworking veneers. We recommend that you do so in normal circumstances (see *Assembly language interworking using veneers* on page 4-14).

The following instructions and directives perform the instruction set and processor state changes:

- `BX`, see *The Branch Exchange instruction*
- `CODE16` and `CODE32`, see *Changing the assembler mode* on page 4-7
- `BLX`, `LDR`, `LDM` and `POP` (ARM architecture v5 and above only), see *ARM architecture v5T* on page 4-9.

This section describes these steps in more detail.

4.2.1 The Branch 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 4-9 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:

`R`*`n`* 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.

Usage

- You can also use `BX` for branches that do not change state. You can use this to execute branches that are out of range of the normal branch instructions. Because `BX` takes a 32-bit register operand it can branch anywhere in 32-bit memory. The `B` and `BL` instructions are limited to:
  - ±32MB in ARM state, for both conditional and unconditional `B` and `BL` instructions (and the `BLX label` instruction in architecture v5)
  - ±4MB in Thumb state, for `BL` instructions (and the `BLX label` instruction in architecture v5)
  - ±2KB in Thumb state, for the unconditional `B` instruction
  - –252 to +258 bytes in Thumb state, for the conditional `B` instruction.

Note

The `BX` instruction is only implemented on ARM processors that are Thumb-capable. If you use `BX` to execute long branches your code will fail on processors that are not Thumb-capable. The result of executing a `BX` instruction on a processor that is not Thumb-capable is unpredictable.

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 Assembler chapter in *ADS Tools Guide* for more information on these directives.

Example ARM header

Example 4-1 on page 4-8 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` instruction to load the branch address and set the least significant bit. The `ADR` instruction generates the address by loading r2 with the value `pc+offset`. See *ADS Tools Guide* for more information on the `ADR` 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 simply adds together the contents of two registers.

The final section labeled `stop` uses the semihosting SWI to report normal application exit. Refer to the *ADS Debug Target 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 4-1

```
     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 0x0123456              ; 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 `asm -g addreg.s` at the command prompt to assemble the source file.
3. Type `armlink addreg.o -o addreg` to link the file.
4. Type `armsd addreg` to load the module into the command-line debugger.
5. Type `step` to step through the rest of the program one instruction at a time. After each instruction, type `reg` to display the registers. Watch the processor enter Thumb state. This is denoted by the T in the *Current Program Status Register* (CPSR) changing from a lowercase t to an uppercase T.

4.2.2 ARM architecture v5T

There are additional interworking instructions available in the ARM v5T architecture used by, for example, the ARM10:

`BLX` *`address`*	The processor performs a pc-relative branch and changes state. *`address`* must be within ±32MB of the current pc value in ARM code or within ±4MB of the pc in Thumb code.
`BLX` *`register`*	The processor branches to an address contained in the specified register. The value of bit 0 determines the new processor state.

In ARM architecture v5 and above, `LDR`, `LDM`, and `POP` can also cause a change of instruction set state if the pc is loaded.

For more information, see the *ARM Architecture Reference Manual*.

4.2.3 Data labels in Thumb code areas

You *must* use the `DATA` directive when you define data labels within a Thumb assembler code area.

When the linker relocates a label in a Thumb code area, it assumes that the label represents the address of a Thumb code routine. Consequently the linker sets bit 0 of the label so that the processor is switched to Thumb state if the routine is called with a `BX` instruction.

The linker cannot distinguish between code and data within a code area. If the label represents a data item, rather than an address, the linker adds 1 to the value of the data item.

The `DATA` directive marks a label as pointing to data within a code area and the linker does not add 1 to its value. For example:

```
            AREA code, CODE
            CODE16
Thumb_fn    ; ...
            MOV   pc, lr

Thumb_Data  DATA
            DCB   1, 3, 4, ...
```

The `DATA` directive must be on the same line as the symbol. Refer to the description of the `DATA` directive in the assembler chapter of the *ADS Tools Guide* for more information.

4.3 C and C++ interworking and veneers

When you compile a C or C++ source file, the object file produced contains either ARM code or Thumb code. It cannot contain both. Interworking between objects must use veneers provided by the linker.

You can freely mix C and C++ code compiled for ARM and Thumb, but 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 all ARM and Thumb C and C++ compilers to compile modules containing routines that can be called by routines compiled for the other processor state:

```
tcc -apcs /interwork
armcc -apcs /interwork
tcpp -apcs /interwork
armcpp -apcs /interwork
```

Modules that are compiled for interworking generate slightly larger code, typically 2% larger for Thumb and less than 1% larger for ARM.

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 the Thumb 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 will never 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 cannot be used on ARM processors that are not Thumb-capable because these processors do not implement the `BX` instruction.

Use the `armlink -info veneers` or `-info sizes,veneers` option to find the amount of space taken by the veneers.

C interworking example

The two modules in Example 4-2 can be built to produce an application where `main()` is 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.

Example 4-2

```
/*********************
*       thumb.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);
}
```

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

To compile and link these modules:

1. Type `tcc -c -apcs /interwork -o thumb.o thumb.c` at the system prompt to compile the Thumb code for interworking.

2. Type `armcc -c -apcs /interwork -o arm.o arm.c` to compile the ARM code for interworking.

3. Type `armlink -o hello arm.o thumb.o` to link the object files.

   Alternatively, type `armlink -info veneers arm.o thumb.o` to view the size of the interworking veneers (Example 4-3).

Example 4-3

```
Adding veneers to the image

    Adding AT veneer (12 bytes) for call to '_printf' from arm.o(.text).
    Adding TA veneer (12 bytes) for call to 'arm_function' from thumb.o(.text).
    Adding AT veneer (12 bytes) for call to '__rt_lib_shutdown' from
kernel.o(x$codeseg).
    Adding AT veneer (12 bytes) for call to '_sys_exit' from kernel.o(x$codeseg).
    Adding AT veneer (12 bytes) for call to '__raise' from rt_raise.o(x$codeseg).
    Adding AT veneer (12 bytes) for call to `_no_fp_display' from
printf2.o(x$fpl$printf2).

6 Veneer(s) (total 72 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 need to return to 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 C/C++ 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.

Note

You must take great care when using function pointers in applications that contain both ARM and Thumb code. The linker warns you about potential illegal calls, but it cannot check them exactly. It is your responsibility to ensure that they are correct.

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 `/interwork` option as it is not guaranteed that the Thumb version will only be called from Thumb state and the ARM version will only be called from ARM state.

The linker allows duplicate definitions provided that each definition is of a different type. That is, one definition defines a Thumb routine and the other defines an ARM routine. The linker generates a warning message if there is a duplicate definition of a symbol:

```
Both ARM & Thumb versions of symbol present in image
```

This is a warning to advise you in case you accidentally include two copies of the same routine. If that is what you intended, you can ignore the warning.

4.4 Assembly language interworking using veneers

The assembly language ARM/Thumb interworking method described in *Basic 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, and no requirement to assemble with the `-apcs /interwork` option that the linker uses to decide whether to add an interworking veneer.

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 just as any non-interworking routine, using a `BL` instruction to make the call. A caller routine may be assembled `/interwork` or `/nointerwork`.
- A callee routine using a `BX` instruction to return. A callee routine must be assembled `/interwork`.

Example of assembly language interworking using veneers

Example 4-4 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. Register 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 4-4

```
     ; *****
     ; 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. Type `armsd count` to load the code into the debugger.
5. Type `list 0x8000` at the armsd command prompt to list the code. Example 4-5 on page 4-15 shows the output.

Example 4-5

```
    armsd: list 0x8000
    ArmProg
    +0000 0x00008000: 0xe3a00001  .... : > mov   r0,#1
    +0004 0x00008004: 0xeb000005  .... :   bl    0x8020  ; (ThumbProg + 0x4)
    +0008 0x00008008: 0xe3a02003  . .. :   mov   r2,#3
    +000c 0x0000800c: 0xe3a00018  .... :   mov   r0,#0x18
    +0010 0x00008010: 0xe59f1000  .... :   ldr   r1,0x00008018 ; = #0x00020026
    +0014 0x00008014: 0xef123456  .... :   swi   0x123456
    +0018 0x00008018: 0x00020026  &... :   dcd   0x00020026  &...
```

```
ThumbProg
+0000 0x0000801c: 0x2102      .!   :   mov   r1,#2
+0002 0x0000801e: 0x4770      pG   :   bx    r14
+0004 0x00008020: 0xe59fc000  .... :   ldr   r12,0x00008028 ; = #ThumbProg+0x1
+0008 0x00008024: 0xe12fff1c  ../. :   bx    r12
+000c 0x00008028: 0x0000801d  .... :   andeq r8,r0,r13,lsl r0
_edata
+0000 0x0000802c: 0xe800e800  .... :   stmda r0,{r11,r13-pc}
```

You can see that the linker has added the required ARM-to-Thumb interworking veneer. This is contained in locations `0x8020` to `0x8028`. Location `0x8028` contains the address of the routine being branch-exchanged to, with bit 0 set.

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 `tcc -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. Type `armsd add` to load the code.
5. Type `go` to run the code.
6. Type `list main` to list the code generated for the `main` function.
7. Type `list arm_function` to list the code generated for the subroutine.
8. Type `list arm_function$$Ven$TA` to list the code generated for the Thumb to ARM interworking veneer.

Chapter 5
Mixed Language Programming

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

5.1 Using the inline assemblers

The inline assemblers built into the C and C++ compilers enable you to use most ARM or Thumb assembly language instructions within a C or C++ program. You can use the inline assembler to:

- use features of the target processor that cannot be accessed from C (the PSR for example)
- achieve more efficient code.

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 armcc and armcpp inline assemblers implement, with two exceptions, the full ARM instruction set including generic coprocessor instructions, halfword instructions and long multiply. The tcc and tcpp inline assemblers implement, again with two exceptions, the full Thumb instruction set. See *Differences between the inline assemblers and armasm* on page 5-6 for information on `BX`, `BLX` and `LDM`.

The inline assembler is a high-level assembler. Some low-level features that are available to the ARM assembler armasm, such as branching by writing to pc, are not supported.

5.1.1 Invoking the inline assembler

The ARM C compilers support inline assembly language with the `__asm` specifier.

The ARM C++ compilers support the **`asm`** syntax proposed in the ANSI C++ Standard, with the restriction that the string literal must be a single string. For example:

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

The **`asm`** syntax is supported by the C++ compilers when compiling both C and C++. The **`asm`** statement must be inside a C or C++ function. Do not include comments in the string literal. An **`asm`** statement can be used anywhere a C or C++ statement is expected.

In addition to the **`asm`** syntax, ARM C++ supports the C compiler `__asm` syntax.

The inline assembler is invoked with the 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.

String copying example

Example 5-1 shows how to use labels and branches in a string copy routine.

This code is also in *`install_directory`*`\examples\inline\strcopy.c.`

The syntax of labels inside assembler blocks is the same as in C. Function calls that use `BL` from inline assembly language must specify the input registers, the output registers, and the corrupted registers. In this example, the inputs to `my_strcpy()` are r0 and r1, there are no outputs, and the default ATPCS registers, r0-r3, r12, lr, and PSR, are corrupted.

Example 5-1 String copy

```
#include <stdio.h>

void my_strcpy(char *src, const char *dst)
{
    int ch;
    __asm
    {
    loop:
#ifndef __thumb
        // ARM version
        LDRB    ch, [src], #1
        STRB    ch, [dst], #1
#else
        // Thumb version
        LDRB    ch, [src]
        ADD     src, #1
        STRB    ch, [dst]
        ADD     dst, #1
#endif
        CMP     ch, #0
        BNE     loop
    }
}

int main(void)
{
    const char * a = "Hello world!";
    char b[20];

__asm
    {
        MOV     R0, a
        MOV     R1, b
        BL      my_strcpy, {R0, R1}
    }
    printf("Original string: %s\n", a);
    printf("Copied   string: %s\n", b);
    return 0;
}
```

5.1.2 ARM and Thumb instruction sets

The ARM and Thumb instruction sets are 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`**, or **`int`**. 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.

Physical registers

The inline assemblers allow restricted access to the physical registers. It is illegal to write to pc. Only branches using `B` and `BL` are allowed. In addition, it is inadvisable to intermix inline assembler instructions that use physical registers and complex C or C++ expressions.

The compiler uses r12 (ip) and, in tcc and tcpp, r3 for intermediate results, and r0-r3, r12 (ip), r14 (lr) for function calls while evaluating C expressions, so these cannot be used as physical registers at the same time.

Physical registers, like variables, must be set before they can be read. When physical registers are used the compiler saves and restores C and C++ variables that might be allocated to the same physical register. However, the compiler cannot restore sp, sl, fp, or sb in calling standards where these registers have a defined role.

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 allowed 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 and Thumb 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
```

You cannot branch to labels using `BL`.

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

SWIs and `BL` instructions must specify exactly the calling standard used. Three optional register lists follow 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.

For example:

```
SWI{cond} swi_num, {input_regs}, {output_regs}, {corrupted_regs}
BL{cond} function, {input_regs}, {output_regs}, {corrupted_regs}
```

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.

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 assemblers and armasm

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

- 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 should be regarded as corrupted if the NZCV flags are updated.
- Physical 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.
- Do not use C variables with the same name as a physical register. When accessed in an `__asm` block, the actual register will be used instead of the variable. (It is possible to access the C variable by enclosing the name in parentheses, but this behavior should not be relied upon.)
- `LDM` and `STM` instructions only allow physical registers to be specified in the register list.
- You cannot write to pc. The `BX` and `BLX` instructions are not implemented.
- You should not modify the stack. This is not necessary because the compiler will stack and restore any working registers as required automatically. It is not allowed 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.

  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)}
  ```

- If you are using physical registers, you must ensure that the compiler does not corrupt them when evaluating expressions. For example:

  ```
  __asm
  {
      MOV r0, x
      ADD y, r0, x / y    // (x / y) overwrites r0
                          // with the result.
  }
  ```

  Because the compiler uses a function call to evaluate `x / y`, it:

  — corrupts r2, r3, ip, and lr

  — updates the NZCV flags in the CPSR

  — alters r0 and r1 with the dividend and modulo.

  The value in r0 is lost. You can work around this by using a C variable instead of r0:

  ```
      mov var,x
      add y, var, x / y
  ```

  The compiler can detect the corruption in many cases, for example when it requires a temporary register and the register is already in use:

  ```
  __asm
  {
    MOV ip, #3
    ADDS x, x, #0x12345678    // this instruction is expanded
    ORR x, x, ip
  }
  ```

  The compiler uses ip as a temporary register when it expands the `ADD` instruction, and corrupts the value 3 in ip. An error message is issued.

- Do not use physical registers to address variables, even when it seems obvious that a specific variable is mapped onto a specific register. If the compiler detects this it either generates an error message or puts the variable into another register to avoid conflicts:

  ```
  int bad_f(int x)        // x in r0
  {
      __asm
      {
          ADD r0, r0, #1  // wrongly asserts that x is
                          // still in r0
      }
      return x;             // x in r0
  }
  ```

  This code returns `x` unaltered. The compiler assumes that `x` and r0 are two different variables, despite the fact that `x` is allocated to r0 on both function entry and function exit. As the assembly language code does not do anything useful, it is optimized away. The instruction should be written as:

  ```
      ADD x, x, #1
  ```

- Do not save and restore physical registers that are used by an inline assembler. The compiler will do this for you. If physical 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;
  }
  ```

5.1.5 Examples

Example 5-2 to Example 5-5 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 `install_directory\examples\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.

This code is also in *`install_directory`*`\examples\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;
}
```

Long multiplies

You can use the inline assembler to customize functions that use **`long long`** type. Example 5-4 shows a simple long multiply routine in C.

Example 5-5 shows how you can use inline assembly language to generate different code for the same routine. You can use the inline assembler to write the high word and the low word of the **`long long`** separately.

The inline assembly language code depends on the word ordering of **`long long`** types, because it assumes that the low 32 bits are at offset 0. Change the code if compiling for big-endian.

This code is also in `*install_directory*\examples\inline\smull.c`.

Example 5-4 Multiply in C

Writing the multiply routine in C:

```
// long multiply routine in C
long long smull(int x, int y)
{
    return (long long) x * (long long) y;
}
```

The compiler generates the following code:

```
 MOV      a3,a2
 MOV      a2,a1
 SMULL    ip,a2,a3,a1
 MOV      a1,ip
 MOV      pc,lr
```

Example 5-5 Multiply in inline assembly language

Writing the same routine using inline assembly language:

```
long long smull(int x, int y)
{
    long long res;
    __asm { SMULL ((int*)&res)[0], ((int*)&res)[1], x, y }
    return res;
}
```

The compiler generates the following code:

```
    MOV a3,a1
    SMULL a1,a2,a3,a2
    MOV pc,lr
```

5.2 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`** (`LDRB`/`STRB` for Architecture 3)
- `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 5-6 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 5-6 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
```

5.3 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++.

5.3.1 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:

```
// C++ code

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

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`*`\include`, together with the standard C header files, and can be referenced in the usual way. For example:

```
// C++ code

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

In ARM C++, these headers simply `#include` the C headers.

Note

Both the C and C++ standard header files are available as precompiled headers in the compilers in-memory file system. Refer to C compilers in the *ADS Tools Guide* for more information.

5.3.2 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 two ways:

- When the file is `#included`. This is shown in Example 5-7.
- By adding the `extern "C"` statement to the header file. This is shown in Example 5-8.

Example 5-7 Directive before include file

```
// C++ code

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

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

Example 5-8 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. */
```

5.4 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.

5.4.1 General rules for calling between languages

The following general rules apply to calling between C, C++, and assembly language.

You should *not* rely on the following C++ implementation details. These implementation details are subject to change in future releases of ARM C++:

- the way names are mangled
- 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++, non-member functions can be declared as `extern "C"` to specify that they have C linkage. In this release of ADS, 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 ARM/Thumb Procedure Calls 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 (non-member) C++ function, declare it `extern "C"` to give it C linkage.
- Member functions (both static and non-static) always have mangled names.
- 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.
- Non-static 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.

5.4.2 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:

- When an object of type **`struct`** or **`class`** is passed to a function and the object has an explicit copy constructor, the object will be copied by the calling code or by the subroutine (callee). If the constructor is overloaded the caller makes the copy. If the constructor is not overloaded, the callee makes the copy.
- Non-static 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`**.

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 as would be expected from the ARM C compiler 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 *plain old data* (POD) structure.
- References are represented as pointers.
- Pointers to data members and pointers to member functions occupy four bytes. They have the same null pointer representation as normal pointers.
- No distinction is made between pointers to C functions and pointers to C++ (non-member) functions.

Symbol name mangling

ARM C++ mangles external names of functions and static data members in a manner similar to that described in section 7.2c of Ellis, M.A. and Stroustrup, B., *The Annotated C++ Reference Manual* (1990). The linker unmangles symbols in messages.

C names must be declared as `extern "C"` in C++ programs. This is done already for the ARM ANSI C headers. Refer to *Using C header files from C++* on page 5-15 for more information.

5.4.3 Examples

The following sections contain code examples that demonstrate:

- *Calling assembly language from C* on page 5-20
- *Calling C from assembly language* on page 5-21
- *Calling C from C++* on page 5-22
- *Calling assembly language from C++* on page 5-23
- *Calling C++ from C* on page 5-24
- *Calling C++ from assembly language* on page 5-25
- *Calling C++ from C or assembly language* on page 5-27
- *Passing a reference between C and C++* on page 5-26

The examples assume a non software-stack checking ATPCS variant.

Calling assembly language from C

Example 5-9 and Example 5-10 show a C program that uses a call to an assembly language subroutine to copy one string over the top of another string.

Example 5-9 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 5-10 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 5-9 is located in `install_directory\examples\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. Type `armsd -e strtest` execute the example.

Calling C from assembly language

Example 5-11 and Example 5-12 show how to call C from assembly language.

Example 5-11 Defining the function in C

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

Example 5-12 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
    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 5-13 and Example 5-14 show how to call C from C++.

Example 5-13 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 5-14 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 5-15 and Example 5-16 show how to call assembly language from C.

Example 5-15 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 5-16 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 5-17 and Example 5-18 show how to call C++ from C.

Example 5-17 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 5-18 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 5-19 and Example 5-20 show how to call C++ from assembly language.

Example 5-19 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 5-20 Defining assembly language function

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

    EXPORT   f
f
    STMDB  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
    LDMIA  sp!,{pc}
    END
```

Passing a reference between C and C++

Example 5-21 and Example 5-22 show how to pass a reference between C and C++.

Example 5-21 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 5-22 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 5-23, Example 5-24 and Example 5-25 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 5-23 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 5-24 Defining the C function

```
struct T;

extern int f__1TFi(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 * f__1TFi(t, 2);    /* like '3 * t->f(2)' */
}
```

Example 5-25 Implementing the function in assembly language

```
EXPORT cfunc
AREA cfunc, CODE
IMPORT  f__1TFi
STMDB   sp!,{lr}  ; r0 already contains the object pointer
MOV r1, #2
BL f__1TFi
ADD r0, r0, r0, LSL #1   ; multiply by 3
LDMIA sp!,{pc}
END
```

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 Overview

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 allow 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 seven 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 just 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 allows 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 prefetched from an *illegal* address. An illegal address is 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.
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.

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 upon exit. You can do this by saving the contents of any registers the handler needs 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. See Chapter 7 *Writing Code for ROM* for more information.

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 assembler chapter in *ADS Tools 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 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.

6.2.1 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. 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.
3. Sets lr_*mode* to the return address, as defined in *The return address and return instruction* on page 6-7.
4. Sets the program counter to the vector address for the exception. This forces a branch to the appropriate exception handler.

Note

If the application is running on a Thumb-capable processor, the processor response is slightly different. See *Handling exceptions on Thumb-capable processors* on page 6-39 for more details.

6.2.2 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 two 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 6-7 for instructions on how to return from each exception type.

Note

You do not need to return from the reset handler because the reset handler should execute 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 must 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 *Implementing stacks with LDM and STM* on page 2-42 for more general information on stack operations.

6.2.3 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 may not necessarily be the next instruction pointed to by the program counter. This section details the instructions to return correctly from handling code for each type of exception.

Note

See *The return address* on page 6-41 for details of the return address on Thumb-capable processors when an exception occurs in Thumb state.

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. Therefore, storing (pc – 4) in lr_ *mode* 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}^
```

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. Storing (pc – 4) in lr_*mode* causes lr_*mode* to point two instructions beyond where 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}^
```

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 invokes the MMU to load the appropriate virtual memory locations into physical memory. It must then return to the address that caused the exception and reload the instruction. The instruction should now load and execute correctly.

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}^
```

Returning from Data Abort handlers

When a load or store instruction tries to access memory, the program counter has been updated. A stored value of (pc – 4) in `lr_ABT` points to the second instruction beyond the address where the exception was generated. When the MMU has loaded the appropriate address into physical memory, the handler should 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}^
```

6.3 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 two 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 may 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.

6.3.1 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 simply 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 6-22).

Example 6-1 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 6-1

```
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
```

If there is RAM at location zero, the vectors (plus the FIQ handler if required) must be copied down from an area in ROM into the RAM. In this case, you must use load pc instructions, and copy the storage locations, to make the code relocatable.

Example 6-2 copies down the vectors given in Example 6-1 to the vector table in RAM.

Example 6-2

```
    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 install the vector table (see Chapter 7 *Writing Code for ROM*).

6.3.2 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 allow 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 6-3 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 6-37 for further details.

Example 6-3

```
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)
    {
        printf ("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 exception handler.
2. Subtract the address of the corresponding vector.
3. Subtract 0x8 to allow for the pipeline.
4. Logically OR this with 0xe59ff000 (the opcode for `LDR pc, [pc,#offset]`) to produce the value to be placed in the vector.
5. Put the address of the handler into the storage location.

Example 6-4 shows a C routine that implements this method.

Example 6-4

```
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;
unsigned *irqaddr = (unsigned *)0x38;
                       /* For example */
*irqaddr = (unsigned)IRQHandler;
Install_Handler (irqaddr,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 6-37 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 datasheet for your target processor.

6.4 SWI handlers

When the SWI handler is entered, it must establish which SWI is being called. This information is usually stored in bits 0-23 of the instruction itself, as shown in Figure 6-1.

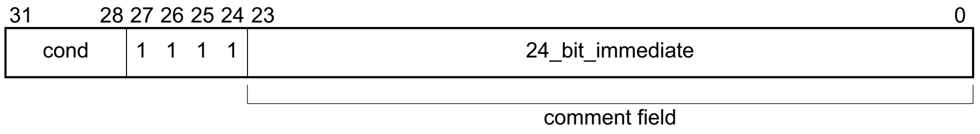

Figure 6-1 ARM SWI instruction

The top-level SWI handler typically accesses the link register and loads the `SWI` instruction from memory, and therefore has to be written in assembly language. The individual routines that implement each SWI handler can be written in C if required.

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 6-5 shows how you can put these instructions together to form a top-level SWI handler.

See *Determining the processor state* on page 6-42 for an example of a handler that deals with both ARM-state and Thumb-state `SWI` instructions.

Example 6-5

```
    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.
```

6.4.1 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 6-6 can be inserted into the top-level handler given in Example 6-5, following on from the `BIC` instruction.

Example 6-6: 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.
```

6.4.2 SWI handlers in C and assembly language

Although the top-level header 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 6-18 for a description of restrictions.

The top-level header 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 6-5:

```
    BL    C_SWI_Handler     ; Call C routine to handle the SWI
```

Example 6-7 shows how the C function can be implemented.

Example 6-7

```
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 may 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 6-2). 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.

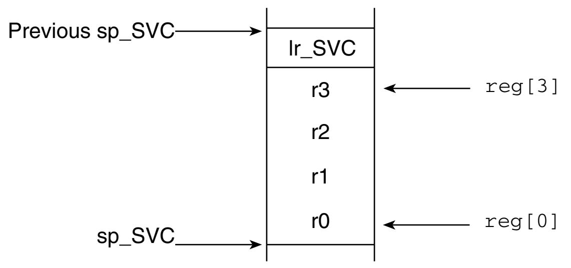

Figure 6-2 Accessing the supervisor stack

6.4.3 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 6-5).

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 needed to return to the instruction following the `SWI` that invoked it. Example 6-8 shows how to do this.

Example 6-8 SWI Handler

```
    STMFD    sp!,{r0-r3,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.
    MOV      r1, sp                ; Second parameter to C routine...
                                   ; ...is pointer to register values.
    MRS      r2, spsr              ; Move the spsr into a general purpose register.
    STMFD    sp!, {r2}             ; Store spsr onto stack. This is
                                   ; only really needed in case of
                                   ; nested SWIs.
    BL       C_SWI_Handler         ; Call C routine to handle the SWI.
    LDMFD    sp!, {r2}             ; Restore spsr from stack into r2...
    MSR      spsr, r2              ; ... and restore it into spsr.
    LDMFD    sp!, {r0-r3,r12,pc}^ ; Restore registers and return.
    END                            ; Mark end of this file.
```

Nested SWIs in C and C++

You can write nested SWIs in C or C++. Code generated by the ARM compilers stores and reloads `lr_SVC` as necessary.

6.4.4 Calling SWIs from an application

The easiest way to call SWIs from your application code is to set up any required register values and call the relevant SWI in assembly language. 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 all ARM instructions.

Calling a SWI from C is more complicated because it is necessary to map a function call onto each SWI with the `__swi` compiler directive. This allows 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 should be placed.

Example 6-9 shows a SWI handler that provides SWI numbers `0x0` and `0x1`. SWI `0x0` takes four integer parameters and returns a single result. SWI `0x1` takes a single parameter and returns four results.

Example 6-9

```
struct four
{    int a, b, c, d;
};

__swi (0x0) int calc_one (int,int,int,int);
__swi (0x1) __value_in_regs struct four calc_four (int);
/* You can call the SWIs in the following manner */
void func (void)
{   struct four result;
    int single, res1, res2, res3, res4;
    single = calc_one (val1, val2, val3, val4);
    result = calc_four (val5);
    res1 = result.a;
    res2 = result.b;
    res3 = result.c;
    res4 = result.d;
}
```

6.4.5 Calling SWIs dynamically from an application

In some circumstances it may 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 would be 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.

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 could be, but need not be, the number of the SWI to be called by the generic SWI.

Example 6-10 shows a C fragment that uses a generic, or *indirect* SWI.

Example 6-10

```
__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
    STR      lr,[sp,#-4]!
    MOV      ip,a3
    SWI      0x80
    LDR      pc,[sp],#4
    EXPORT DoSelectedManipulation
```

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 need 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.

6.5 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 two 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 need for a branch and its associated delays, and also means that if the system has a cache, the vector table and FIQ handler may 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 should contain code to clear the source of the interrupt.

6.5.1 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 store all the required state. In this context, reentrant means that the handler re-enables interrupts, and may itself be interrupted. See *Reentrant interrupt handlers* on page 6-25 for more information.

The `__irq` keyword:

- preserves all APCS 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* on page 6-23 for more information.

Note

C interrupt handlers cannot be produced in this way using tcc. The `__irq` keyword is faulted by tcc because tcc can only produce Thumb code, 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 6-11 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 6-11

```
__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 6-11 produces the following code:

```
IRQHandler
        STMDB    sp!,{a1-v1,ip,lr}
        MOV      v1,#0x80000000
        LDR      a1,[v1,#0]
        CMP      a1,#1
        BLEQ     C_int_handler
        MOV      a1,#0
        STR      a1,[v1,#4]
        LDMIA    sp!,{a1-v1,ip,lr}
        SUBS     pc,lr,#4

        EXPORT IRQHandler
```

Compare this to the result of *not* using the `__irq` keyword:

```
IRQHandler
        STMDB    sp!,{v1,lr}
        MOV      v1,#0x80000000
        LDR      a1,[v1,#0]
        CMP      a1,#1
        BLEQ     C_int_handler
        MOV      a1,#0
        STR      a1,[v1,#4]
        LDMIA    sp!,{v1,pc}

        EXPORT IRQHandler
```

6.5.2 Reentrant interrupt handlers

Note

The following method works for both IRQ and FIQ interrupts. However, because FIQ interrupts are meant to be serviced as quickly as possible there will normally be only one interrupt source, so it may not be necessary to allow for reentrancy.

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 store all the state information required for reentrant interrupt handlers, so you must write your top level interrupt handler in assembly language.

A reentrant interrupt handler must save the necessary IRQ state, switch processor modes, and save the state for the new processor mode before branching to a nested subroutine or C function.

In ARM architecture v4 or later you can switch to System mode. System mode uses the User mode registers, and allows privileged access that may be required by your exception handler. See *System mode* on page 6-44 for more information. In ARM architectures prior to ARM architecture v4 you must switch to Supervisor mode instead.

The steps needed 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 6-12 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 6-12

```
    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
```

6.5.3 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 6-28
- *Interrupt prioritization* on page 6-29
- *Context switch* on page 6-31.

Single-channel DMA transfer

Example 6-13 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 6-13

```
    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 ?
    SUBLES  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 6-14 is similar to Example 6-13, except that there are two channels being handled (which may be the input and output side of the same channel). 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 6-14

```
    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 6-15 dispatches up to 32 interrupt sources to their appropriate handler routines. Because it is designed for use with the normal interrupt vector (IRQ), it should be 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 6-15

```
; 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 6-16 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 6-3 shows the layout of the PCBs that the example expects.

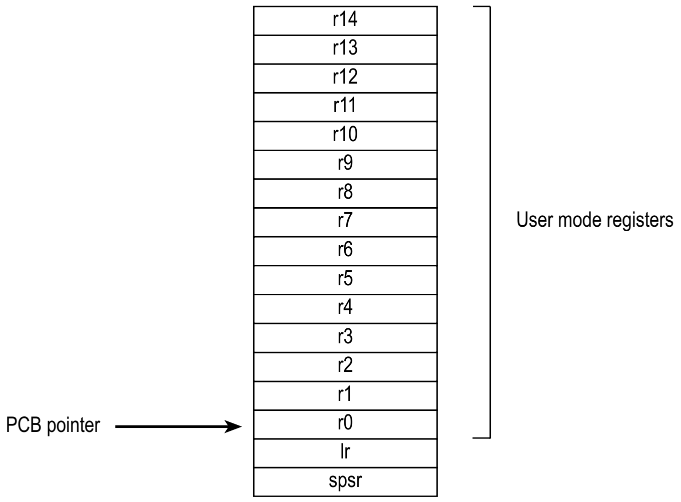

Figure 6-3 PCB layout

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 6-16

```
    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.
```

6.6 Reset handlers

The operations carried out by the Reset handler depend on the system for which the software is being developed. For example, it may:

- Set up exception vectors. See *Installing an exception handler* on page 6-9 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 7 *Writing Code for ROM* for more information.

6.7 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 could 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 should:

1. Attach itself to the Undefined Instruction vector and store the old contents.
2. Examine the undefined instruction to see if it should 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 should handle the instruction, the emulator should process the instruction and return to the user program.
3. Otherwise the emulator should 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 should report an error and quit. See *Chaining exception handlers* on page 6-37 for more information.

——— **Note** ———

The Thumb instruction set does not have coprocessor instructions, so there should be no need for the Undefined Instruction handler to emulate such instructions.

6.8 Prefetch Abort handler

If the system has no MMU, the Prefetch Abort handler can simply 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 should therefore return to the same instruction rather than the following one, for example:

```
    SUBS    pc,lr,#4
```

6.9 Data Abort handler

If there is no MMU, the Data Abort handler should simply report the error and quit. If there is an MMU, the handler should 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.

Three 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 will not have been updated.
  - If the processor is in late abort mode and writeback was requested, the address register will have been updated. The change must be undone.
- If the abort takes place on an ARM7-based processor, including the ARM7TDMI, the address register will have 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 will have been 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 will be 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 *install_directory*`/examples/databort`.

6.10 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 may also wish to implement some SWIs.

In such situations there are two approaches that can be taken to extend the exception handling code:

- *A single extended handler*
- *Several chained handlers*.

6.10.1 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.

6.10.2 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 may well have its own entirely independent exception handler that it wants to install, but which cannot simply 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 allows 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 6-11 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 allow 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 will branch to the original handler.

The location used to store the address of the original handler

If the application handler failed to handle the exception, it would then need to load the program counter from that location.

In most cases, such calculations may not be necessary because information on the debug monitor or RTOS handlers should be 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.

6.11 Handling exceptions on Thumb-capable processors

This section describes the additional considerations you must take into account when writing exception handlers suitable for use on Thumb-capable processors.

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.

Note

This section applies only to Thumb-capable ARM processors.

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* on page 6-5.

6.11.1 Thumb processor response to an exception

When an exception is generated, the processor takes the following actions:

1. Copies cpsr into spsr_*mode*.
2. Switches to ARM state.
3. Sets the CPSR mode bits.
4. Stores the return address in lr_*mode*. See *The return address* on page 6-41 for further details.
5. Sets the program counter to the vector address for the exception. The switch from Thumb state to ARM state in step 1 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.

Handling the exception

Your top-level veneer routine should save the processor status and any required registers on the stack. You then have two 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 6-4. See Chapter 4 *Interworking ARM and Thumb* for details of how to combine ARM and Thumb code in this way.

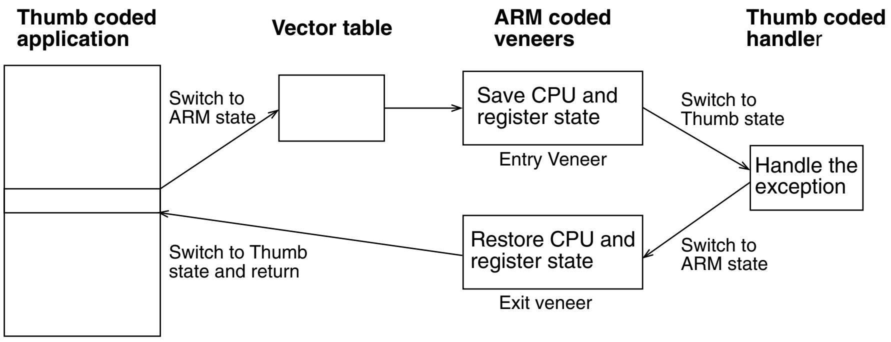

Figure 6-4 Handling an exception in Thumb state

6.11.2 The return address

If an exception occurs in ARM state, the value stored in lr_*mode* is (pc – 4) as described in *The return address and return instruction* on page 6-7. However, if the exception occurs 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 allows the handler to have a single return instruction that will return correctly, regardless of the processor state (ARM or Thumb) at the time the exception occurred.

The following sections give a summary of the values to which the processor sets lr_*mode* if an exception occurs when the processor is in Thumb state.

SWI and Undefined Instruction handlers

The handler's 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).

FIQ and IRQ handlers

The handler's 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.

Prefetch Abort handlers

The handler's 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.

Data Abort handlers

The handler's 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).

6.11.3 Determining the processor state

An exception handler may need to determine whether the processor was in ARM or Thumb state when the exception occurred. SWI handlers, especially, may need 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 three things:

- 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 6-5)
- the SWI number is held in 8 bits instead of the 24 bits in ARM state.

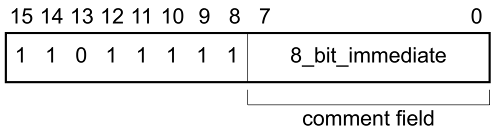

Figure 6-5 Thumb SWI instruction

Example 6-17 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 could 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 upon 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 6-14).

Example 6-17

```
T_bit   EQU    0x20                    ; Thumb bit of CPSR/SPSR, that is, bit 5.
        :
        :
SWIHandler
        STMFD   sp!, {r0-r3,r12,lr}     ; Store the registers.
        MRS     r0, spsr                ; Move SPSR into general purpose
                                        ; register.
        TST     r0, #T_bit              ; Test if bit 5 is set.
        LDRNEH  r0,[lr,#-2]             ; T_bit set so load halfword (Thumb)
        BICNE   r0,r0,#0xff00           ; and clear top 8 bits of halfword
                                        ; (LDRH clears top 16 bits of word).
        LDREQ   r0,[lr,#-4]             ; T_bit clear so load word (ARM)
        BICEQ   r0,r0,#0xff000000       ; and clear top 8 bits of word.

        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
        :
```

6.12 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 will be 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 may not have the privileged access it needs 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 6-25 for an example.

Chapter 7
Writing Code for ROM

This chapter describes how to build ROM images, typically for embedded applications. There are also suggestions on how to avoid the most common errors in writing code for ROM.

This chapter contains the following sections:

7.1 About writing code for ROM

This chapter describes how to write code for ROM, and shows different methods for simple and complex images. Sample initialization code is given, as well as information on initializing data, stack pointers, interrupts, and so on.

This chapter contains examples of using scatter loading to build complex images. For detailed reference information on the linker and scatter loading, refer to *ADS Tools Guide*.

The reference example in `install_directory\ROM\embed` can be built in four different configurations in increasing levels of complexity:

- As a simple semihosted application that links with the C libraries. This example uses the semihosting SWI functions of the C libraries for I/O. See *The reference C example using semihosting* on page 7-11.
- As an application that links with the C libraries and can be embedded into ROM. This example does not use the semihosting SWI functions, but instead uses a retargeting layer for I/O. See *Loading the ROM image at address 0* on page 7-14.
- As an application that uses scatter loading and runs under the ARMulator or can be embedded into ROM. The example displays the linker-generated scatter symbols on the screen. See *Using a simple scatter-loading file* on page 7-23.
- As an application that uses scatter loading and memory remapping to move RAM to `0x0` after initialization. See *Using both scatter-loading and remapping* on page 7-26.

A C++ example is supplied in `install_directory\ROM\embed_cpp`.

The *ARM Reference Peripheral Specification* (RPS) example in `install_directory\ROM\rps_irq` can also be built in four different configurations. Two of these configurations are described in detail:

- As a simple semihosted application that links with the C libraries. This example uses the semihosting SWI functions of the C libraries for I/O. See *The reference C example using semihosting* on page 7-11.
- As an application that uses scatter loading and runs under the ARMulator or can be embedded into ROM. This example does not use the semihosting SWI functions, but instead uses a retargeting layer for I/O. See *Using a simple scatter-loading file* on page 7-23.

CodeWarrior projects are available for the examples as `embed.mcp`, `embed_cpp.mcp`, and `rps_irq.mcp`.

7.2 Memory map considerations

A major consideration in the design of an embedded ARM application is the layout of the memory map, in particular the memory that is situated at address `0x0`. Following reset, the processor starts to fetch instructions from `0x0`, so there must be some executable code accessible from that address. In an embedded system, this requires ROM to be present, at least initially, at address `0x0`.

7.2.1 ROM at 0x0

The simplest layout is to locate the application in ROM at address 0 in the memory map (see Figure 7-1). The application can then branch to the real entry point when it executes its first instruction (at the reset vector at address `0x0`).

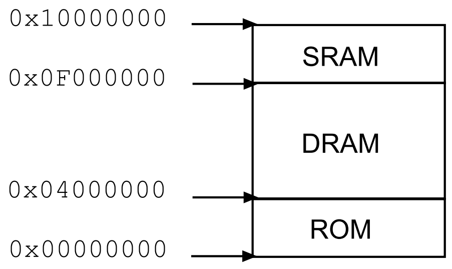

Figure 7-1 Example of a system with ROM at 0x0

However, there are disadvantages with this layout. ROM is typically narrow (8 or 16 bits) and slow (requires more wait states to access it) compared to RAM. This slows down the handling of processor exceptions (especially interrupts) through the vector table. Also, if the vector table is in ROM, it cannot be modified by the code.

For more information on exception handling, see Chapter 6 *Handling Processor Exceptions*.

7.2.2 RAM at 0x0

RAM is normally faster and wider than ROM. For this reason, it is better for the vector table and interrupt handlers if the memory at `0x0` is RAM.

However, if RAM is located at address 0x0 on power-up, there is not a valid instruction in the reset vector entry. Therefore, you must allow ROM to be located at `0x0` at power-up (so there is a valid reset vector), but to also allow RAM to be located at `0x0` during normal execution. The changeover from the reset to the normal memory map is normally caused by writing to a memory-mapped register (see Figure 7-2).

For example, on reset, an aliased copy of ROM is present at `0x0`, but RAM is remapped to zero when code writes to the RPS REMAP register. For more information, refer to the ARM *Reference Peripheral Specification*.

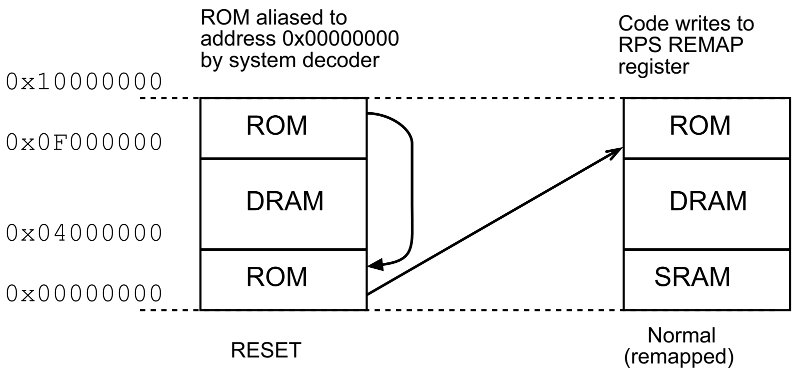
ROM aliased to address 0x00000000 by system decoder
Code writes to RPS REMAP register
0x10000000
0x0F000000
0x04000000
0x00000000
ROM
DRAM
ROM
ROM
DRAM
SRAM
RESET
Normal (remapped)

Figure 7-2 Example of a system with RAM at 0x0

Implementing RAM at 0x0

A sample sequence of events for implementing RAM at `0x0` is:

1. Power on to fetch the RESET vector at `0x0` (from the aliased copy of ROM).
2. Execute the RESET vector:

   ```
   LDR PC, =0x0F000004
   ```

   This causes a jump to the real address of the next ROM instruction. This assembles to a position-independent instruction

   ```
   LDR PC, [PC, offset]
   ```

3. Write to the REMAP register and set REMAP = 1.
4. Complete the rest of the initialization code as described in *Initializing the system* on page 7-6.

System decoder

ROM can be aliased to address `0x0` by the system memory decoder. A simple memory decoder might implement this as:

```
case ADDR(31:24) is
    when "0x00"
        if REMAP = "0" then
            select ROM
        else
            select SRAM
    when "0x0F"
        select ROM
    when ....
```

7.3 Initializing the system

There are two initialization stages:

1. Initializing the execution environment, for example exception vectors, stacks, I/O.
2. Initializing the application, C variables for example.

For a hosted application, the execution environment was initialized when the OS starts (initialization is done by, for example, Angel, an RTOS, or ARMulator). The application is then entered automatically through the `main()` function. The C library code at `__main` initializes the application.

For an embedded application without an operating system, the code in ROM must provide a way for the application to initialize itself and start executing.

No automatic initialization takes place on reset, so the application entry point must perform some initialization before it can call any C code.

Typically, the initialization code, located at address zero after reset, should:

- mark the entry point for the initialization code
- set up exception vectors
- initialize the memory system
- initialize the stack pointer registers
- initialize any critical I/O devices
- initialize any RAM variables required by the interrupt system
- enable interrupts (if handled by the initialization code)
- change processor mode if necessary
- change processor state if necessary.

After the environment has been initialized, the sequence continues with the application initialization and should enter the C code.

These items are described in more detail below. See Example 7-2 on page 7-17 and Example 7-3 on page 7-18 for code examples.

7.3.1 Initializing the execution environment

There are some aspects of the execution environment that must be initialized before the application starts. If the application is hosted by an operating system, the initialization will be done by the application loader. If the application runs standalone, the C library can perform the initialization of the environment and call the application entry point at `main()`.

The state of ARM processor cores after reset is:

- SVC mode
- interrupts disabled
- ARM state.

Identifying the entry point

An executable image must have an entry point. An embedded rommable image usually has an entry point at `0x0`. An entry point can be defined in the initialization code by using the assembler directive `ENTRY`. It is possible to have multiple entry points in an embedded application. When there are multiple entry points, one of the points must be specified as the *initial* entry point by using `-entry`. See also the section on linker selection of entry points in the *ADS Tools Guide*.

If you have created a C program that includes a `main()` function, there is also an entry point within the C library initialization code. See also the library chapter in *ADS Tools Guide* for more information on creating applications that use the library.

Setting up exception vectors

Your initialization code must set up the required exception vectors, as follows:

- If the ROM is located at address 0, the vectors consist of a sequence of hard-coded instructions to branch to the handler for each exception.
- If the ROM is located elsewhere, the vectors must be dynamically initialized by the initialization code (See *Using both scatter-loading and remapping* on page 7-26).

See Example 7-3 for a listing of typical initialization code.

Initializing the memory system

If your system has a Memory Management or Protection Unit, you must make sure that it is initialized:

- *before* interrupts are enabled
- *before* any code is called that might rely on RAM being accessible at a particular address, either explicitly, or implicitly through the use of stack.

Initializing the stack pointers

The initialization code initializes the stack pointer registers. You might have to initialize some or all of the following stack pointers, depending on the interrupts and exceptions you use:

`sp_SVC`	This must always be initialized.
`sp_IRQ`	This must be initialized if IRQ interrupts are used. It must be initialized before interrupts are enabled.
`sp_FIQ`	This must be initialized if FIQ interrupts are used. It must be initialized before interrupts are enabled.
`sp_ABT`	This must be initialized for Data and Prefetch Abort handling.
`sp_UND`	This must be initialized for Undefined Instruction handling.

Generally, `sp_ABT` and `sp_UND` are not used in a simple embedded system. However, you might want to initialize them for debugging purposes.

You can set up the stack pointer `sp_USR` when you change to User mode to start executing the application.

Initializing any critical I/O devices

Critical I/O devices are any devices that you must initialize before you enable interrupts. Typically, you must initialize these devices at this point. If you do not, they might cause spurious interrupts when interrupts are enabled.

Initializing RAM variables required by the interrupt system

If your interrupt system has buffer pointers to read data into memory buffers, the pointers must be initialized before interrupts are enabled.

Enabling interrupts

The initialization code can now enable interrupts if necessary, by clearing the interrupt disable bits in the CPSR. This is the earliest point that it is safe to enable interrupts.

Changing processor mode

At this stage the processor is still in Supervisor mode. If your application runs in User mode, change to User mode and initialize the User mode sp register, `sp_USR`.

Changing processor state

All ARM cores, including Thumb-capable processors, start up in ARM state on reset. The initialization code (at least the reset handler) will be ARM code. If the application is compiled for Thumb, `main()` is Thumb code. The linker can add ARM to Thumb interworking veneers automatically to change state between the ARM initilization code and the Thumb application. You can also write initialization code to manually switch from ARM to Thumb state using:

```
    ORR lr, pc, #1
    BX lr
```

For more details on changing between ARM and Thumb state, see Chapter 4 *Interworking ARM and Thumb*.

7.3.2 Initializing the application

An application is initialized by:

- initializing the non-zero writable data by copying the initializing values to the writable data region
- setting to zero the ZI writable data region.

After memory initialization, control is passed to the entry point of the application in, for example, C library code.

Initializing memory required by C code

The initial values for any initialized variables (RW) must be copied from ROM to RAM. All other ZI variables must be initialized to zero. The library initialization code called at `__main` performs the copying and initialization.

——— **Note** ———

The linker assigns memory addresses for RO code, RW data, and ZI data. If a scatter-load file is not used, the linker uses one of the default scatter load formats. Scatter loading examples are given in *Using a simple scatter-loading file* on page 7-23 and *Using both scatter-loading and remapping* on page 7-26.

Using the main function

When the compiler compiles a function called `main()`, it generates a reference to the symbol `__main` to force the linker to include the basic C run-time system from the ANSI C library.

7.4 The reference C example using semihosting

This example shows an application that uses the semihosting SWIs. `printf()` for example, is compiled as a call to a C library function that uses semihosting SWIs to display information on the debugger console. The application consists of a single C file.

The code for `main.c` is in *`install_directory`*`\Examples\ROM\embed` directory, and is included in Example 7-1 on page 7-13 for reference.

To build the example from the CodeWarrior IDE:

1. Use the CodeWarrior project `embed.mcp`
2. Select `Target=Semihosted`.

To build the example from the command line, execute `build_a.bat` or follow the steps below:

1. Compile the C file `main.c` with one of the following commands:

   `armcc -g -O1 -c main.c` (if compiling for ARM)

   `tcc -g -O1 -c main.c` (if compiling for Thumb)

   where:

`-O1`	specifies the level of optimization.
`-g`	tells the compiler to add debug tables.
`-c`	tells the compiler to compile only (not to link).

2. Link the image using, all on one line, the following command:

   `armlink main.o -o embed.axf`

   where:

`-o`	specifies the output file as `embed.axf`.

3. Use ARMulator to test the image or download the image to a development board using Multi-ICE or Angel.

7.4.1 Memory map

Figure 7-3 shows RAM starting at address 0x08000 (see Figure 7-3).

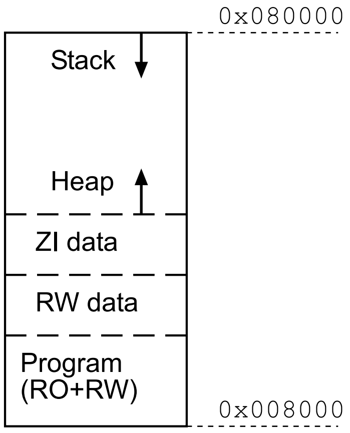

Figure 7-3 Memory map for reference example

By default, the linker sets the start of code at address `0x8000`. The RW data is placed immediately above the program code and the ZI data above the RW data.

By default, the stack pointer sp is initialized to `0x80000000` for ARMulator or `0x80000` (the top of memory as indicated by the value of the debugger internal variable `$top_of_memory`) for remote targets.

7.4.2 Sample code

The C code fragment in Example 7-1 shows the use of semihosting SWIs to output text. See the `main.c` source code for the definitions of `demo_malloc()`, `demo_sscanf()`, `demo_printf()`, `demo_float_print()`, and `demo_sprintf()`.

The code selected by the `#ifdef EMBEDDED` will be used in *Loading the ROM image at address 0* on page 7-14 and other examples.

Example 7-1 extract from main.c

```
/* Copyright (C) ARM Limited, 1999. All rights reserved. */

int main(void)
{
    printf("C Library Example\n");

#ifdef EMBEDDED
/* ensure no C library functions that uses semi-hosting SWIs are linked */
    __use_no_semihosting_swi();
#endif
    demo_printf();
    demo_sprintf();
    demo_float_print();
    demo_malloc();
    demo_sscanf();
    return 0;
}
```

7.5 Loading the ROM image at address 0

This example shows how to convert the code in *The reference C example using semihosting* on page 7-11 into a minimal application that can be embedded in ROM using a retargeting layer. In a real system, more initialization code would be required.

The code for `retarget.c` and `serial.c` is in *`install_directory`*`\Examples\ROM\embed` directory, and is included in *Sample code* on page 7-17 for reference.

To build the example from the CodeWarrior IDE:

1. Use the CodeWarrior project `embed.mcp`
2. Select `Target=Embedded`.

To build the example from the command line, execute `build_b.bat` or follow the steps below:

1. Assemble the initialization code:

   ```
   armasm -g vectors.s
   armasm -g init.s
   ```

2. Compile the `main` example and the new retargeting files `retarget.c` and, optionally, `serial.c` with the following commands:

   ```
   armcc -c -g -O1 main.c -DEMBEDDED
   armcc -c -g -O1 retarget.c
   armcc -c -g -O1 serial.c -I..\include
   ```

   where:

`-D`	tells the compiler to define the symbol `EMBEDDED`.
`-I`	tells the compiler where to find the include files.

3. Link the image using the following command (all on one line):

   ```
   armlink vectors.o init.o main.o retarget.o serial.o
       -ro-base 0x0 -rw-base 0x00040000
       -first vectors.o(Vect) -entry 0x0
        -o embed.axf -info totals -map -list list.txt
   ```

   where:

   `-ro-base 0x0`

   This option tells the linker that the read-only or code segment will be placed at `0x00000000` in the address map.

`-rw-base 0x00040000`	This option tells the linker that the read-write or data segment will be placed at `0x00040000` in the address map. This is the base of the RAM in this example.
`-first vectors.o(Vect)`	This option tells the linker to place this input section first in the image. On UNIX systems you might have to put a backslash `\` before each parenthesis.
`-entry`	This option defines the reset vector as the unique entry point.
`-o`	This option specifies the output file.
`-info totals`	This option tell the linker to print information on the code and data sizes of each object file along with the totals for each type of code or data. The output generated is shown in *Output from list option* on page 7-16.
`-map`	This option tells the linker to print an input section map or listing showing where each code or data section will be placed in the address space.

4. Run the fromELF utility to produce a plain binary version of the image:

```
fromelf embed.axf -bin -o embed.bin
```

where:

`-bin`	specifies a binary output image with no header.

5. Use ARMulator to test the image or download and execute the ROM image to the development board.
   - For armsd use:

     ```
     getfile embed.bin 0x0
     readsyms embed.axf
     ```
   - For ADW & ADU, select:

     File→ Get File and specify `embed.bin` with load address `0x0`.

     File→ Load symbols only and specify `embed.axf`.
   - For AXD select:

     File → Load Memory From File and specify `embed.bin` with load address `0x0`.

     File → Load Debug Symbols and specify `embed.axf`.

7.5.1 Memory map

Figure 7-4 shows:

- ROM is address 0 as specified by `-ro-base`. See Figure 7-4.
- RAM is at `0x040000`, as specified by `-rw-base`, to hold the stack, heap, and data.
- The stack pointer is initialized to `0x80000` in `init.s`.
- The heap base is initialized to `0x060000` by `__user_initial_stackheap()` in `retarget.c`.

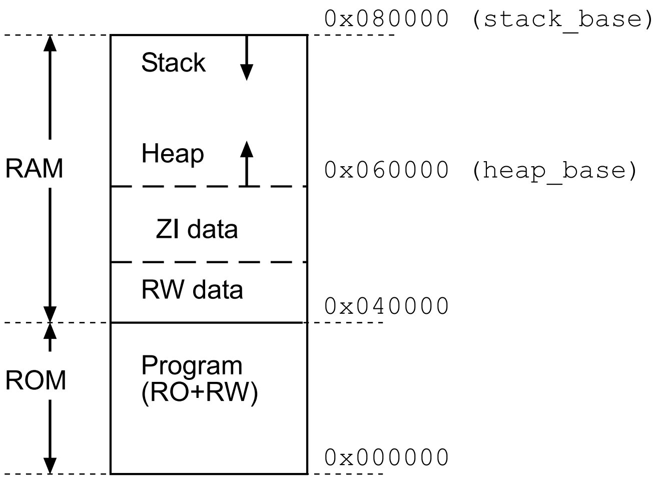

Figure 7-4 Memory map for ROM at address 0

7.5.2 Output from list option

The file `list.txt` shows the map (segment listing) for the sample code:

```
==============================================================
Image component sizes
Code   RO Data  RW Data  ZI Data   Debug
   1224      64        8      12      14024   Object Totals
  23044     728        0      64       8652   Library Totals
================================================================
   Code  RO Data   RW Data  ZI Data   Debug
  24268     792        8      76      22676   Grand Totals
================================================================
  Total RO  Size(Code + RO Data)              25060 (24.47KB)
  Total RW  Size(RW Data + ZI Data)              84 (0.08KB)
  Total ROM Size(Code + RO Data + RW Data)      25068 (24.48KB)
    Total input debug size                       19448 (18.99KB)
    Total output debug size                      17716 (17.30KB)
    Image debug size reduction  8.91 percent
==============================================================
```

7.5.3 Sample code

The code in Example 7-2 contains example exception vectors and exception handlers. For this application, ROM is fixed at `0x0` and the exception table is hard-coded at `0x0`. For *Using a simple scatter-loading file* on page 7-23, ROM/RAM remapping occurs and the vectors are copied from ROM to RAM.

Example 7-2 vectors.s

```
;;; Copyright ARM Ltd 1999. All rights reserved.
    AREA Vect, CODE, READONLY
; *****************
; Exception Vectors
; Note: LDR PC instructions are used here because branch (B) instructions
; could not simply be copied (the branch offsets would be wrong).  Also,
; a branch instruction might not reach if the ROM is at an address >32MB).
    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
    IMPORT  Reset_Handler    ; In init.s
Reset_Addr      DCD     Reset_Handler
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
; ***********************
; Exception Handlers
; The following dummy handlers do not do anything useful in this example.
; They are set up here for completeness.
Undefined_Handler
    B       Undefined_Handler
SWI_Handler
```

```
    B       SWI_Handler
Prefetch_Handler
    B       Prefetch_Handler
Abort_Handler
    B       Abort_Handler
IRQ_Handler
    B       IRQ_Handler
FIQ_Handler
    B       FIQ_Handler
    END
```

The code in Example 7-3 performs ROM/RAM remapping (if required), initializes stack pointers and interrupts for each mode, and finally branches to `__main` in the C library (`__main` eventually calls `main()`). On reset, the ARM core starts up in Supervisor (SVC) mode, in ARM state, with IRQ and FIQ disabled.

Example 7-3 init.s

```
;;; Copyright ARM Ltd 1999. All rights reserved.
    AREA    Init, CODE, READONLY

; --- Standard definitions of mode bits and interrupt (I & F) flags in PSRs
Mode_USR        EQU     0x10
Mode_FIQ        EQU     0x11
Mode_IRQ        EQU     0x12
Mode_SVC        EQU     0x13
Mode_ABT        EQU     0x17
Mode_UNDEF      EQU     0x1B
Mode_SYS        EQU     0x1F ; available on ARM Arch v4 and later
I_Bit           EQU     0x80 ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40 ; when F bit is set, FIQ is disabled

; --- System memory locations
RAM_Limit       EQU     0x00080000          ; For 512KByte ARM Development Board
                                            ; For 2MByte, change to 0x200000
SVC_Stack       EQU     RAM_Limit           ; 256 byte SVC stack at top of memory
IRQ_Stack       EQU     RAM_Limit-256       ; followed by IRQ stack
; add FIQ_Stack, ABT_Stack, UNDEF_Stack here if you need them
USR_Stack       EQU     IRQ_Stack-256       ; followed by USR stack
ROM_Start       EQU     0x04000000          ; Base address of ROM after remapping
Instruct_2      EQU     ROM_Start + 4       ; Address of second instruction in ROM
ResetBase       EQU     0x0B000000          ; RPS Remap and Pause Controller
ClearResetMap   EQU     ResetBase + 0x20    ; Offset of remap control from base
         ENTRY
```

```
; --- Perform ROM/RAM remapping, if required
IF :DEF: ROM_RAM_REMAP

; On reset, an aliased copy of ROM is at 0x0.
; Continue execution from 'real' ROM rather than aliased copy
    LDR     pc, =Instruct_2
; Remap by writing to ClearResetMap in the RPS Remap and Pause Controller
    MOV     r0, #0
    LDR     r1, =ClearResetMap
    STRB    r0, [r1]
; RAM is now at 0x0.
; The exception vectors (in vectors.s) must be copied from ROM to the RAM
; The copying is done later by the C library code inside __main
ENDIF

    EXPORT  Reset_Handler

Reset_Handler
; --- Initialize stack pointer registers
; Enter SVC mode and set up the SVC stack pointer
    MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit ; No interrupts
    LDR     SP, =SVC_Stack
; Enter IRQ mode and set up the IRQ stack pointer
    MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
    LDR     SP, =IRQ_Stack
; Set up other stack pointers if necessary
; ...
; --- Initialize memory system
; ...
; --- Initialize critical IO devices
; ...
; --- Initialize interrupt system variables here
; ...
; --- Enable interrupts if required
; This is the earliest point at which interrupts may be safely enabled.
    MSR     CPSR_c, #Mode_SVC:OR:F_Bit ; Enable IRQ
; --- Now change to user mode and set up user mode stack.
    MSR     CPSR_c, #Mode_USR:OR:I_Bit:OR:F_Bit ; No interrupts
    LDR     SP, =USR_Stack
    IMPORT  __main
; --- Now enter the C code
; note use B not BL, because an application will never return this way
    B      __main
     END
```

The code in Example 7-4 implements a retarget layer for low-level I/O. Typically, this would contain your own target-dependent implementations of `fputc()`, `ferror()`, and so on. This example provides implementations of `fputc()`, `ferror()`, `_sys_exit()`, `_ttywrch()`, and `__user_initial_stackheap()`.

Semihosting SWIs are used to display text onto the console of the host debugger. This mechanism is portable across ARMulator, Angel, Multi-ICE and EmbeddedICE. `serial.c` is an alternative option that outputs characters from the serial port of an ARM Development (PID) Board. To use `serial.c`, add `#define USE_SERIAL_PORT` to the code or compile with `-DUSE_SERIAL_PORT`.

Example 7-4 retarget.c

```
/*Copyright (C) ARM Limited, 1999. All rights reserved. */
#include <stdio.h>
/* #define USE_SERIAL_PORT */
#ifdef __thumb
/* Thumb Semihosting SWI */
#define SemiSWI 0xAB
#else
/* ARM Semihosting SWI */
#define SemiSWI 0x123456
#endif

/* Write a character */
__swi(SemiSWI) void _WriteC(unsigned op, char *c);
#define WriteC(c) _WriteC (0x3,c)

/* Exit */
__swi(SemiSWI) void _Exit(unsigned op, unsigned except);
#define Exit() _Exit (0x18,0x20026)

struct __FILE { int handle;   /* Add whatever you need here */};
FILE __stdout;

extern void sendchar( char *ch );    /* in serial.c */

int fputc(int ch, FILE *f)
{
    char tempch=ch;
    /* Place your implementation of fputc here, for example write a character */
    /* to a UART, or to the debugger console with SWI WriteC */
#ifdef USE_SERIAL_PORT
    sendchar( &tempch );
#else
    WriteC( &tempch );
#endif
    return ch;
}

int ferror(FILE *f)
{   /* Your implementation of ferror */
    return EOF;
}

void _sys_exit(int return_code)
{
    Exit();         /* for debugging */
label:  goto label; /* endless loop */
}

void _ttywrch(int ch)
{
char tempch = ch;
#ifdef USE_SERIAL_PORT
    sendchar( &tempch );
#else
    WriteC( &tempch );
#endif
}

__value_in_regs struct R0_R3
     {unsigned heap_base, stack_base, heap_limit, stack_limit;}
    __user_initial_stackheap(unsigned int R0, unsigned int SP, unsigned int R2,
      unsigned int SL)
{
    struct R0_R3 config;
    config.heap_base = 0x00060000;
    config.stack_base = SP;
/*
To place heap_base directly above the ZI area, use:
    extern unsigned int Image$$ZI$$Limit;
    config.heap_base = (unsigned int)&Image$$ZI$$Limit;
    (or &Image$$region_name$$ZI$$Limit for scatterloaded images)

To specify the limits for the heap & stack, use e.g:
    config.heap_limit = SL;
    config.stack_limit = SL;
*/
    return config;
}
```

The code in Example 7-5 implements a simple polled RS232 serial driver for the ARM Development (PID) Board. It outputs single characters on Serial Port A at 9600 Baud, 8 bit, no parity, 1 stop bit. Initialize the port with `init_serial_A()` before calling `sendchar()`.

Example 7-5 serial.c

```
/* Copyright (C) ARM Limited, 1999. All rights reserved. */
#include "pid7t.h"
#include "nisa.h"
#include "st16c552.h"


void init_serial_A(void)
{
    *SerA_FCR = FCR_Fifo_Enable   |  /* Enable Tx and Rx FIFO Operation */
                FCR_Rx_Fifo_Reset |  /* Clear Rx FIFO and FIFO Counters */
                FCR_Tx_Fifo_Reset ;  /* Clear Tx FIFO and FIFO Counters */


    *SerA_MCR = 0;                  /* Switch Off loopback mode               */
    *SerA_LCR = LCR_Divisor_Latch ; /* Enable Baud Divisor Latch              */
    *SerA_DLL = DLL_9600_Baud ;     /* Set Divisor LSB value for 9600 baud    */
    *SerA_DLM = DLM_9600_Baud;      /* Set Divisor MSB value for 9600 baud    */
    *SerA_LCR = LCR_8_Bit_Word_1;   /* Set for 8-bit word length - 1 stop bit */
}


void sendchar( char *ch )
{
   while (!(*SerA_LSR & LSR_Tx_Hold_Empty))   /* Wait until Port A Tx FIFO
           {}                                   is empty */
   *SerA_THR = *ch;                           /* Transmit next character */
}
```

7.6 Using a simple scatter-loading file

Scatter loading provides a more flexible mechanism for mapping code and data onto your memory map than the armlink `-ro-base` and `-rw-base` options. These options are described in detail in the *ADS Tools Guide*.

This section shows a scatter-loaded version of the application in *Loading the ROM image at address 0* on page 7-14.

The scatter-loading description file, `scat_c.scf`, for this example is in *`install_directory`*`\Examples\rom\embed`.

7.6.1 Memory map

Figure 7-5 shows:

- ROM is fixed at 0x0 and is not remapped (see Figure 7-5)
- RAM is at `0x00040000` to hold the data, stack and heap.

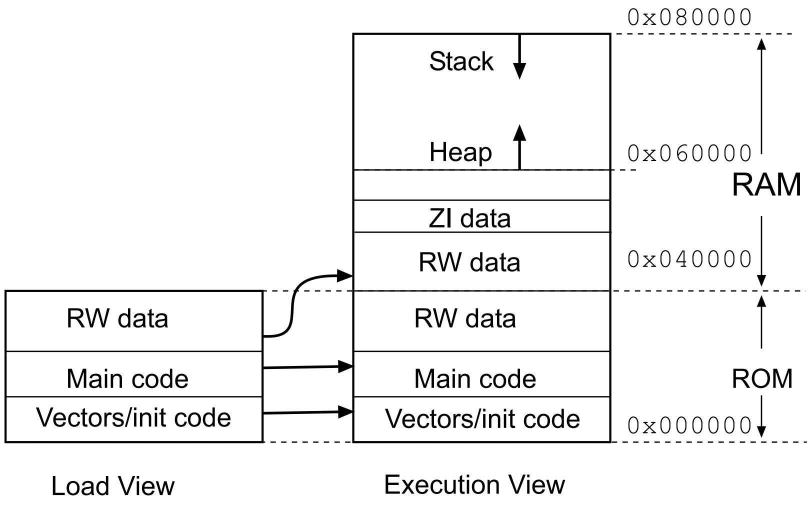

Figure 7-5 Memory map for simple scatter loading

7.6.2 Scatter-loading description file

The scatter-loading description file shown in Example 7-6 defines:

- one load region, ROM, at `0x0`.
- two execution regions:
  - ROM (at `0x0`) contains all the read-only code, including the library code. The exception vector table in `vectors.o` is placed first in this region. All other read-only code (`*`) is placed after `vectors.o`.
  - RAM (at `0x00040000`) contains the RW and ZI data regions for the application.

Example 7-6 scat_c.txt

```
ROM 0x0
{
    ROM 0x0
    {
        vectors.o (Vect, +First)
        * (+RO)
    }
    RAM 0x00040000
    {
        * (+RW,+ZI)
    }
}
```

7.6.3 Sample code

The C code for `main.c` is identical with the previous examples. This demonstrates how to use one code source to compile and link for different targets.

7.6.4 Building the example

To build the example, either:

- load the supplied `embed` project into the CodeWarrior IDE and select `Target=EmbeddedScatter`.
- use the `build_c.bat` batch file or a makefile containing the following (the indented lines are a continuation of the single line above):

```
armasm -g vectors.s
armasm -g init.s
armcc -g -01 -c main.c -DEMBEDDED
armcc -g -01 -c retarget.c
armcc -g -01 -c serial.c -I..\include
armlink vectors.o init.o main.o retarget.o serial.o
         -scatter scat_c.scf -o embed.axf
         -entry 0x0 -info totals -info unused
fromelf embed.axf -bin -o embed.bin
```

This creates:

- an ELF debug image (`embed.axf`) for loading into a debugger (AXD, ADW, ADU, or armsd)
- a binary ROM image (`embed.bin`) suitable for downloading into the memory of an ARM development board.

The use of `serial.c` is optional. Keep this line if you wish the output to be sent over the serial port of the development board.

7.7 Using both scatter-loading and remapping

This section shows how to convert the application in *Using a simple scatter-loading file* on page 7-23 into a more complex scatter-loading application. This example uses memory remapping to exchange the ROM and RAM regions after the application has started. The example also shows how to use two separate RAM areas (SSRAM and SRAM).

The code for this example is in `install_directory\Examples\rom\embed`.

7.7.1 Memory map

Figure 7-6 shows:

- FLASH is at `0x04000000`. An aliased copy of the FLASH appears at 0x0 on reset.
- After remapping, fast SSRAM is at `0x00000000` to hold the exception vectors and any exception handlers.
- After remapping, SRAM is at `0x00002000` for the storage of program variables.

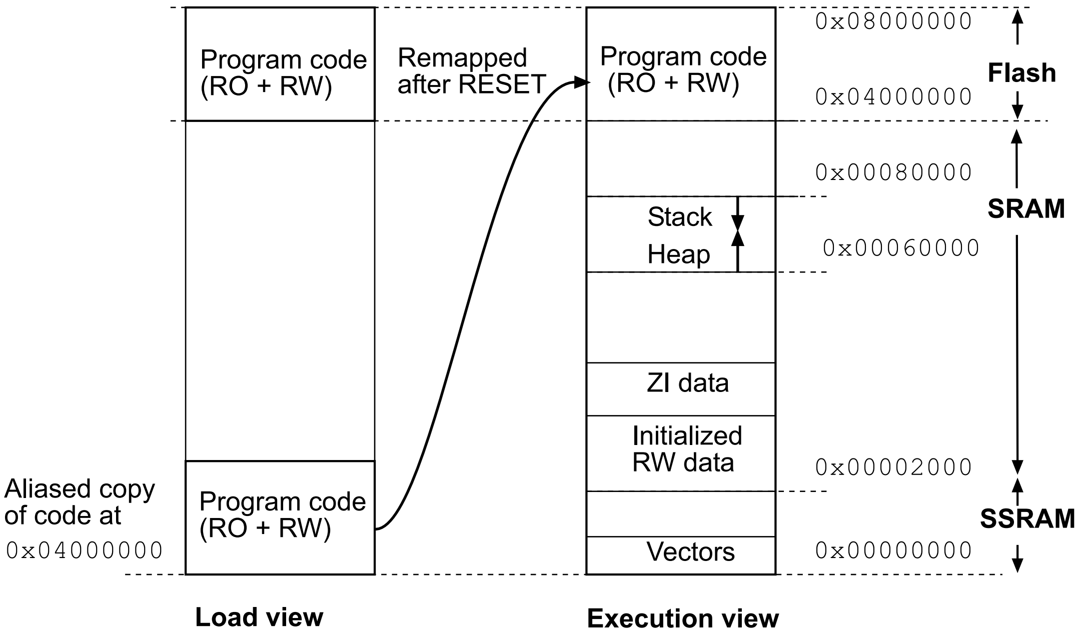

Figure 7-6 Memory map for remapping

7.7.2 Scatter-loading description file

The scatter-loading description file shown in Example 7-7 defines one load region (FLASH) and three execution regions:

- FLASH (at `0x04000000`) of size `0x80000`
- 32-bit SSRAM (at `0x00000000`)
- 16-bit SRAM (at `0x00002000`).

Example 7-7 scat_d.txt

```
FLASH 0x04000000 0x080000
{
    FLASH 0x04000000
    {
        init.o (Init, +First)
        * (+RO)
    }
    SSRAM 0x0000
    {
        vectors.o (Vect, +First)
    }
    SRAM 0x2000
    {
        * (+RW,+ZI)
    }
}
```

The program code and data is placed in Flash that resides at `0x04000000`. On reset, an aliased copy of Flash is remapped by hardware to address 0x0. Program execution starts at AREA `Init` in `init`.s. The `+First` option is used to place this code first in the image. After reset the first few instructions of `init.s` remap 32-bit RAM to address 0x0. The ARM Development (PID) Board remaps its Flash in this way.

Most of the RO code will execute from Flash. The RO execution address is the same as its load address (`0x04000000`), so it does not have to be moved.

SSRAM might be fast on-chip 32-bit RAM. Fast RAM is typically used for the stack and code that must be executed quickly. The exception vectors (AREA `Vect` in `vectors.s`) get relocated from Flash to 32-bit SSRAM at address `0x0` for speed. The `Vect` code is placed first in the region.

SRAM might be slower off-chip 16-bit DRAM. Slower RAM is typically used for less frequently accessed RW variables and ZI data. The RW data will get relocated from Flash to 16-bit RAM at `0x2000` The ZI data will be created in 16-bit RAM above the RW data.

7.7.3 Initialization code

Example 7-8 illustrates the use of initialization code (`init.s`) to perform ROM/RAM remapping. The portion of the initialization code that handles remapping is also listed:

Example 7-8 ROM/RAM remapping

```
; --- Perform ROM/RAM remapping, if required
IF :DEF: ROM_RAM_REMAP

; On reset, an aliased copy of ROM is at 0x0.
; Continue execution from 'real' ROM rather than aliased copy
    LDR     pc, =Instruct_2
; Remap by writing to ClearResetMap in the RPS Remap and Pause Controller
    MOV     r0, #0
    LDR     r1, =ClearResetMap
    STRB    r0, [r1]
; RAM is now at 0x0.
; The exception vectors (in vectors.s) must be copied from ROM to the RAM
; The copying is done later by the C library code inside __main
ENDIF
```

The initialization code in the C library copies the RO and RW execution regions from their load addresses to their execution addresses before creating any zero-initialized areas.

7.7.4 Building the example

To build the example, either:

- load the supplied `scatter` project into the CodeWarrior IDE
- use the `build_d.bat` batch file or a makefile containing the following (the indented lines are a continuation of the single line above):

```
armasm -g vectors.s
armasm -g -PD "ROM_RAM_REMAP SETL {TRUE}" init.s
armcc -c -g -O1 main.c -DEMBEDDED -DROM_RAM_REMAP
armcc -c -g -O1 retarget.c
armlink vectors.o init.o main.o retarget.o
        -scatter scat_d.scf -o embed.axf
        -info totals -entry 0x04000000
        -info unused
fromelf embed.axf -bin -o embed.bin
```

This creates:

- an ELF debug image (`embed.axd`) for loading into an ARM debugger
- a binary ROM image (`embed.bin`) suitable for downloading into the RAM or Flash memory of the ARM development boards.

The `readme.txt` file contains additional details of how the image can be downloaded to the Flash memory of an ARM Development Board and debugged there.

7.7.5 Additional examples of remapping

The *`install_directory`*`\Examples\rom\ledflash` directory contains a simple interrupt-driven LED flasher that runs on an ARM development board. It is derived from the LED example given in the *PID7T Example Code Suite*, but modified to use ROM/RAM remapping and scatter loading.

To build the example, a batch-file (`build.bat`) and CodeWarrior project file (`ledflash.mcp`) are provided. Full instructions for downloading the code to Flash are available in the directory.

See also *Using scatter loading with memory-mapped I/O* on page 7-37.

7.8 A semihosted application with interrupt handling

This section illustrates an *Reference Peripheral Specification* (RPS) based interrupt-driven timer, suitable for embedded applications. The `main()` function initializes and starts two RPS timers.

When a timer expires, an interrupt is generated. The interrupt is handled in `int_handler.c`. The code simply sets a flag and clears the interrupt. The interrupt flags are checked below in a endless loop. If a flag is set, a message is displayed and the flag is then cleared.

7.8.1 Memory map

There are no memory specification options in the linker command options and the default values are used. The code region starts at `0x00008000`. The RW data region and the ZI data region are placed sequentially after the code region. The stack top is `0x80000`.

7.8.2 Building the example

To build the example, either:

- load the supplied `rps_irq.mcp` project into the CodeWarrior IDE
- use a batch file or makefile containing the following:

```
armcc -c -g -O1 main.c -I..\include
armcc -c -g -O1 int_handler.c -I..\include
armlink main.o int_handler.o -o rps_irq.axf -info totals
```

7.8.3 Sample code

The code in Example 7-9 is compiled and linked on its own and executed in the semi-hosting environment and `Install_Handler` is called to install the interrupt vector. The code in Example 7-10 demonstrates an interrupt handler. The example can also be built as an embedded application with no semihosting (see *An embeddable application with interrupt handling* on page 7-35).

Example 7-9 Sample main.c code for rps_irq

```
/*
 * Copyright (C) ARM Limited, 1999. All rights reserved.
 */
#include <stdio.h>
#include <stdlib.h>
```

```
#include "stand.h"

int IntCT1 = 0;
int IntCT2 = 0;
int Count  = 0;
#ifndef EMBEDDED
    extern IRQ_Handler(void);
    unsigned *irqvec = (unsigned *)0x18;
    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)
    {
        printf ("Installation of Handler failed");
        exit(1);
    }
    vec = 0xea000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
#endif

/*
Enabling and disabling interrupts
Interrupts are enabled or disabled by reading the cpsr flags and updating bit 7.
These functions work only in a privileged mode, because the control bits of the
cpsr and spsr cannot be changed while in User mode.
*/

__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
    }
}
#ifdef EMBEDDED
  extern void init_serial_A(void);
#endif

int main(void)
{
#ifdef EMBEDDED
    __use_no_semihosting_swi(); /* ensure no functions that use semi-hosting SWIs
                                   are linked in from the C library */
    init_serial_A();            /* initialize serial A port */
#endif

    printf("RPS Timer Interrupt Example\n");
    printf("To execute this example under ARMulator, "
    printf("you MUST modify armul.cnf with:\n");
    printf("TimerEnabled=TRUE\n");
    printf("IntCEnabled=TRUE\n\n");

#ifdef EMBEDDED
  #ifdef ROM_RAM_REMAP
    printf("Embedded (ROM/RAM remap, no SWIs) version\n");
  #else
    printf("Embedded (ROM at 0x0, no SWIs) version\n");
  #endif
#else
    Install_Handler ((unsigned)IRQ_Handler, irqvec);

    printf("Normal (RAM at 0x8000, semi-hosting) version\n\n");
#endif

    printf("Initializing...\n");

    enable_IRQ();

    *IRQEnableClear = ~0;   /* Clear/disable all interrupts */

    *Timer1Control = 0;     /* Disable counters by clearing the control bytes */
    *Timer2Control = 0;
    *Timer1Clear = 0 ;       /* Clear counter/timer interrupts by writing to   */
    *Timer2Clear = 0 ;       /*  the clear register - any data will work       */

    *Timer1Load = FAST_LOAD;           /* Load counter values */
    *Timer2Load = MED_FAST_LOAD;

    *Timer1Control = (TimerEnable   |   /* Enable the Timer */
                      TimerPeriodic |   /* Periodic Timer producing interrupt */
                      TimerPrescale8 ); /* Set Maximum Prescale - 8 bits  */

    *Timer2Control = (TimerEnable   |   /* Enable the Timer          */
                      TimerPeriodic |   /* Periodic Timer producing interrupt */
                      TimerPrescale8 ); /* Set Maximum Prescale - 8 bits      */

    *IRQEnableSet = IRQTimer1 | IRQTimer2 ; /* Enable the timer interrupts */

    printf("Running...\n");

    IntCT1 = 0;                         /* Clear CT 1 Flag  */
    IntCT2 = 0;                         /* Clear CT 2 Flag  */
    Count  = 0;

    while ( Count < 20 )
    {
        if (IntCT1 != 0)        /* Timer 1 Interrupt occurred */
        {
            Count++;
            printf("IntCT1\n");
            IntCT1 = 0;         /* Reset the Timer 1 Interrupt Flag */
        }
        if (IntCT2 != 0)        /* Timer 2 Interrupt occurred */
        {
            Count++;
            printf("IntCT2\n");
            IntCT2 = 0;         /* Reset the Timer 2 Interrupt Flag */
        }
    }

    disable_IRQ();
}
```

Example 7-10 Sample int_handler.c code

```
/*
 * Copyright (C) ARM Limited, 1999. All rights reserved.
 */
#include "stand.h"
/*******************************************************************************
* IRQ_Handler                                                                  *
* This function handles IRQ interrupts.  In this example, these may come from *
* Timer 1 or Timer 2                                                           *
* This handler simply clears the interrupt and sets corresponding flags.      *
* These flags are then checked by the main application.                      *
*******************************************************************************/

void __irq IRQ_Handler(void)
{
    unsigned status;

    status = *IRQStatus;
    /* Deal with source of interrupt */

    if (status & IRQTimer1)
    {
        *Timer1Clear = 0;/* clear the interrupt */
        IntCT1++;        /* set the flag        */
    }
    else
    if (status & IRQTimer2)
    {
        *Timer2Clear = 0;/* clear the interrupt */
        IntCT2++;        /* set the flag        */
    }
}
```

7.9 An embeddable application with interrupt handling

This section describes how to convert the application in *A semihosted application with interrupt handling* on page 7-30 into an embeddable application. Converting the application requires four additional files:

`vectors.s` This file contains exception vectors and exception handlers. For this example ROM is fixed at `0x0`.

`init.s` This file performs ROM/RAM remapping (if required), initializes stack pointers and interrupts for each mode, and branches to `__main` in the C library. The C library code at `__main` eventually calls `main()`.

ROM/RAM remapping is not used in this example. A sample scatter load description for remapping is available in `*install_directory*\Examples\ROM\rps_irq`.

`retarget.c` This file implements a retarget layer for low-level I/O. Typically, this would contain your own target-dependent implementations. This example provides implementations of `fputc()`, `ferror()`, `_sys_exit()`, `_ttywrch()` and `__user_initial_stackheap()`.

The `#define USE_SERIAL_PORT` selects code to output characters from the serial port of an ARM Development (PID) Board.

`serial.c` This file implements a simple polled RS232 serial driver for the ARM Development (PID) Board. It outputs single characters on Serial Port A at 9600 Baud, 8 bit, no parity, 1 stop bit.

To ensure that no semi-hosting SWI-using function is linked in from the C library, `__use_no_semihosting_swi()` is called from `main()`.

7.9.1 Memory map

The scatter-loading descriptor file defines one load region, `ROM`, and two execution regions, `ROM` and `RAM` (the memory map is the same as displayed in Figure 7-5). The entire program is placed in ROM. The RO code will execute from `ROM`. The execution address of `ROM` is the same as its load address (`0x0`), so it does not have to be moved.

The exception vector table `vectors.s` must appear at `0x0`, so the `+First` command is used to place it first in the image. The RW data is relocated from ROM to RAM at `0x00040000`. The ZI data is initialized in RAM above the RW data.

7.9.2 Building the example

To build the example, use the `build_c.bat` batch file, the CodeWarrior IDE project file `rps_irq.mcp` with a target of **EmbedScatter**, or a makefile containing the following (the indented lines are a continuation of the single line above):

```
armasm -g vectors.s
armasm -g init.s
armcc -c -g -O1 main.c -DEMBEDDED -I..\include
armcc -c -g -O1 retarget.c
armcc -c -g -O1 serial.c -I..\include
armcc -c -g -O1 int_handler.c -I..\include
armlink vectors.o init.o main.o retarget.o serial.o
       int_handler.o -scatter scat_c.scf -o rps_irq.axf
       -entry 0x0 -info totals
fromelf rps_irq.axf -bin -o rps_irq.bin
```

7.9.3 Scatter-loading description file

The scatter file is equivalent to linking with `armlink -ro-base 0x0 -rw-base 0x00040000`.

```
ROM 0x0
{
    ROM 0x0
    {
        vectors.o (Vect, +First)
        * (+RO)
    }
    RAM 0x00040000
    {
        * (+RW,+ZI)
    }
}
```

7.9.4 Sample code

The retargetting code is the same as the code used in *Loading the ROM image at address 0* on page 7-14. The source is available in `*install_directory*\Examples\ROM\rps_irq`.

7.10 Using scatter loading with memory-mapped I/O

In most ARM embedded systems, peripherals are located at specific addresses in memory. You often need to access a memory-mapped register in a peripheral by using a C variable. In your code, you will need to consider not only the size and address of the register, but also its alignment in memory.

7.10.1 Using pointers to access I/O

The simplest way to implement memory-mapped variables is to use pointers to fixed addresses. If the memory is changeable by external factors, for example by some hardware, it must be labelled as **`volatile`**. Consider a simple example:

```
volatile unsigned *port = (unsigned int *) 0x40000000;
```

The data on the port can be accessed by:

```
*port = value;    /* write to port */
value = *port;    /* read from port */
```

The use of **`volatile`** ensures that the compiler always carries out the memory accesses, rather than optimizing them out. If the access was in a loop and the variable was not **`volatile`**, only one read of the memory address would be done.

This approach can be used to access 8, 16 or 32 bit registers, but you must declare the variable with the appropriate type for its size, **`int`** for 32-bit registers, **`short`** for 16-bit, and **`char`** for 8-bit. The compiler will then generate the correct single load/store instructions, `LDR`/`STR`, `LDRH`/`STRH`, or `LDRB`/`STRB`.

You must also ensure that the memory-mapped registers lie on appropriate address boundaries. Alignment must be either all word-aligned or on their natural size boundaries. The natural size of 16-bit registers is on half-word addresses. ARM recommends that all registers, whatever their size, be aligned on word boundaries, see *Alignment of registers* on page 7-39.

You can use **`#define`** to simplify your code. For example, the source code in Example 7-11 produces the interleaved code in Example 7-12.

Example 7-11

```
#define PORTBASE  0x40000000    /* Counter/Timer Base */
#define PortLoad  ((volatile unsigned int *) PORTBASE)          /* 32 bits */
#define PortValue ((volatile unsigned short *)(PORTBASE + 0x04)) /* 16 bits */
#define PortClear ((volatile unsigned char *)(PORTBASE + 0x08))  /*  8 bits */

void init_regs(void)
{
    unsigned int int_val;
    unsigned short short_val;
    unsigned char char_val;
    *PortLoad = (unsigned int) 0xF00FF00F;
     int_val = *PortLoad;
    *PortValue = (unsigned short) 0x0000;
     short_val = *PortValue;
    *PortClear = (unsigned char) 0x1F;
     char_val = *PortClear;
}
```

Example 7-12 Output fragment from compiler using -S and -fs

```
;;;11      *PortLoad = (unsigned int) 0xF00FF00F;
                  init_regs PROC
000000  e59f1024       LDR      a2,|L1.44|
000004  e3a00440       MOV      a1,#0x40000000
000008  e5801000       STR      a2,[a1,#0]
;;;12          int_val = *PortLoad;
00000c  e5901000       LDR      a2,[a1,#0]
;;;13         *PortValue = (unsigned short) 0x0000;
000010  e3a01000       MOV      a2,#0
000014  e1c010b4       STRH     a2,[a1,#4]
;;;14          short_val = *PortValue;
000018  e1d010b4       LDRH     a2,[a1,#4]
;;;15         *PortClear = (unsigned char) 0x1F;
00001c  e3a0101f       MOV      a2,#0x1f
000020  e5c01008       STRB     a2,[a1,#8]
;;;16          char_val = *PortClear;
000024  e5d00008       LDRB     a1,[a1,#8]
;;;17     }
```

```
000028  e1a0f00e        MOV      pc,lr
                    |L1.44|
00002c  f00ff00f        DCD      0xf00ff00f
                        ENDP
;;;18
```

ARM recommends word alignment of peripheral registers even if they are 16-bit or 8-bit peripherals. In a little-endian system, the peripheral databus can connect directly to the least significant bits of the ARM databus and there is no need to multiplex (or duplicate) the peripheral databus onto high bits of the ARM databus. In a big-endian system, the peripheral databus can connect directly to the most significant bits of the ARM databus and there is no need to multiplex (or duplicate) the peripheral databus onto low bits of the ARM databus.

The AMBA APB bridge uses this technique to simplify the bridge design. The result is that only word-aligned addresses should be used (whether byte, halfword or word transfer), and a read will read garbage on any bits that are not connected to the peripheral.

If a 32-bit word is read from a 16-bit peripheral, the top 16 bits of the register value must be cleared before use. For example, to access some 16-bit peripheral registers on 16-bit alignment, you might write:

```
volatile unsigned short u16_IORegs[20];
```

For little-endian systems, this works if your peripheral controller can route the peripheral databus to the high part (D31..D16) of the ARM databus as well as the low part (D15..D0) depending upon the address that you are accessing. You should check if this multiplexing logic exists in your design (the standard ARM APB bridge does not support this).

7.10.2 Alignment of registers

If you wish to map 16-bit registers on 32-bit alignment as recommended, then you could use a **`short`** or **`int`** array.

Using an array of shorts

If you use shorts, you can access registers at even numbered addresses by declaring.

```
volatile unsigned short u16_IORegs[40];
```

The array element number is double the register number. For example to access register 4 you could use:

```
x = u16_IORegs[8];
u16_IORegs[8] = newval;
```

Using an array of ints

Access the registers as 32-bit by declaring:

```
volatile unsigned int u32_IORegs[20];
```

A peripheral controller such as the AMBA APB bridge will read garbage into the top bits of the ARM register from the signals that are not connected to the peripheral (D31 to D16 for a little-endian system). So, when such a peripheral is read, it must be cast to an **`unsigned short`** to get the compiler to discard the upper 16 bits. For example, access r4 using:

```
x = (unsigned short) u32_IORegs[4];
u32_IORegs[4] = newval;
```

Using a struct

The advantages of using a struct over an array are:

- descriptive names can be used (more maintainable and legible)
- different register widths can be accommodated.

Padding should be made explicit rather than relying on automatic padding added by the compiler, for example:

```
struct PortRegs {
  unsigned short ctrlreg;  /* offset 0 */
  unsigned short dummy1;
  unsigned short datareg;  /* offset 4 */
  unsigned short dummy2;
  unsigned int data32reg;  /* offset 8 */
} iospace;

x = iospace.ctrlreg;
iospace.ctrlreg = newval;
```

——— **Note** ———

Peripheral locations should *not* be accessed using **`__packed`** structs (where unaligned members are allowed and there is no internal padding), or using C bitfields. This is because it is not possible to control the number and type of memory access that is being performed by the compiler.

The result is code that is non-portable, has undesirable side effects, and will not work as intended. The recommended way of accessing peripherals is through explicit use of architecturally-defined types such as **`int`**, **`short`**, **`char`** on their natural alignment.

7.10.3 Mapping variables to specific addresses

Memory mapped registers can be accessed from C in two efficient ways:

- by forcing an array or struct variable to a specific address
- by using a pointer to an array or struct.

Forcing a struct or array to a specific address

The variable should be declared it in a file on its own. When it is compiled, the object code for this file will only contain data. This data can be placed at a specified address using the ARM scatter-loading mechanism. This is the recommended method for placing regions at required locations in the memory map.

Create a file, for example `iovar.c` that contains a declaration of the variable, array, or struct. For example:

```
volatile unsigned short u16_IORegs[20];
```

or

```
struct{
   volatile unsigned reg1;
   volatile unsigned reg2;
} mem_mapped_reg;
```

Create a scatter load description file, called for example `scatter.txt`, containing the following:

```
ALL 0x8000      ; one load region ALL at 0x8000
{
     ALL 0x8000 ; by default, everything goes into this region
     {
         * (+RO,+RW,+ZI)
     }
}

IO  0x40000000             ; region for variables
{
     IO  0x40000000 UNINIT ; register variables go here
```

```
                            ; initial zeros are not written
     {
         iovar.o (+ZI)      ; a single module is selected by name
     }
}
```

The scatter load file must be specified to the linker using the `-scatter scatter.txt` command-line option. The `UNINIT` keyword in the description file indicates that the ZI region will not be initialized with zeros when the application is reset. If you want the peripheral registers to have zero written to them on reset, omit the `UNINIT` keyword. The scatter load file creates two different regions in your image (`ALL` and `IO`). The zero-init area from `iovar.o` (containing your array) goes into the `IO` area located at `0x40000000`. All code (RO) and data areas (RW and ZI) from other object files go into the `ALL` region that starts at `0x8000`.

If you have more than one group of variables (more than one set of memory mapped registers) you must define each group of variables as a separate execution region (they could, however, all lie within a single load region). Each group of variables must be defined in a separate module.

The benefits of using a scatter description file are:

- All the (target-specific) absolute addresses chosen for your devices, code and data are located in one file and maintenance is simplified.
- If you decide to change your memory map (for example if peripherals are moved), you do not have to rebuild your entire project but only to re-link the existing objects.

For a description of scatter loading, see the linker chapter in the *ADS Tools Guide*

Using a pointer to struct/array

```
struct PortRegs {
  unsigned short ctrlreg;  /* offset 0 */
  unsigned short dummy1;
  unsigned short datareg;  /* offset 4 */
  unsigned short dummy2;
  unsigned int data32reg;  /* offset 8 */
};
volatile struct PortRegs *iospace =
                (struct PortRegs *)0x40000000;
x = iospace->ctrlreg;
iospace->ctrlreg = newval;
```

The pointer could be either local or global. If you want the pointer to be global in order to avoid the base pointer being reloaded after function calls, make `iospace` a constant pointer to the struct by changing its definition to:

```
volatile struct PortRegs * const iospace =
    (struct PortRegs *)0x40000000;
```

7.10.4 Code efficiency

The ARM compiler will normally use a base register plus the immediate offset field available in the load/store instruction to compile struct member or specific array element access.

The ARM instruction set, `LDR`/`STR` word/byte have a 4Kbyte range, but `LDRH`/`STRH` has a smaller immediate offset of 256bytes.

The Thumb instruction set is much more restricted in addressing range than the ARM instructions. The Thumb `LDR`/`STR` has a range of 32 words, `LDRH`/`STRH` has a range of 32 halfwords, `LDRB`/`STRB` has a range of 32 bytes. You must group related peripheral registers near to each other if possible. The compiler will generally do a good job of minimizing the number of instructions required to access the array elements or structure members by using base registers.

There is a choice between one big C struct/array for the whole I/O space and smaller per-peripheral structs. There is not much difference in efficiency. The big struct might be a benefit if you are using ARM code where a base pointer can have a 4Kbyte range (for word/byte access) and the entire I/O space is less than 4Kbyte. Smaller structs for each peripheral are more maintainable.

7.11 Troubleshooting

This section provides solutions to the following common problems:

- *Linker error __semihosting_swi_guard* on page 7-44
- *Replacing the Write0() SWI call* on page 7-44
- *Setting $top_of_memory* on page 7-44.

7.11.1 Linker error __semihosting_swi_guard

The linker reports `__semihosting_swi_guard` as being multiply defined.

Cause

The linker loaded the semihosting implementation of a function from the ANSI C library. If you have called the guard function `use_no_semihosting_swi()` and have also called a library function that uses semihosting, you will get this error.

Solution

This problem can be fixed in one of the following ways:

- If the semihosted functions are used only when building an application version of your ROM image for debugging purposes, comment them out with an `#ifdef` when building a ROM image.
- Redefine the semihosted functions with your own implementation. The new functions will be used instead of the C library versions.

7.11.2 Replacing the Write0() SWI call

Users of EmbeddedICE 2.04 or earlier might find problems with the semihosting SWI `SYS_WRITE0`, used by the examples in this chapter to print to the debugger console. You should upgrade your EmbeddedICE to the latest ICE agent, currently 2.07, or upgrade to Multi-ICE to remedy this problem.

7.11.3 Setting $top_of_memory

The debugger internal variable `$top_of_memory` tells Multi-ICE and EmbeddedICE where the highest writable address is in the memory map of a remote target. This address is used to place the stack and heap. The default value for `$top_of_memory` is `0x80000`, to match the (unexpanded) ARM Development (PID) Board.

Different boards may have different memory maps, so `$top_of_memory` must be changed to one plus the address of the top of the RAM for your board. This must be done before running an application, otherwise you may experience data aborts or crashes.

For the ARM Development (PID) Board with extra DRAM modules fitted, you should change `$top_of_memory` appropriately.

For the ARM Evaluation Board (AEB), reset the board after connecting Multi-ICE and then set `$top_of_memory` to `0x20000`. To avoid damaging the AEB, do *not* attempt to connect and disconnect Multi-ICE without first removing power to the AEB board.

`$top_of_memory` only applies to Multi-ICE and EmbeddedICE. It does not apply to Angel. (The top of memory for Angel is hard-coded in the porting).

7.12 Measuring code and data size

To measure code size, do not look at the linked image size or object module size, as these include symbolic information that is not part of the binary data. Instead, use one of the following armlink options:

`-info sizes`	This option gives a breakdown of the code and data sizes of each object file or library member making up an image.
`-info totals`	This option gives a summary of the total code and data sizes of all object files and all library members making up an image

7.12.1 Interpreting size information

The information provided by the `-info sizes` and `-info totals` options can be broken down into:

- code (or read-only) segments
- data (or read-write) segments
- debug data.

Code (or read-only) segments

`code size`	Size of code, excluding any data that has been placed in the code segment.
`RO data`	Size of read-only data included in the code segment by the compiler. Typically, this data contains the addresses of variables that are accessed by the code, plus any floating-point immediate values or immediate values that are too big to load directly into a register. It does not include inline strings (these are listed separately).

Data (or read-write) segments

`RW data`	Size of read-write data. This is data that is read-write and also has an initializing value. `Read-write data` occupies the displayed amount of RAM at runtime, but also requires the same amount of ROM to hold the initializing values that are copied into RAM on image startup.
`ZI data`	Size of read-write data that is zero-initialized at image startup. Typically this contains arrays that are not initialized in the C source code. Zero-initialized data requires the displayed amount of RAM at runtime but does not require any space in ROM.

Debug data

`debug data`	Reports the size of any debugging data if the files are compiled with the `-g` option.

Note

There are totals for the debug data, even though the code has not been compiled for source-level debugging, because the compiler automatically adds information to an AIF file to allow stack backtrace debugging.

7.12.2 Calculating ROM and RAM requirements

The linker calculates the ROM and RAM requirements for code and data as follows:

ROM	`Code size + RO data + RW data`
RAM	`RW Data + ZI data`

In addition you must allow some RAM for stacks and heaps.

In more complex systems, you may require part (or all) of the code segment to be downloaded from ROM into RAM at runtime. This increases the system RAM requirements but could be necessary if, for example, RAM access times are faster than ROM access times and the execution speed of the system is critical.

Glossary

ADS	See *ARM Developer Suite*.
ADU	See *ARM Debugger for UNIX*.
ADW	See *ARM Debugger for Windows*.
ANSI	American National Standards Institute. An organization that specifies standards for, among other things, computer software.
Angel	Angel is a program 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*.

ARM Debugger for UNIX	*ARM Debugger for UNIX* (ADU) and *ARM Debugger for Windows* (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively. This debugger was issued originally as part of the *ARM Software Development Toolkit*. It is still fully supported and is now supplied as part of the *ARM Developer Suite*.
ARM Debugger for Windows	*ARM Debugger for Windows* (ADW) and *ARM Debugger for UNIX* (ADU) are two versions of the same ARM debugger software, running under Windows or UNIX respectively. This debugger was issued originally as part of the *ARM Software Development Toolkit*. It is still fully supported and is now supplied as part of the *ARM Developer Suite*.
ARM Developer Suite	A suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors.
ARM eXtendable Debugger	The *ARM eXtendable Debugger* (AXD) is the latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
armsd	The *ARM Symbolic Debugger* (armsd) is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. It is a command-line debugger that runs on all supported platforms.
ATPCS	ARM and Thumb Procedure Call Standard defines how registers and the stack will be used for subroutine calls.
AXD	See *ARM eXtendable Debugger*.
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	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.
Debugger	An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.
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
Environment	The actual hardware and operating system that an application will run on.
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.
Host	A computer which provides data and other services to another computer.
ICE	In Circuit Emulator.
IDE	Integrated Development Environment (Code Warrior).
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 may 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 sections
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
Interworking	Producing an application that uses both ARM and Thumb code.
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	Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.
MMU	See *Memory Management Unit*.
Multi-ICE	Multi-processor in-circuit emulator. 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 will be 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 card. *See also* RWPI
PIE	A platform-independent evaluator card designed and supplied by ARM Ltd.
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code. *Call-graph profiling* provides great detail but slows execution significantly. *Flat profiling* provides simpler statistics with less impact on exectution speed. For both types of profiling you can specify the time interval between statistics-collecting operations.
Program image	See Image.
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 allow RAM to replace ROM once 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	Read Only Position Independent. Code and read-only data addresses can be changed at run-time.
RTOS	Real Time Operating System.
RWPI	Read Write Position Independent. Read/write data addresses can be changed at run-time.
Scatter loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.
Scope	The accessibility of a function or variable at a particular point in the application code. Symbols which have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.
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.
SWI	Software Interrupt. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.
Target	The actual target processor, (real or simulated), on which the application is running. The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.
Thread	A context of execution on a processor. A thread is always related to a processor and may or may not be associated with an image.
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.
Watchpoint	A location within the image which will be monitored and which will cause execution to break when it changes.
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.